Хмельницький національний університет
Факультет програмування
та комп'ютерних і телекомунікаційних систем
Кафедра інженерії програмного забезпечення

ДИПЛОМНИЙ ПРОЕКТ

Програмне забезпечення для автоматизації обміну замовленнями між різними сервісами таксі

Назва теми

Рівень вищої освіти Перший (бакалаврський)

Галузь знань 12 «Інформаційні технології»

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

Шифр ДППЗ.170113.01.12.00

Виконав студент IV курсу група ІПЗ-17-1 _____ І. С. Мисик
Підпис Ініціали, прізвище

Керівник д-р фіз.-мат. наук, професор _____ Л.П. Бедратюк
Науковий ступінь, звання Підпис Ініціали, прізвище

Нормоконтролер канд. техн. наук, доцент _____ Г. І. Радельчук
Підпис Ініціали, прізвище

До захисту допускаю:
Завідувач кафедри інженерії
програмного забезпечення _____ Л. П. Бедратюк
Підпис Ініціали, прізвище

9 06 2021 р.

Хмельницький 2021

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Програмування та комп'ютерних і телекомунікаційних систем

Кафедра Інженерії програмного забезпечення

Освітній рівень бакалавр

Галузь знань 12 «Інформаційні технології»

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ:
Завідувач кафедри _____
Л. П. Бедратюк
05. 02 20__

ЗАВДАННЯ

НА ДИПЛОМНИЙ ПРОЕКТ

Мисик Іван Сергійович
Прізвище, ім'я, по батькові студента

1. Тема проекту Програмне забезпечення для автоматизації обміну замовленнями між різними сервісами таксі

Керівник проекту Бедратюк Леонід Петрович, доктор фізико-математичних наук, професор
Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджено наказом ректора університету від 05.02 2021 р. №11

2. Строк подання студентом проекту на кафедру: 01.06.2021 р.

3. Вихідні дані до проекту Матеріали переддипломної практики

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)
Дослідження предметної області та постановка задачі, проектування програмного забезпечення, програмна реалізація, тестування програмної системи.

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень)
Презентаційні матеріали (слайди, 13 шт.)

6. Консультанти розділів дипломного проекту

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Радельчук Г. І. доцент кафедри ІПЗ		
Антиплагіат	Гурман І. В. доцент кафедри ІПЗ		

7. Дата видачі завдання 05 лютого 2021 р.

КАЛЕНДАРНИЙ ПЛАН

Назва етапів (розділів) дипломного проекту	Строк виконання етапів проекту	Примітка
1 Ознайомлення з тематикою дипломного проектування (ДП), визначення та узгодження індивідуальних там ДП	01.12 – 30.12.2020	
2 Дослідження предметної області, в якій планується використання програмного засобу (ПЗ), визначення задач та вимог, розробка технічного завдання	02.01 – 31.01.2021	
3 Проектування програмного забезпечення	01.02 – 28.02.2021	
4 Програмна реалізація	01.03 – 10.04.2021	
5 Тестування програмного забезпечення	11.04 – 30.04.2021	
6 Написання вступу, загальних висновків, оформлення джерел посилання та додатків. Оформлення пояснювальної записки ДП згідно вимог стандартів	01.05 – 25.05.2021	
7 Попередній захист ДП	Травень 2021 (згідно графіка)	
8 Перевірка ДП на плагіат, нормоконтроль, отримання відгуків та рецензій. Брошурування (зшиття) пояснювальної записки)	26.05 – 30.05.2021	
9 Підготовка до захисту та захист ДП	з 01.06.2021	

Студент _____ І. С. Мисик
Підпис Ініціали, прізвище

Керівник проекту _____ Л. П. Бедратюк
Підпис Ініціали, прізвище

АНОТАЦІЯ

Тема дипломного проекту: «Програмне забезпечення для автоматизації обміну замовленнями між різними сервісами таксі».

Автор проекту: Мисик Іван Сергійович.

Керівник проекту: Бедратюк Леонід Петрович.

Пояснювальна записка: 93 с., 26 рис., 6 табл., 4 дод., 17 джерел.

Графічна частина: 13 презентаційних слайдів.

REST, WEB API, POSTGRESQL, JAVA, SPRING FRAMEWORK, CUNCURRENT, HIBERNATE, SWAGGER.

Метою проекту є створення сервісу для обміну таксі замовлень, який дозволятиме різним таксі компаніям, абсолютно автоматично, підтримувати зв'язок для підвищення швидкості знаходження машини для клієнта в межах певного міста.

Дипломний проект містить специфіку систем обміну замовленнями між різними сервісами таксі. Були проаналізовані типи архітектури а саме клієнт-серверна. Визначене наявне програмне забезпечення з чого були виявлені всі функціональні та нефункціональні характеристики. Проаналізовані системи керування базами даних, визначені їх переваги та недоліки. Обрані інструменти реалізації програмної системи. Реалізовано програмну систему автоматизації обміну замовленнями між різними сервісами таксі.

При реалізації системи було застосовано мову програмування Java на базі фреймворку Spring. Для керування даними було використано PostgreSQL. Для представлення даних на сервері використано ORM фреймворк Hibernate.

Таким чином, результатом виконання дипломного проекту є реалізований програмний продукт, що об'єднує служби таксі в єдину мережу для пришвидшення пошуку водія по замовленню в межах певного міста.

01.06.21р
Дата

Підпис

ВДОМІСТЬ ДОКУМЕНТІВ

№ рядка	Формат	Позначення документа	Найменування документа	К-сть аркушів	№ екз.	Примітка
			Текстові документи			
1	А4	ДППЗ.170113.01.12.00	Пояснювальна записка	93		
2	А4		Завдання на дипломний	1		
			проект			
3	А4		Анотація	1		
			Графічні документи			
4	А4		Презентаційні матеріали	13		

Змн.	Арк.	№ докум	Підпис	Дата	ДППЗ.170113.01.12.ВД		
Виконав		Мисик І. С.			Програмне забезпечення для автоматизації обміну замовленнями між різними таксі сервісами	Літера	Арк 1 / Аркушів 1
Керівник		Бедратюк Л.П.					
Н. Контр.		Рабельчук Г. І.		8.06	Відомість документів	ХНУ, ІПЗ-17-1	
Зав. Каф.		Бедратюк Л. П.					

ЗМІСТ

ВСТУП

Сьогодні сфера обслуговування людей у галузі таксі має велику популярність, яка росте за рахунок якісного обслуговування та легкого доступу. Адже на сучасному етапі розвитку людства існує доступ до різноманітних варіантів реалізації власного бізнесу таксі і не тільки. Загалом таксі сервіс, в технічному плані, складається з серверу, мобільного додатку водія, мобільного додатку клієнта, панель адміністратора та робочої частини оператора. Зараз, щоб увійти у ринок таксі послуг, потрібен саме такий набір.

Але яке б нове, інноваційне, оригінальне програмне забезпечення не було у сервісу таксі завжди присутній людський вплив. Сучасний таксі сервіс як і десять років тому не може існувати без людського контролю. Завжди є присутній технічний адміністратор або оператор який контролює хід роботи.

Актуальність теми полягає в тому, що кожен сервіс, коли виходить на ринок з часом стикається з різними факторами, що погано впливають на популярність компанії. Одним з таких факторів є нехватка кадрів. Водії постійно у пошуках більшого заробітку або комфортніших умов, через що міняють компанію на більш зручних правилах. Дане програмне забезпечення дозволить об'єднати служби таксі в єдину мережу, за рахунок чого не буде важливо в якій службі таксі знаходиться водій. На даний момент існує дуже мало аналогів даної системи, які є відкритими. В більшості випадків вони є приватними і зафіксованими за одним сервісом таксі.

Метою проекту є створення сервісу для обміну таксі замовлень, який дозволятиме різним таксі компаніям, абсолютно автоматично, підтримувати зв'язок для підвищення швидкості знаходження машини для клієнта в межах певного міста.

Для вирішення даної проблеми потрібно виконати наступні завдання:

- виконати аналіз сучасних сервісів таксі;
- визначити модель роботи таксі сервісу;
- розробити модель сервісу для обміну замовленнями на основі роботи

таксі сервісів, що наразі працюють;

- розробити архітектуру сервісу для обміну замовленнями;
- розробити алгоритм розподілення замовлення між сервісами;
- виконати програмну реалізацію сервісу для обміну замовленнями.

Результатом дипломного проекту є програмне забезпечення для автоматизації обміну замовленнями між різними таксі сервісами.

1 ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Змістовний аналіз предметної області, її структурних та функціональних особливостей

Агрегатор таксі, в сучасному розумінні, є транспортною компанією, онлайн сервісом, що дозволяє клієнту знайти водія не обираючи конкретну службу таксі. Оригіналне визначення агрегатору звучить так - компанія, що займається встановленням множинних домовленостей з окремими контент- і сервіс-провайдерами, а також з операторами для полегшення процесу організації доставки контенту його споживачам — абонентам. Але в нашому випадку буде виведено зовсім інше поняття, а в висновку і реалізацю.

Агрегатор замовлень таксі (Біржа замовлень) – це сервіс, що надає послуги розподілу замовлень між зареєстрованим таксі сервісам. Тобто, агрегатор взаємодіє не з клієнтом а з таксі компаніями. Компанія таксі може зареєструватись до агрегатора вказавши домен свого серверу та підключивши REST API до власного сервісу [1].

Для того, щоб визначити структуру та функціональні можливості агрегатору потрібно провести аналіз роботи звичайного сервісу таксі. В сучасних реаліях посередня компанія таксі арендує програмний комплекс.

Така програмна система загалом складається з наступних частин:

а) сервер або Backend. Він регулює весь процес передачі замовлення від клієнта до водія;

б) робоча частина оператора. Роль данного веб застосунку полягає у відображенні всіх процесів що відбуваються з замовленнями, водіями та клієнтами. Оператори через дане програмне забезпечення можуть приймати замовлення від клієнтів, вирішувтаи не стандартні проблеми та зв’язуватись з водіями для вирішення різних питань;

в) панель адміністратора. До неї має доступ власник компанії або довірена

людина. Такий веб застосунок дає можливість повністю налаштувати та підтримувати таксі сервіс, а саме:

1) вказати тарифи;
2) розподілити права доступу;
3) вказати налаштування роботи додатків (якщо такі є):
   – оператор;
   – клієнт;
   – водій;
4) налаштувати розподілення замовлень;
5) розграфити карту на сектора та вказати межі роботи сервісу таксі;

г) мобільний додаток водія. за допомогою нього водій може:

1) контролювати свій дохід від сервісу таксі;
2) приймати та виконувати замовлення;
3) зв’язуватись з операторами, клієнтами та іншими водіями;

д) мобільний додаток клієнта. Його наявність є опціональною але зараз він присутній майже в кожній службі таксі. Його головна роль заключається в наданні можливості клієнту замовити таксі.

Якщо викоремити головне то вийде такий список:

а) вказати початкову та кінцеві точки;
б) обрати тариф;
в) обрати послуги:
   1) порожній багажник;
   2) клієнт з обмеженою дієздатністю;
   3) перевізка тварин;
   4) різні типи оплати;
г) спостерігати за тим де знаходиться водій;
д) відмінити замовлення.

Робота даного комплексу зводитться до виконання замовлень. Узагальнений процес роботи сервісу таксі можна побачити на рисунку 1.1. Такий процес наочно показує, що для виконання замовлення потрібні модулі різних типів, які можуть

виконувати прості дії по типу збереження замовлення до бази даних або розсилання до різних сервісів таксі. Кожна дія є транзакційною і незалежною, що підвищує надійність роботи системи. Слідом такі дії складають алгоритм, що вирішує задачу виконання замовлення.

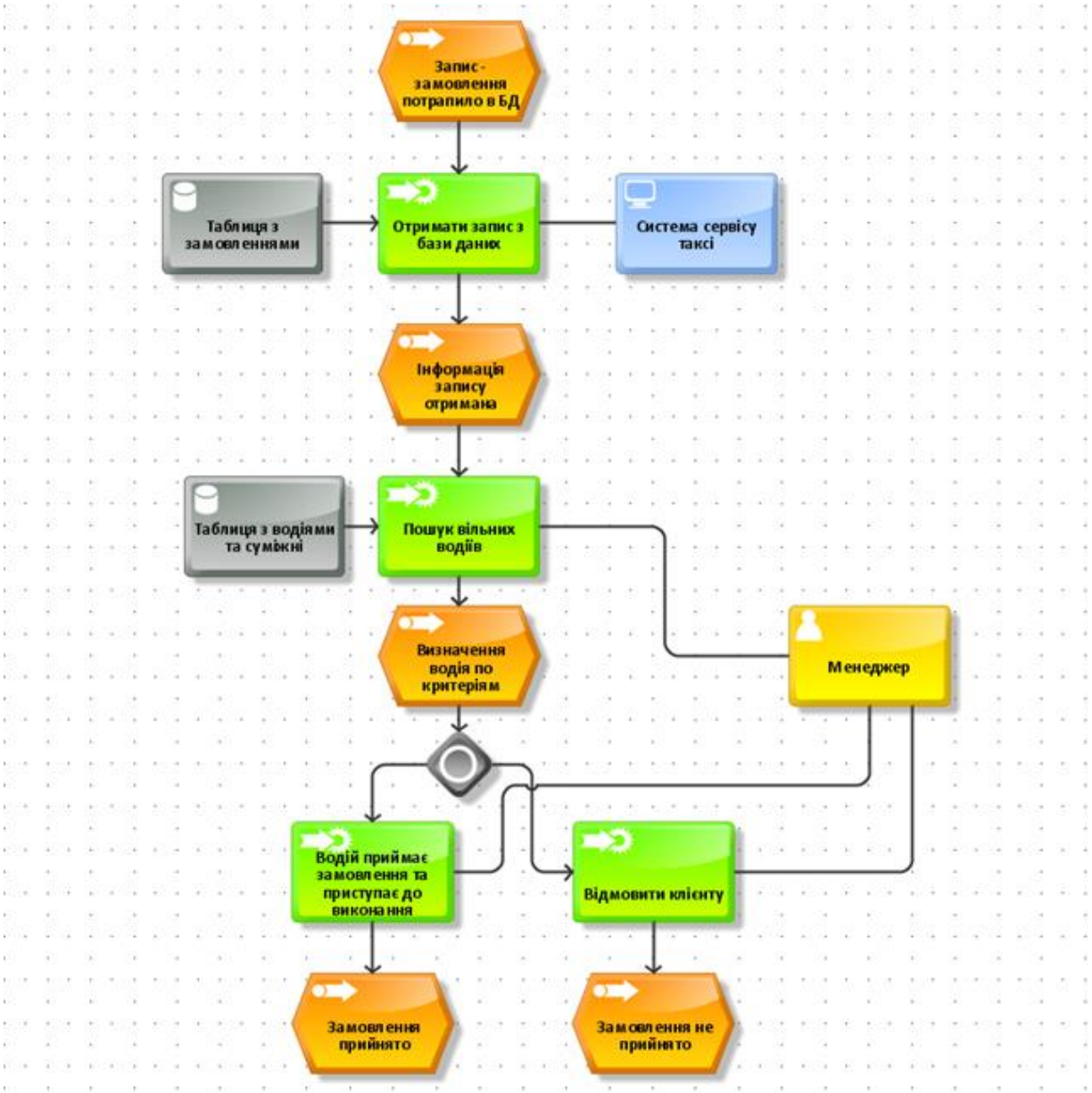


Рисунок 1.1 - eEPC-діаграма приймання замовлення

Сам процес виконання замовлення можна поділити на етапи:

- водій прийняв замовлення;
- водій прибув по адресу замовлення;

– водій почав поїздку з клієнтом;

– водій завершив поїздку;

– в окремому випадку замовлення може бути скасовано на етапі a та b.

Дивлячись на загальну структуру роботи сервісу таксі можна зрозуміти, що агрегатор має бути підключено в момент того, коли для замовлення почався пошук водія. Тобто одним з наступних критеріїв пошуку може бути не «Знайти наступного водія в даному сервісі таксі» а «Надіслати замовлення до агрегатору». З цього випливає те, що агрегатор має бути REST API сервісом з можливістю приймати замовлення на обробку від сервісу таксі. Для опису алгоритму процесу обробки замовлення було побудовано DFD діагарму першого рівня, що наочно зображено на рисунку 1.2.

Агрегатор таксі відразу позиціонується як відкрите API. До нього будуть мати доступ всі бажаючі хто пройде реєстрацію. Тобто для того щоб була можливість відправити замовлення на агрегатор потрбіно мати власний ідентифікатор (токен), який отримує сервіс таксі під час реєстраці. Всі основні дані після проходження валідації потрапляють до бази даних та проходять розподілення між сервісами таксі, що є зареєстровані. Кожен користувач може налаштувати розподілення так як він бажає. Під цим розуміється те, що компанія може вказати кому хоче передавати замовлення та від кого хоче їх отрмувати.\

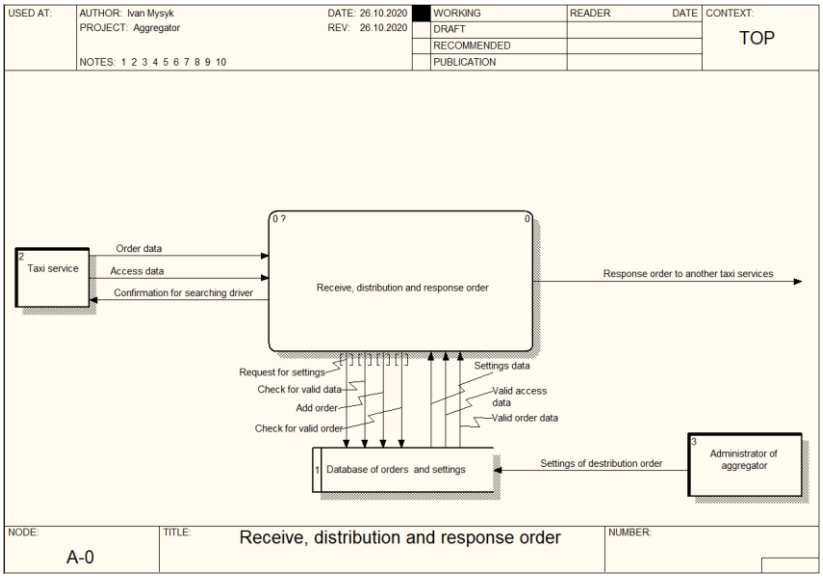


Рисунок 1.2 – DFD діаграма обробки замовлення

Процес валідації та розподілення зображено на DFD діаграмі другого рівня декомпозиції на риснуку 1.3. Після проходження валідації, збереження до бази даних та розподілення починається очікування відповіді від сервісів, які отримали замовлення. Якщо є сервіс, який почав пошук водія для даного замовлення, агрегатор повідомить провайдера замовлення про це. В іншому випадку агрегатор повідомить про невдачу пошуку.

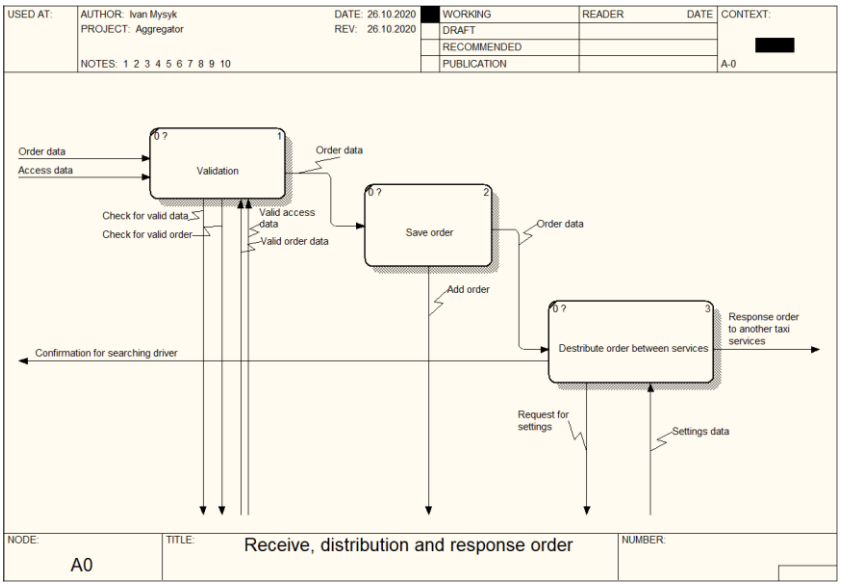


Рисунок 1.3 – DFD діаграма другого рівня декомпозиції обробки замовлення

У випадку якщо водій іншої служби взяв замовлення агрегатор отримає повну інформацію про водія та передасть сервісу клієнта. Далі, про кожен етап виконання замовлення, сервіс таксі має сповіщати агрегатор до моменту доки водій не завершить поїздку або замовлення не буде скасоване. На кожному етапі відбувається оновлення та збереження даних про замовлення. Міняється статус, який слугує ідентифікатором завершення виконання. Про всі зміни буде сповіщено провейдера замовлення, за допомогою чого клієнт буде знати де знаходиться водій та коли він прибуде. При закритті замовлення агрегатор видаляє його та записує в каталог.

У результаті побудовано діаграми потоків даних та структури роботи прогрмного збезпечення таксі. Вони показують структуру за допомогою, якої визначені всі функціональні та нефункціональні вимоги.

1.2 Аналіз наявного програмно-технічного забезпечення предметної області

На даний момент ринок програмних систем такого типу має досить не великий вибір. Відкриті агрегатор, українського виробництва, знайти не вдалось. Із зарубіжних аналогів можна виділити такі програмні застосунки:

- Up&Up;
- SeDi;
- TMMMARKET;
- РБТ.

Кожен з запропонованих сервісів представляє з себе певну Біржу на якій можуть обмінюватись замовленнями як диспетчери таксі так і водії. Усі вони мають свої переваги так і недоліки.

Біржа замовлень Up&Up зарекомендувала себе якісною роботою обмінника, справедливим вирішенням спірних питання та швидкою роботою.

Обмінник можливо підключити в будь-якому місті Росії але мінімальні вимоги – два робочих сервіси таксі, які хочуть обмінюватись замовленнями. Також є можливим запустити обмінник в іншій країні але обов’язковими умовами є десять таксі сервісів які готові обмінюватись замовленнями. Для роботи біржі потрібно провести інтеграцію з диспетчерським робочим місцем.

Відправити замовлення на біржу можна двома шляхами. Перший - це автоматична відправка, за допомогою правил розподілу замовлення. Другий спосіб - ручна відправка на біржу. Як тільки замовлення буде куплене, програма сповістить про це оператора. Також можна переглядати детальну інформацію про водія, який купив замовлення. Відсоток продажу замовлення встановлюється в особистому кабінеті обмінника. Купити замовлення можуть тільки водії в додатку LigaTaxi. В особистому кабінеті Up&Up можна налаштувати максимальний відсоток заробітку з замовлення водієм. Після купівлі, замовлення відразу ж створюється на робочому місці у оператора[2].

Автоматична біжа замовлень таксі SeDi пропонує програмний комплекс, який включає в себе водійський додаток та диспетчерську робочу частину для

керуванням обміном замовлень. Замовлення можна продавати, як в автоматичному режимі так і в ручному режимі.

SeDi пропонує можливість виставляти свої замовлення на аукціон, щоб більше заробити або навпаки - швидше продати замовлення. Також можна брати участь в аукціонах з іншими партнерами за право виконати замовлення. Повністю автоматизовані фінансові розрахунки з партнерами за продані і куплені на біржі SeDi замовлення в режимі реального часу.

Біржа створена в тому числі і для партнерів, які вже працюють на будь-якому ПО і/або поки не бажають переходити на роботу в системі SeDi. У компаній, що використовують програму SeDiManager, біржовий механізм використовується більш ефективно, і всі операції максимально автоматизовані [3].

TMMarket або ЦОЗ (Центр обміну замовленнями) позиціонує себе як атоматизован система для продажу та купівлі замовлень в службах таксі. Пропонуєтсья можливість шукати машину на замовлення відразу в усіх зареєстрованих сервісах таксі. Даний сервіс обмежений в роботі з кокретними системами по обміну замовленням та не має відкритого API [4].

РБТ таксі розшифорвується як «Російська біржа таксі». Біржа працює без без посередників, її не контролює диспетчер, що дає високу швидкість в обробці замовлень. Даний агрегатор працює на пряму з водіями і це дає велику перевагу. Адже без посередників дохід водія збільшується [5].

Більшість існуючих бірж направлені на те, щоб заробити швидше та більше, що погано впливає на швидкість обробки замовлень. Також через те, що стати таксистом стає все простіше завдяки таким обмінникам, погіршується статистика якості обслуговування в цілому.

У нинішніх умовах агрегаторам таксі в Україні навіть немає необхідності мати якесь представництво або офіс. Щоб оформитися водієві потрібно смартфон, автомобіль і водійське посвідчення, а сам процес підключення відбувається найчастіше віддалено. Це дало можливість підключитися і возити пасажирів практично будь-якому водієві. Можна зустріти оголошення про набір водіїв таксі без досвіду, а вік починається від 19-20 років. Це фактично діти за кермом

автомобіля, без повноцінного досвіду, знання міста, сфери та професії, яким українці готові довірити своє здоров'я і життя [6].

Розуміючи нинішню ситуацію відкритих аграгаторів таксі можемо вивести такий перечень характеристик майбутньої програмної системи:

− відкритий для всіх існуючих сторонніх служб таксі,

− абсолютна автоматизація всіх етапів виконання замовлення;

− замовлення, в першу чергу, не продається а розподіляється між всіма користувачами системи;

− захист доступу до системи від сторонніх служб;

В результаті такий перелік характеристик надає бачення системи, як надійної та безпеченої до інформації користувача. Враховуючи те, що користувачем виступає сервіс таксі.

1.3 Визначення вимог до програмного забезпечення та розробка технічного завдання

Для визначення та відображення вимог до програмного забезпечення можна використати уніфіковану мову об’єктно-орієнтованого моделювання – UML. Таку мову може бути застосовано на всіх етапах життєвого циклу аналізу бізнес-систем і розробки додатків. Різні види діаграм які підтримуються UML, і найбагатший набір можливостей представлення певних аспектів системи робить UML універсальним засобом опису як програмних, так і ділових систем.

Для опису вимог буде використана діаграма варіантів використання. Така діаграма, шляхом використання сутностей та акторів, зображує взаємодію програмного комплексу та те, що саме буде виконувати програма. Кожен актор може бути або іншою програмою або людиною, що вмію відтворювати дію над модельованою системою [7].

Актори, що взаємодіють з програмною системою, сервіс таксі та адміністратор, описано в таблиці 1.1. Можливі варіанти використання з описом

додані до таблиці 1.2. У складів акторів є як людина так і система, що може звертатись до нашого програмного комплексу по відповідному інтерфейсу. Різниця в інтерфейсі адміністратора полягає в тому, що адміністратором виступає, як людина і йому потрібний графічний інтерфейс, що надає зручне керування. У випадку стороннього сервісу таксі потрібен інтерфейс на програмному рівні, що представляє собою певний набір функцій. Діаграми варіантів використання зображено на рисунку 1.4 та на рисунку 1.5.

Таблиця 1.1 – Опис акторів

Актор	Опис
Сервіс таксі	Має доступ до надсилання замовлення іншим сервісам таксі. Може змінювати статус замовлення відповідно до його виконання водієм. До прикладу може виконувати прийняття замовлення. Проводити процес виконання та відміняти його.
Адміністратор	Має доступ до панелі керування налаштуваннями розподілення замовлення між сервісами таксі. Може додавати до системи нових користувачів сервісу.

Таблиця 1.2 – Опис варіантів використання

Актор	ВВ	Опис ВВ
1	2	3
Таксі сервіс	Передати нове замовлення	Користувач надсилає власне замовлення до сервісу розподілення між сервісами таксі.
	Взяти замовлення	Після розподілення замовлень між сервісами, користувач може взяти замовлення, якщо водій прийняв його на виконання.
	Вказати що водій по адресу	Під час виконання замовлення водій має повідомити про те, що приїхав до клієнта.
	Вказати що водій почав поїздку	Під час виконання замовлення водій має повідомити про те, що почав поїздку з клієнтом.
	Вказати, що водій завершив поїздку	Під час виконання замовлення водій має повідомити про те, що водій успішно доставив клієнта до місця призначення.
	Відмінити замовлення	Клієнт або водій може відмовитись від виконання замовлення.
Адміністратор	Налаштування розподілу замовлення	Адміністратор вказує які таксі сервіси будуть отримувати замовлення від інших таксі сервісів. Також вказує час затримки розсилання для кожного сервісу таксі.
	Реєстрація нового таксі сервісу	Відбувається додавання особистих даних сервісу таксі.

Рис. 1.4 – Діаграма варіантів використання для адміністратора

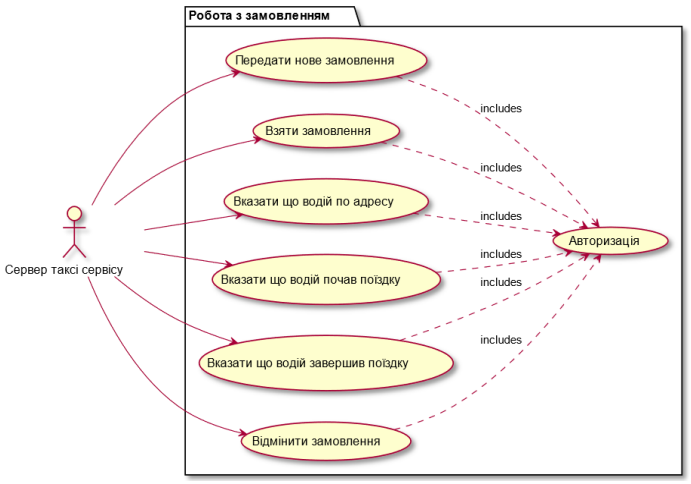


Рис. 1.5 – Діаграма варіантів використання для таксі сервісу

Згідно з проведеним аналізом предметної області було встановлено, що відкриті сервіси автоматичного розподілення замовлень таксі є досить тонко направленою сферою. Структура таких застосунків залежить від існуючих сервісів таксі та інших факторів.

Аналіз існуючого програмного забезпечення показав, що є багато різного роду програмних комплексів за кордоном та ні одного відкритого в Україні. Або такі комплекси мають обмежений доступ для окремих лиць. Усі такі програмні комплекси маю свої позитивні сторони але і більш явні мінуси, а саме застарілість технологій та підходу до вирішення проблеми. Кожен такий сервіс хоче, в першу чергу, заробити а потім виконати поставлену мету.

Загалом визначено усі функціональні властивості програмного комплексу [8]. Описано вимоги, що вказують на вирішення поставлених задач та сформульовано технічне завдання, що знаходиться в додатку А.

2 ПРОЕКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Вибір типу архітектури та шаблонів проектування

При виборі типу архітектури потрібно звернути увагу на те які бувають типи та провести аналіз проектування серверної архітектури. В основі сучасного веб-сервісу лежить клієнт-серверна архітектура. Вона складається з клієнту та серверу. Коли говорять «клієнт», під цим мається на увазі, що це пристрій, який володій вмінням надсилати дані по протоколу, що вказує сервер. Сам сервер – це інший пристрій, що має високу потужність та вміє обробляти запити клієнтів. Загалом його задача полягає в обробці даних клієнта та надсиланні коректної відповіді по певному протоколу.

Під час роботи серверу може існувати декілька підключених клієнтів одночасно. Кожен клієнт надсилає запити та очікує відповіді. Звідси можна зрозуміти, що сервер повинен підключати клієнтів асинхронно задля швидшого відгуку до клієнтів.

Як вже говорилось, сервер вказує по якому протоколу будуть передаватись дані. Таких протоколів може бути безліч але основним рахується HTTP – протокол передачі гіпертексту. На разі API серверних додатків швидко розвивається і дані передаються в любих форматах. Такий підхід залишає усі принципи роботи та переваги HTTP але дозволяє більш зручно передавати дані великого об’єму. Звідси слідує, що HTTP це протокол передачі не лише гіпертексту але і любих інших даних.

Для створення прикладного програмного інтерфейсу з’явилось безліч архітектурних підходів, що полегшують проектування API. Самий популярний – REST (Representational State Transfer) – передача стану представлення. В нашому випадку, такий архітектурний підхід, максимально підходить через використовуваний протокол. В HTTP, при передачі даних, потрібно вказувати тип запиту, що повідомляє про дію, яку потрібно зробити. А в той же час REST архітектура передбачає метод в інтерфейсі, що обробляє конкретний запит [9].

Ми визначились з архітектурою програмної системи після чого можна

визначити шаблон проектування. Архітектура серверного програмного забезпечення може бути поділена на мікросервіси та моноліт. Монолітна архітектура має під собою принцип єдиного розвертання. Всі сервіси включені в дин модуль, що включає в себе всі шари від представлення до даних. Мікросервісна архітектура – це слабо зв’язані модулі, що ділять по логічному використанню кожного. У порівнянні з монолітним сервісом у мікросервісів декілька одиниць розвертання.

Монолітна архітектура має такі переваги:

− легше та швидше реалізація за рахунок того, що є лише один модуль і не потрібно думати як реалізувати взаємодію між сервісами;

− за рахунок єдиного інтерфейс end-to-end тести проводити легше;

− легке масштабування.

Недоліки монолітної архітектури:

− швидко нарощують надлишкові сервіси та під модулі, що ускладнюють підтримку загальної архітектури;

− потрібно на старті правильно обрати основу, тому що її потрібно буде притримуватись на протязі всього проекту;

− відсутня ізоляція. помилка в одному модульному тесті може поламати весь програмний застосунок.

Мікросервісна архітектура має такі переваги:

− кожен мікросервіс не великого розімру через, що його легко розуміти і підтримувати протягом довгого періоду;

− швидший запуск або перезапуск;

− не прив’язані до конкретної технології або мови програмування;

− правильне проектування може пережити збій одного сервісу.

Недоліки мікросервісної архітектури:

− складніше тестування;

− транзакції виконувати складно і потрібні додаткові рішення;

− зміни в декількох мікросервісах складно координувати.

Дивлячись на завдання дипломного проекту та аналіз архітектури було

обрано клієнт-серверну архітектуру з поправкою на те, що клієнтом системи виступатимуть інші сервери. Звідси слідує, що в даного програмного забезпечення немає клієнтського сервісу чи додатку. Шаблоном серверної архітектури буде виступати моноліт. Таке рішення було виведено через те що, сервіс, який буде результатом завдання не є великого розміру. Він легко масштабується та не потребує складних тестів.

2.2 Аналіз та вибір бази даних

Любий enterprise сервіс працює з великими об’ємами інформації. Завжди потрібно інформацію зберігати, модифікувати та видаляти. Такі операції завжди присутні у великій кількості, що займає найбільше часу. Через що, вибір сховища даних, по важливості, на рівні з проектування всього програмного комплексу.

Усі сховища даних а точніше – бази даних можна поділити на категорії:

− реляційна база даних. Модель такого типу пропонує табличну структуру та зберігання даних. Такі бази даних гарантують надійний захист інформації від втрат або пошкодження. Така ефективність була досягнута завдяки суворим принципам та правилам.

− безмодельний підхід або іншими словами NoSQL. Такий підхід позбавляє від всіх обмежень збереження та структуризації даних. Використовуючи дану модель бази даних зручно зберігати та модифікувати текстові документи.

Порівнюючи два типи баз даних можна зрозуміти, що реляційні бази даних потребують суворо визначену структуру на відміну від NoSQL баз даних. Реляційна модель підтримує стандартизовану мову запитів SQL, коли NoSQL база даних реалізовує свій механізм запитів. Надійність SQL баз даних вища. Складні структури даних на багато ефективніше реалізовувати на РСУБД. Тому було прийнято рішення використовувати реляційну базу даних [10].

Розглядаючи рішення реляційних баз даних можна побачити такі популярні варіанти, як MySQL та PostgresSQL. Кожна з цих реалізацій реляційної бази даних

мають переваги та недоліки.

Для вибору бази даних є декілька керуючих властивостей: документація, потужність, складність. Кажучи про документацію Postgres є найкращим прикладом. Компанія описала не просто набір пояснень до кожної можливої функції а повноцінний посібник з прикладами та рішеннями можливих бізнес проблем. В той час MySQL завжди пропонує шукати певні пояснення на сторонніх ресурсах таких як Percona.

Складність MySQL та PostgresSQL абсолютно різна і це випливає на їхню ефективність та потужність. Простота налаштування MySQL показує те, що її алгоритми оптимізації не покривають всіх потреб розробників, що ускладнює всю розробку. PostgresSQL має потужну основу, яка вміє модифікувати складні запити для оптимальної швидкості виконання. Набір інструментів Postgres дозволяє аналізувати навантаження на базу даних і відносно зібраних даних проводити гнучке налаштування [11].

Таким чином ми робимо вибір в сторону PostgresSQL. Для рішення наявної задачі дипломного проекту таке рішення є оптимальним. Адже, проект є не великого розміру а інструмент, що надається базою даних покриває потреби.

2.3 Опис декомпозиції

Серверне програмне забезпечення побудоване на трирівневій архітектурі. Така архітектура дозволяє захиститись від SQL ін’єкцій, розгалужити доступ до даних та модифікувати їх перед відправкою. Кожен рівень прийнято називати шарами де кожен відповідає за власну функцію:

− шар представлення. В нашому випадку це шар доступу до API з потрібним набором функціоналу. Також це може бути клієнтський додаток, що написаний на іншій мові програмування. Веб сторінка, що відкривається у браузері. Таку сторінку може генерувати сервер або просто передавати по відповідному запиту;

− шар бізнес логіки обробляє дані з шару представлення та доступу до даних. Цей рівень абстракції відповідає за виконання CRUD операцій над даними та передачею їх у відповідний контролер презентаційного шару;

− шар доступу до даних. На цьому рівні відбувається опис функції доступу до даних. Будується весь модельний ряд та зв’язки між ними. Такий шар називають репозиторієм. Інтерфейс репозиторію може реалізовуватись для будь якої бази даних.

Шар представлення містить в собі функціонал з конкретного набору функцій для сервісів таксі. Кожна функція потребує певні дані для правильної роботи. Шар представлення повинен правильно обробити всі дані, перевірити їх на валідність, та унікальність. Умова унікальності повинна бути дотримана в функції створення замовлення. Не може існувати більше ніж одне конкретне замовлення на сервері.

Сервіси бізнес логіки мають відповідний набір функцій для збереження даних та їх обробки для подальшого виконання замовлення. Функція збереження замовлення також стартує пошук сервісів таксі для розсилки. Після чого починається виконання замовлення. На кожному етапі виконання зберігається стан замовлення та сповіщається сервер провайдера замовлення. По завершенню виконання замовлення потрапляє до звітів та видаляється.

Структура проекту являє собою набір пакетів, що містять в собі набори класів та ключових компонент. Кожен пакет відноситься до одного з трьох шарів архітектури проекту або до кожного з них.

Пакет configuration містить в собі налаштування контексту програми. Описані дії що до використання ресурсів. Проведена конфігурація бази даних і визначення реалізації доступу до даних також описані налаштування мережі і типів даних, що може оброблять сервер.

Controller містить в собі класи контролери, які виступають API інтерфейсом для користувача. Вони виконують функцію валідації даних та авторизації користувачів, що виконують запити по API.

Dto – пакет, що містить кали для обробки моделей вхідних даних та

складається з такого набору під пакетів: Fare, Interfaces, Order, Request, Response. Пакет Fare містить в собі структуру класів тарифу по якому виконується замовлення. Пакет Interfaces складає в собі опис інтерфейсу типового DTO для замовлення. В наступних пакетах знаходяться класи для запитів кожного етапу виконання замовлення та модель відповідей.

Пакет Entity містить в собі класи, що відображають сутності предметної області. В таких сутностях описані всі зв’язки та поля, що відповідають таблицям в базі даних.

Model містить в собі моделі спеціальних структур даних, що зображені в таблиці, як JSON поля. Такі поля використовуються, для кращої міграції між різними сервісами таксі, тому що можуть існувати різні формати адрес та тарифів.

Пакет Repository містить інтерфейси репозиторії, для доступу до даних. Вони реалізовують функціонал SQL запитів до таблиць та приводять табличні дані до об’єктів.

Пакет Service містить клас бізнес логіки, пакети з логікою валідації та реалізацією стороннього HTTP клієнта. Тут відбувається вся логіка виконання замовлення та інші додаткові дії.

В результаті були описана структура пакетів проекту. Вказані характеристики кожного пакету, що відображає логіку побудови системи розподілення та системи контролю запитів.

2.4 Опис залежностей та інтерфейсів

Структуру проекту, як вже говорилось раніше, розбито на головних три шари, які залежать один від одного. Кожен пакет є частиною конкретного шару або до декількох одночасно.

Шар представлення містить в собі клас Controller, що залежить від класу OrderService. Ця залежність обумовлення тим, що клас представлення має визначити яку дію хоче виконати користувач, обробити вхідні дані та направити

на виконання конкретної логіки.

Валідація даних проходить за допомогою класу DataBaseValidator. Назва класу визначається тим, що валідація проходить шляхом порівняння вхідних даних з даними бази даних. Під валідацією мається на увазі, що йде перевірка на ключ доступу до серверу та уніфікація замовлення. Анотація Authorize, що реалізується класом ServiceAuthorize, проводить перевірку на ключ доступу. Анотація UniqueOrder, що реалізується класом OrderUnique, проводить перевірку на наявність такого ж замовлення в базі даних. Діаграма залежностей даного модуля зображено на рисунку 2.1.

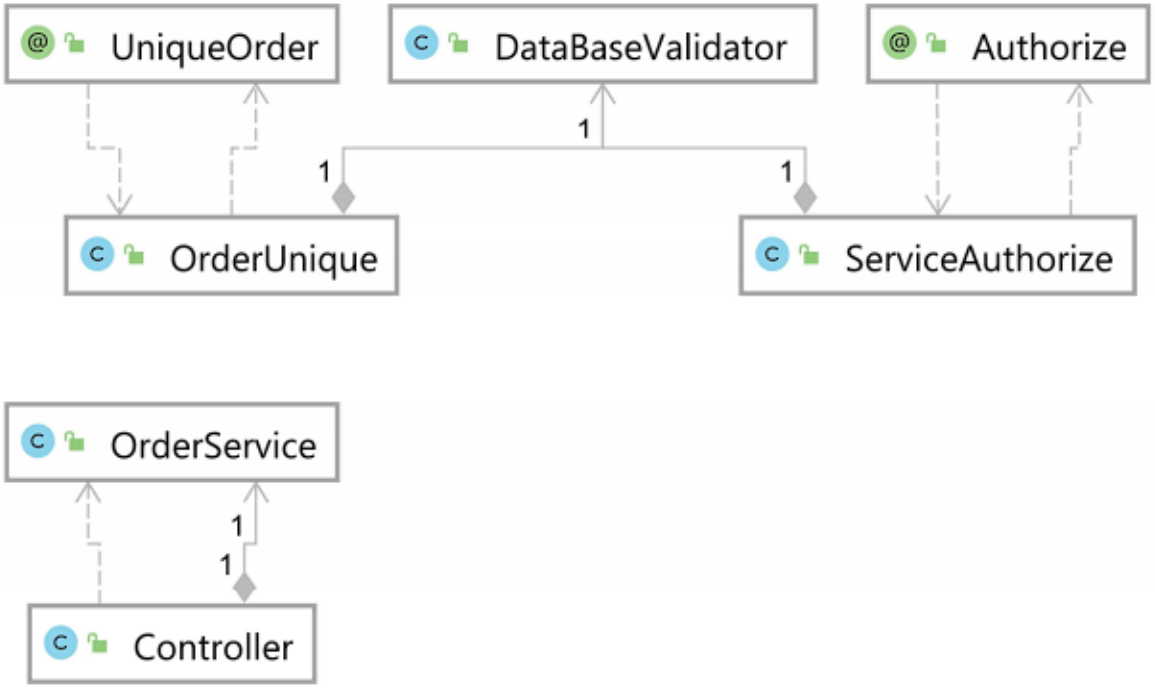


Рисунок 2.1 – Діаграма залежностей шару представлення

Функціональний набір класу Controller зображено на рисунку 2.2. Кожен метод відповідає етапам виконання замовлення. Для всіх методів відповідно є клас, що містить в собі інформацію для збереження та надсилання до інших сервісів. Для передачі даних використовується так зване DTO. Такий патерн представлення вхідних даних дозволяє розмежувати сутності предметної області від вхідних даних користувача, адже кожен сервіс таксі може мати відмінні структури даних які потрібно зберегти в базі.

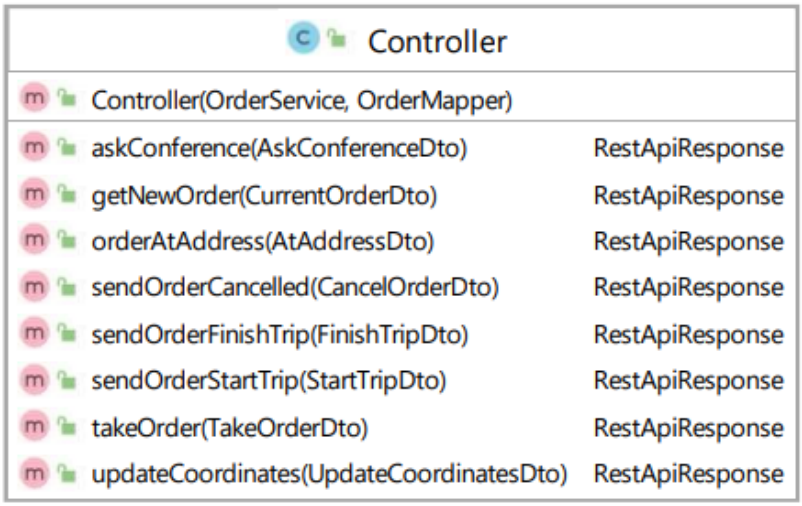


Рисунок 2.2 – Функціональний набір класу Controller

Функціонал класу DataBaseValidator зображено на рисунку 2.3. Він складається з двох методів, що перевіряють сервіс на наявність валідного ключа доступу та унікальність замовлення в базі.

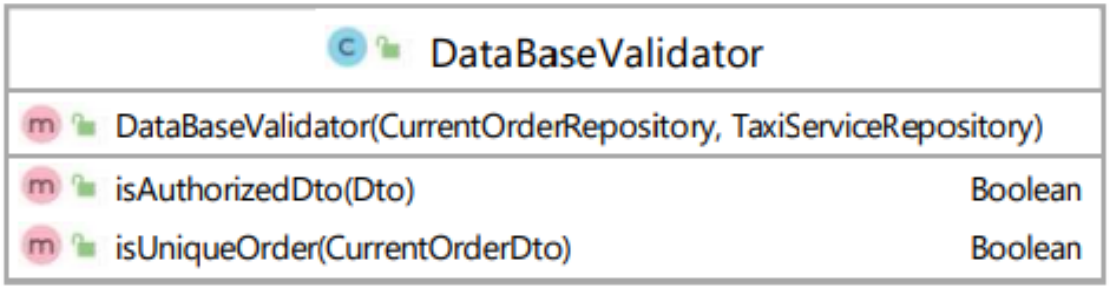


Рисунок 2.3 – Функціональний набір класу DataBaseValidator

Класи OrderUnique та ServiceAuthorize реалізовують перевірку валідності надходження інформації шляхом вибірки даних з бази даних та порівнянням їх. Функціональний набір класів зображено на рисунках 2.4 та 2.5.

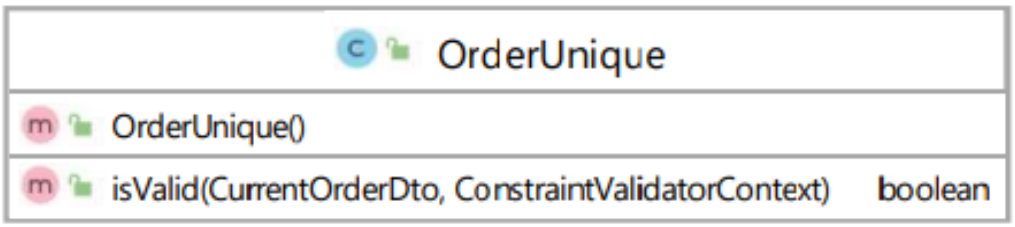


Рисунок 2.4 – Функціональний набір класу OrderUnique

Рисунок 2.5 – Функціональний набір класу ServiceAuthorize

Для реалізації структури вхідних даних потрібно дев’ять класів, що реалізовують інтерфейс Dto. Кожен клас, що зображено на рисунку 2.6, містить інформацію про конкретний етап виконання замовлення та обов’язкові дані авторизації, що реалізуються з Dto. Структура даних цього типу дозволяє максимально ефективно міняти та вдосконалювати систему.

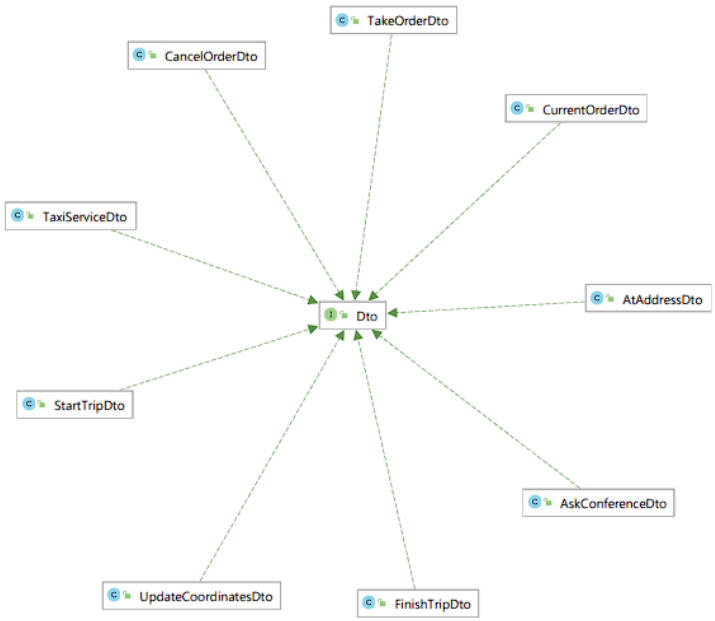


Рисунок 2.6 – Діаграма залежностей DTO класів

Клас поточного замовлення CurrentOrderDto містить в собі клас тарифу. Діаграму залежностей зображено на рисунку 2.7. Особливістю даної структури тарифу являється наявність не обмеженої кількості варіацій надання комісій та коефіцієнтів за різні послуги. За рахунок цих особливостей набір послуг, що включає в себе тариф є максимально гнучким. Кожен сервіс таксі, який отримує

замовлення з встановленим тарифом може зберегти його в єдиному форматі, що дозволяє працювати з різними програмними забезпеченнями.

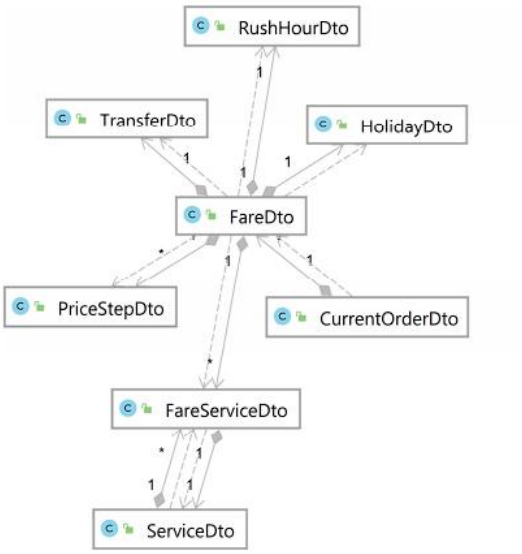


Рисунок 2.7 – Діаграма залежностей тарифу

Шар бізнес логіки включає в себе клас OrderService, що залежить від класів репозиторіїв, класів трансформаторів та HTTP клієнта. Діаграма залежностей шару бізнес логіки зображена на рисунку 2.8.

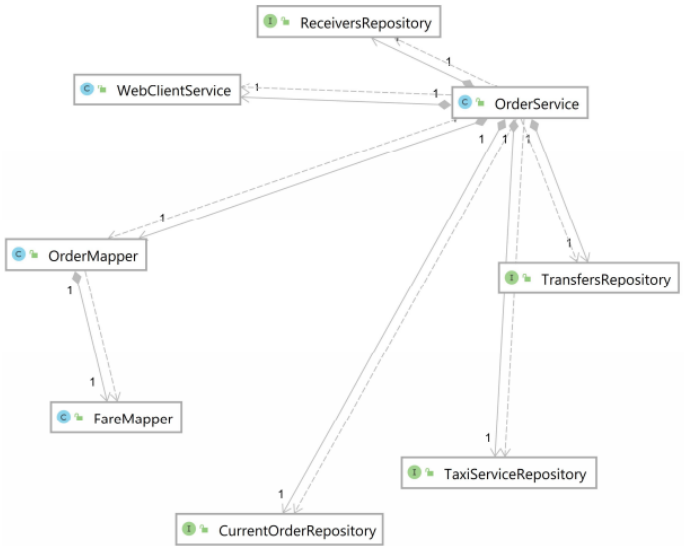


Рисунок 2.9 – Діаграма залежностей шару бізнес логіки

Клас OrderMapper виконує функціонал перетворення DTO в Entity та

навпаки. Через це є залежність від класу FareMapper, адже тариф є залежністю замовлення, що провокує собою автоматичну трансформацію. Функціональний набір класу OrderMapper зображено на рисунку 2.10.

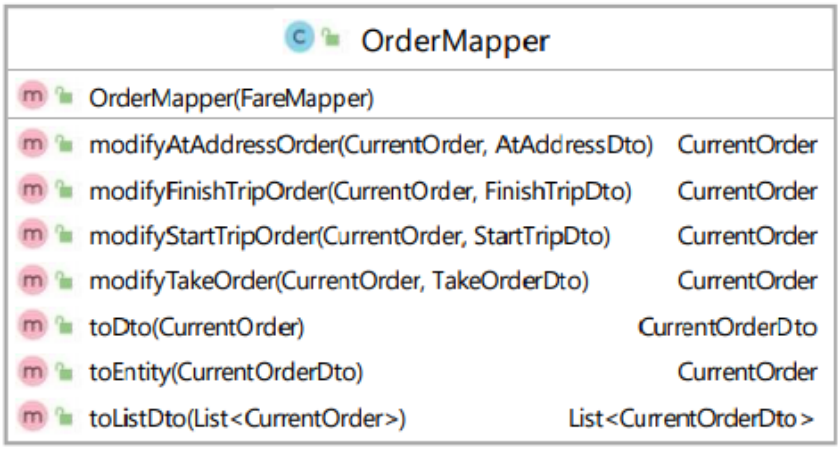


Рисунок 2.10 – Функціональний набір класу OrderMapper

Весь функціонал бізнес логіки використовує доступ до бази даних. Надається він за допомогою інтерфейсів репозиторіїв. Їхня реалізація лягає на сторонній фреймворк, що сильно спрощує завдання. Функціональний набір CurrentOrderRepository з основними модифікованими функціями доступу до даних зображено на рисунку 2.11.

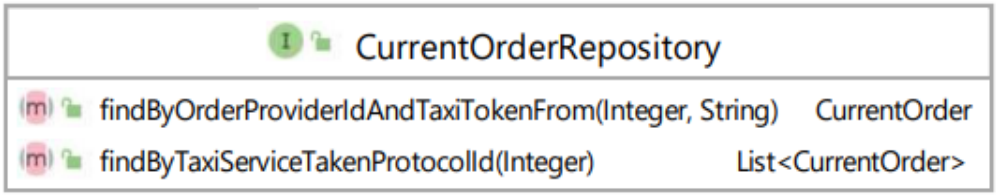


Рисунок 2.11 – Функціональний набір класу CurrentOrderrepository

Головною задачею в налаштуванні доступу до бази даних це опис сутностей. Діаграма залежностей сутностей зображена на рисунку 2.12. Класи Transfers та Receivers обумовленні налаштуваннями агрегатора та мають залежність один до одного. Вони втілюють налаштування переадресації замовлення. Сервіси, що хочуть передавати іншому сервісу відображаються в

Transfers. А сервіс таксі, що хоче приймати від іншого таксі - Receivers.

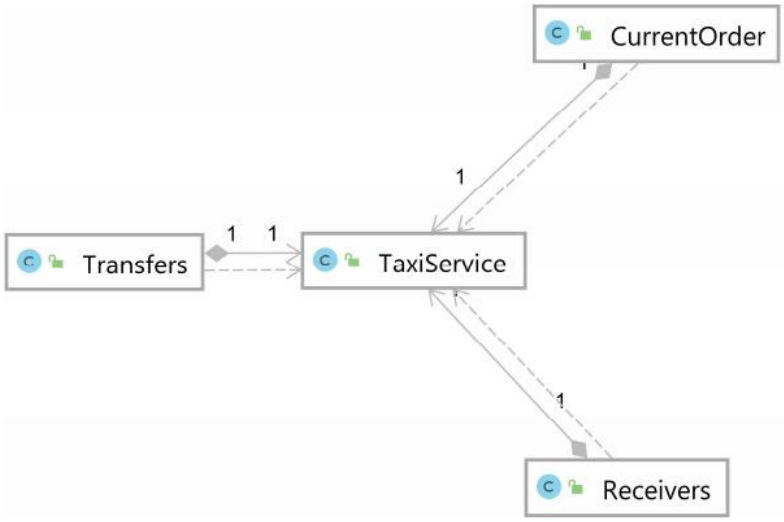


Рисунок 2.12 – Діаграма залежностей сутностей

Головний клас бізнес логіки OrderService, використовуючи, попередньо описані, залежності, має наступний функціональний набір, що зображено на рисунку 2.13.

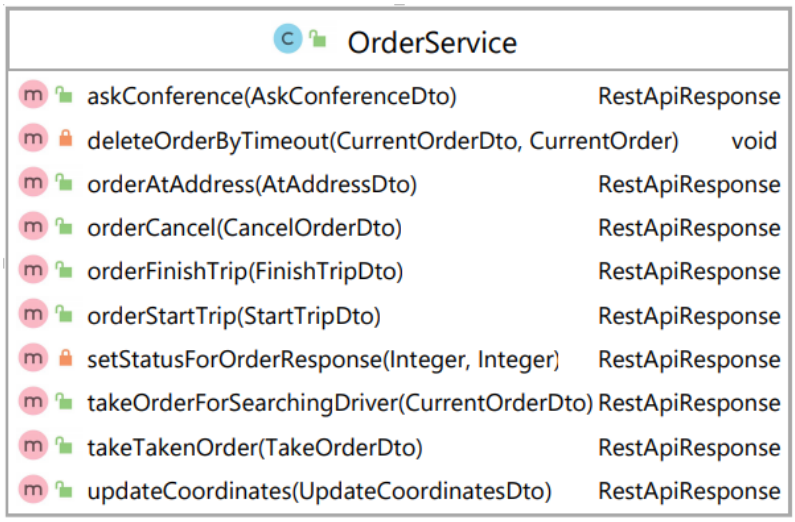


Рисунок 2.13 – Функціональний набір класу orderService

В висновку розділу було сформовані інтерфейси та залежності в системі. Визначені функціональні властивості сервісів. Побудовані діаграми зв’язків підсистем, що формують проект.

2.5 Аналіз та вибір технологій і методів реалізації додатку

Для розробки серверного забезпечення було проаналізовано декілька популярних технологій, що надають інструменти для рішення таких задач:

- web API;
- об’єктно орієнтоване представлення таблиць баз даних;
- багатопотоковість;
- автоматичне створення контексту;
- кросплатформність.

Під такі характеристики підпадають такі технології написання програмного забезпечення, як Java та C#. Обидві мови програмування мають свої переваги та недоліки, що впливають на загальний вибір.

C#, як мова програмування не може покрити всі потреби та задачі, що стоять перед дипломним проектом. Тому її використовують лише з .NET Framework, що включає в себе багато бібліотек [12], таких як:

- ASP.NET – це найбільша технологія створення серверного програмного забезпечення від Microsoft.
- WPF – API для розгортання графічних програм для настільних ПК;
- Silverlight технологія для створення потужних додатків, які виконуються в браузерах під управлінням різних операційних систем.
- Entity Framework надає можливість роботи з базами даних через об'єктно-орієнтоване представлення коду.

Як можна зрозуміти такі технології є надлишковими, хоч і частково покривають потреби дипломного проекту. Також C# має певний перелік недоліків, таких як:

- використання JIT-компілятора, що сповільнює роботу на початку;
- система в першу чергу буде працювати на Windows, що сильно обмежує запуск серверу на інших платформах. І в той же час сам Windows може забирати багато ресурсів комп’ютера замість того, щоб сервер використовував їх для продуктивнішої роботи;

Java, як об’єктно-орієнтована мова програмування не може покрити всі задачі дипломного проекту, через що потрібно прийти до використання фреймворків [13], таких як:

− Spring Framework – контейнер для ін’єкції залежностей;

− Hibernate – фреймворк, що підтримується всіма JVM мовами програмування, для представлення таблиць баз даних у вигляді об’єктів.

Зазвичай коли говориться про enterprise проект на java завжди мається на увазі фреймворк Spring. Даний фреймворк пропонує не лише інверсію управління а і додаткові пакети корисних засобів:

− Spring Security;

− Spring JPA;

− Spring WebClient.

Дані пакети надають розширений функціонал для швидкої та ефективної розробки. Бібліотека Spring JPA надає можливість автоматичної генерації SQL запитів та форматування даних в POJO об’єкти. Spring також пропонує набір утиліт для розробки гнучкого WEB API. Великий набір функціоналу для багатопотоковості суттєво зменшує складність розробки логіки розсилки.

Особистими перевагами мови програмування є кросплатформність та велика спільнота, що може допомогти у вирішенні типових та складних задач. Вся компіляція коду та збірка проекту відбувається під час першого запуску, що надає перевагу перед C#. Завдяки кросплатформності сервер можливо запустити на простій ОС Ubuntu, що суттєво полегшить роботу у момент простою та максимального навантаження.

Згідно з проаналізованими методами та технологіями реалізації подібних систем були обрані інструменти реалізації серверного програмного забезпечення. А саме мова програмування java у взаємодії з Spring.

При аналізі баз даних було визначено, що для поставленої мети реляційна база даних підходить найбільше. Даний вибір впав на такий тип СКБД за рахунок зручності доступу, швидкістю фільтрації та організації даних. Аналізуючи наявні реляційні бази даних вибір впав на Postgres за рахунок великої кількості якісної

документації та великого спектру налаштувань.

На етапі опису декомпозиції була визначена трирівнева архітектура системи. Вибір впав на таку архітектуру за рахунок можливостей розділення різних типів функціоналу та захист від SQL ін’єкцій. Також спроектовані пакети з відповідним функціоналом. Під час опису залежностей та інтерфейсів визначені всі функціональні властивості до програмної системи. Побудована ієрархія залежностей між усіма компонентами.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ

3.1 Розробка бази даних

Відповідно проаналізованій предметній області база даних має містити набір таблиць з мінімальною кількістю посилань один на одну для швидшого виконання запитів. Завдяки реляційній моделі бази даних буде реалізована швидка вибірка та фільтрація даних.

Для збереження інформації про замовлення створена таблиця current_orders. Вона має зовнішні ключі на таблиці taxi_services та order_status. Дані зв’язки обумовлені тим, що кожне замовлення має сервіс таксі, що його створив та сервіс, що його виконує. Статуси замовлення не завжди статичні, і можуть мінятись в майбутньому, що буде спрощено завдяки таблиці order_status. Діаграма зв’язків таблиць зображена на рисунку 3.1.

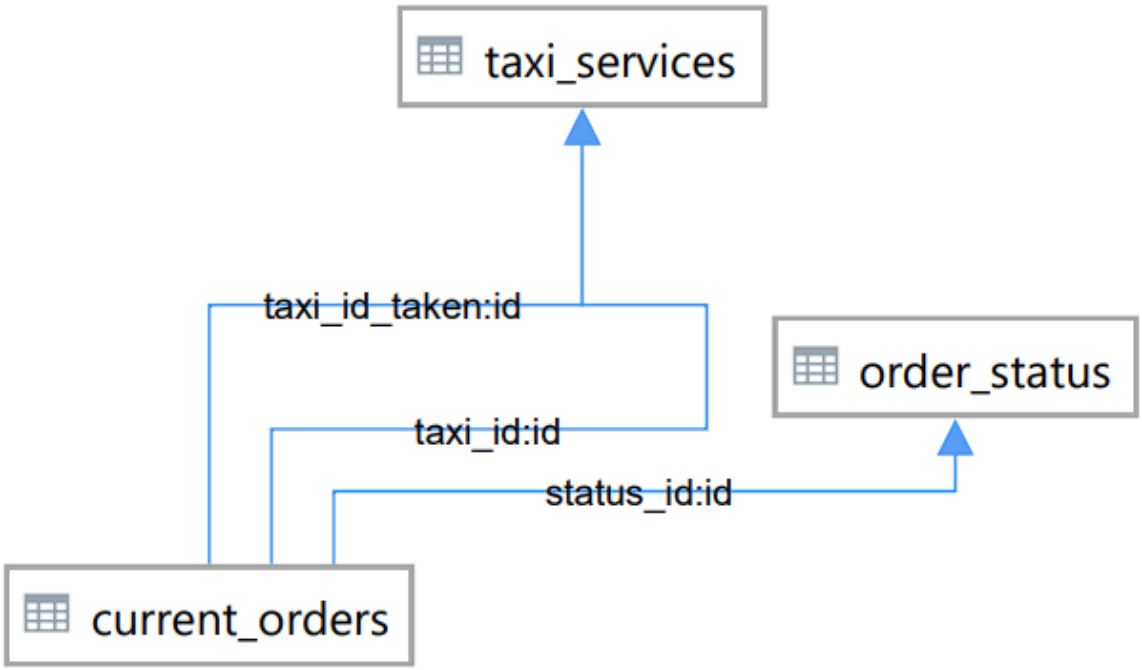


Рисунок 3.1 – Діаграма зв’язків current_orders до інших таблиць

Таблиця для збереження замовлень має велику кількість полів, що не було розбито на під таблиці задля підвищення швидкості виконання запитів та спрощення структури бази даних. Таблиця зображена на рисунку 3.2.

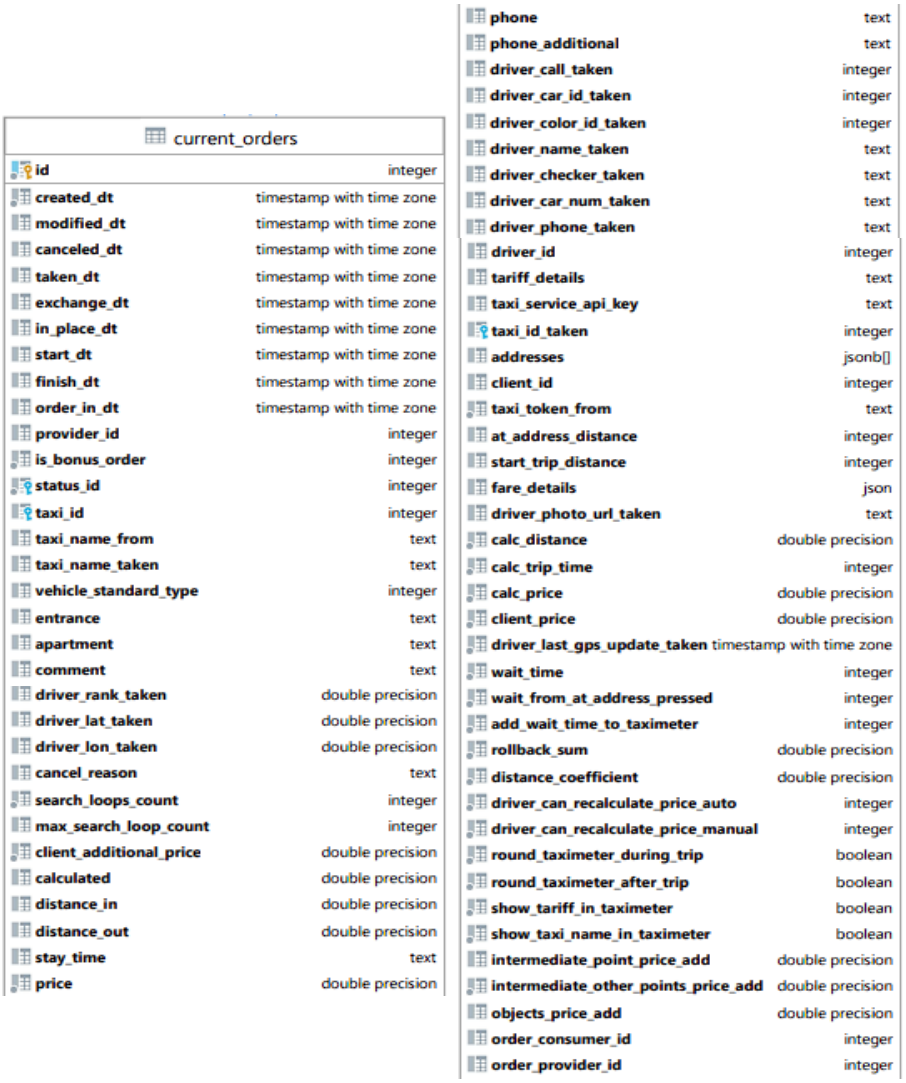


Рисунок 3.2 – Поля таблиці current_orders

Інформація про сервіси таксі буде зберігатись в таблиці taxi_services. Поля таблиці зображено на рисунку 3.3.

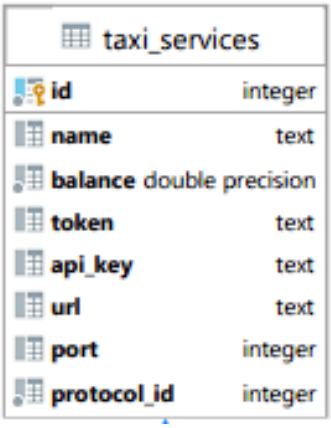


Рисунок 3.3 – Поля таблиці taxi_services

Статуси замовлень будуть налаштовуватись та зберігатись в таблиці order_status. Для зручності можна буде називати статуси на декількох мовах. Поля таблиці зображено на рисунку 3.4.

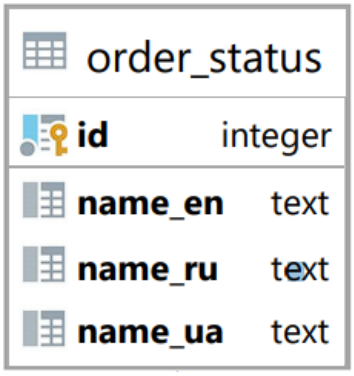


Рисунок 3.4 – Поля таблиці order_status

Налаштування розподілення замовлення між сервісами таксі зберігатиметься в структурі з трьох таблиць, що . Діаграма зв'язків таблиць для налаштувань зображена на рисунку 3.5.

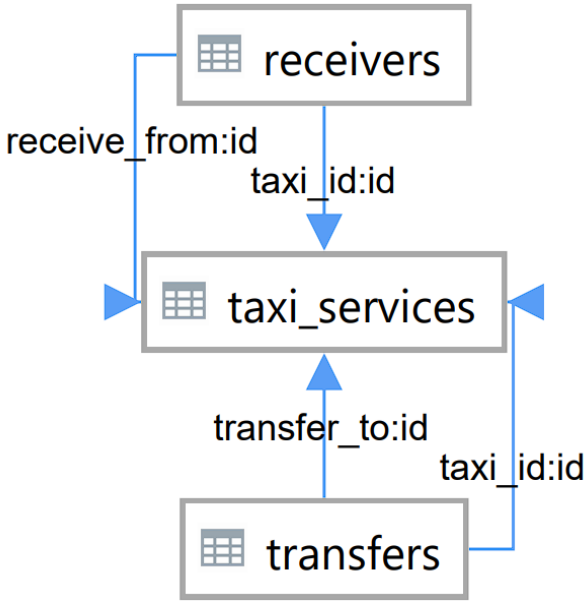


Рисунок 3.5 – Діаграма зв'язків налаштувань

Логіка таблиці receivers полягає у тому, що в ній буде знаходитись сервіс таксі з налаштуваннями того, від яких сервісів таксі він хоче отримувати

замовлення. Таблиця transfers має протилежну логіку. Зберігатись будуть дані про те, які сервіси таксі можуть передавати дані до інших сервісів. Поля таблиці receivers зображено на рисунку 3.6.

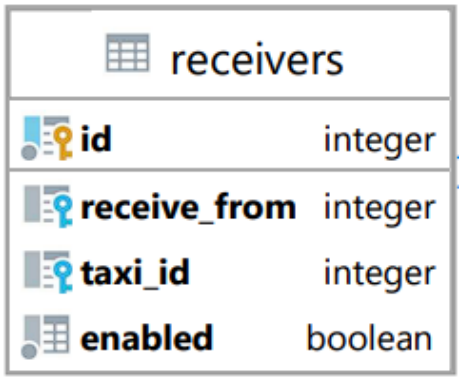


Рисунок 3.6 – Поля таблиці receivers

Поля таблиці transfers зображено на рисунку 3.7. Як можна побачити, комісію та затримку надсилання диктує той хто передав замовлення. Затримка надсилання потрібна для визначення пріоритету сервісу таксі відносно інших.

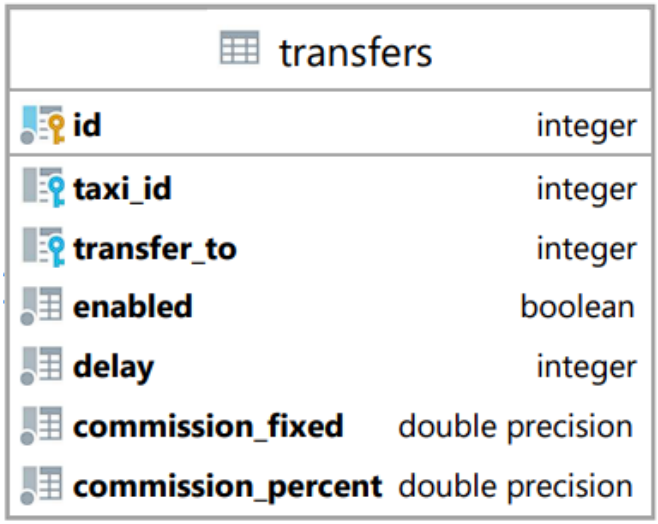


Рисунок 3.7 – Поля таблиці transfers

В результаті побудови бази даних була сформована архітектура реляційних зв’язків, що дозволяють зберігати інформацію пов’язану з предметною областю. Перераховані всі таблиці, що містять поля для збереження потрібної інформації.

3.2 Реалізація модулів системи

Кожен модуль системи має певний набір класів, що реалізує логіку певного шару до якого він відноситься. Для повного розуміння реалізації потрібно описати структури даних та їх валідацію, що використовуються в системі.

Усі дані, що попадають на сервер є DTO класами, що містять в собі структуру вхідних даних. Кожна DTO має анотацію Authorize. Та DTO замовлення має анотацію OrderUnique.

Анотація в Java – це певна форма метаданих, які можна додати до вихідного коду. Анотацією може бути позначені:

- властивість класу;
- метод класу;
- клас;
- параметри методів;
- пакети.

Анотації бувають різних типів та призначення. При створенні власної анотації потрібно вказати її Retention, що допоможе визначити її життєвий цикл. Під життєвим циклом вказується те, де буде анотація активна:

- лише в вихідному коді програми;
- в скомпільованому та вихідному коді програми;
- в процесі виконання.

Кожен параметр допомагає уточнити, як анотація буде використовуватись. Чи вона буде позначати метод який не повертає NULL значення чи під час роботи буде звертатись до певного класу і виконувати певну логіку.

В нашому випадку анотації Authorize та OrderUnique працюють у процесі виконання, тобто Retention є Runtime.

Як говорилось вище, анотація може помічати різні синтаксичні конструкції мови Java. Анотація Target, що має на анотації, допомагає визначити на що анотацію можна прикріпити.

Для визначення сервісу валідації і логіки використовується анотація

Constraint. Як параметр, дана анотація приймає клас, що реалізовує ConstraintValidator. В свою чергу такий інтерфейс пропонує до реалізації методи initialize та isValid. Приклад реалізації сервісу авторизації:

```
package com.fastaxi.Aggregator.service.Validation.Authorization;

public class ServiceAuthorize implements
ConstraintValidator<Authorize, Dto> {

    @Autowired
    private DataBaseValidator dataBaseValidator;

    @Override
    public void initialize(Authorize constraintAnnotation) {

    }

    @Override
    public boolean isValid(Dto dto, ConstraintValidatorContext
constraintValidatorContext) {
        if (dto == null) return false;
        return dataBaseValidator.isAuthorizedDto(dto);
    }
}
```

Метод ініціалізації грає роль для встановлення певних початкових параметрів, якщо такі є. IsValid має сигнатуру в якій передається тіло поміченого об’єкту та контекст валідатора. У такому методі виконується перевірка даних на авторизацію. Алгоритми авторизації та валідації:

```
package
com.fastaxi.Aggregator.service.Validation.ValidationFunction;

@Service
public class DataBaseValidator {

    private final CurrentOrderRepository currentOrderRepository;
    private final TaxiServiceRepository taxiServiceRepository;

    public DataBaseValidator(
CurrentOrderRepository currentOrderRepository,
TaxiServiceRepository taxiServiceRepository)
{
        this.currentOrderRepository = currentOrderRepository;
        this.taxiServiceRepository = taxiServiceRepository;
```

```
    }

    public Boolean isUniqueOrder(CurrentOrderDto orderDto) {
        try {
            CurrentOrder currentOrder =
currentOrderRepository.findByOrderProviderIdAndTaxiTokenFrom(orderDt
o.getId(), orderDto.getTaxiTokenFrom());
            return currentOrder == null;
        } catch (Exception e) {
            return true;
        }
    }

    public Boolean isAuthorizedDto(Dto orderDto) {
        String key;
        try {
            key =
taxiServiceRepository.findByTokenAndApiKey(orderDto.getTaxiToken(),
orderDto.getApiKey())
.getApiKey();
            return
(key != null && key.equals(orderDto.getApiKey())
);
        } catch (Exception e) {
            return false;
        }
    }
}
```

Метод isAuthorizedDto виконує запит до бази даних за допомогою Spring репозиторію та їх методів запитів. У нашому випадку це створений метод findByTikenAndApiKey. Отримавши, за запитом, ключ ми порівнюємо їх та визначаємо чи валідні дані.

Метод isUniqueOrder виконує запит до бази даних за допомогою Spring репозиторію та їх методів запитів. У нашому випадку це створений метод findByOrderProviderIdAndTaxiTokenFrom. Отримавши, за запитом, об’єкт замовлення ми перевіряємо його на існування.

Можемо побачити, що усі запити до бази виконуються без явного написання SQL коду. Така можливість є завдяки таким фреймворкам, як Hibernate та Spring Data. Перший фреймворк, відповідно, виконує відображення об’єктно-орієнтованих моделей на звичайні реляційні бази даних. Другий фреймворк потрібен для полегшення доступу до даних, шляхом мінімізації використання

SQL коду в шарі доступу до даних.

Для того, щоб створити репозиторії з методами доступу до даних потрібно описати сутності, що відповідають таблицям в базі даних. До прикладу сутність Receivers з вихідним кодом:

```
@Entity
@Table(name = "receivers")
public class Receivers {
    @Id
    @Column(name = "id")
    private Integer id;
    @ManyToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "receive_from", referencedColumnName = "id")
    private TaxiService taxiReceiveFrom;

    @ManyToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "taxi_id", referencedColumnName = "id")
    private TaxiService taxiService;

    @Column(name = "enabled")
    private Boolean enabled = false;

    //constructors, getters and setters
}
```

Анотація Entity має ціль на класи та працює в Runtime. Її роль полягає в помітці класу як сутності для Hibernate. Тобто анотація є маркером. Для уточнення таблиці, на яку йде відтворення моделі, використовується анотація Table. Дана анотація також є маркером та працює в Runtime. Вона приймає текстовий параметр, в якому описано назву таблиці в базі даних.

Анотація Id є маркером, що повідомляє про поле, як ідентифікатор. Така анотація може бути одна на сутність. Column – анотація маркер, яка повідомляє про те на яке поле посилається властивість сутності. В текстовому параметрі передається назва поля в таблиці.

Для поєднання таблиць через зовнішні ключі є декілька способів. В нашому випадку це через такі анотації:

− OneToOne. Дана анотація відображає зв’язок один до одного. Потрібно в текстовий параметр вказати назву поля в відповідній сутності до якого йде зв’язок

для відтворення відношення;

− OneToMany та ManyToOne. Анотація зв’язку один до багатьох. Потрібно в текстовий параметр вказати назву поля в відповідній сутності до якого йде зв’язок. Також застосувати анотацію JoinColumn де вказуються відповідні поля по який йде підключення;

− ManyToMany. Така анотація реалізовує відношення багато до багатьох. Вона має усі особливості як і в інших анотаціях. Але використовується в парі з анотацією JoinTable. В ній проставляються поля по яких йде зв’язок та по інвертовані поля, що проміжній таблиці;

Коли ми маємо прописані усі сутності ми можемо створити класи репозиторії, що реалізовують методи доступу до даних. Клас репозиторій CurrentOrdersRepository з методами доступу до даних:

```
package com.fastaxi.Aggregator.reposetory;

import com.fastaxi.Aggregator.entity.CurrentOrder;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import java.util.List;

@Repository
public interface CurrentOrderRepository extends
JpaRepository<CurrentOrder, Integer> {

    CurrentOrder findByOrderProviderIdAndTaxiTokenFrom(Integer
aggregatorId, String taxiTokenFrom);

    List<CurrentOrder> findByTaxiServiceTakenProtocolId(Integer
protocolId);

}
```

Особливістю реалізації такого доступу до даних є можливість відсутності SQL коду, що полегшує підтримку. Клас репозиторій для реалізації можливості виконання запитів до даних через, попередньо налаштовані методи, повинен реалізувати один з інтерфейсів, що пропонує Spring Data. У нашому випадку інтерфейс JpaRepository пропонує набір стандартних методів таких як:

```
List<T> findAll();
List<T> findAll(Sort var1);
List<T> findAllById(Iterable<ID> var1);
<S extends T> List<S> saveAll(Iterable<S> var1);
void flush();
<S extends T> S saveAndFlush(S var1);
void deleteInBatch(Iterable<T> var1);
void deleteAllInBatch();
T getOne(ID var1);
<S extends T> List<S> findAll(Example<S> var1);
<S extends T> List<S> findAll(Example<S> var1, Sort var2);
```

Такі методи дають виконувати усі CRUD операції. Також відкривається можливість написання власних методів для виконання запитів. Розглянемо створення запитів через методи на прикладі, що зображено на рисунку 3.8.

Рисунок 3.8 – Схема побудови метода запиту

Рисунок 3.8 зображує сигнатуру методу запиту де кожна частина відповідає певному номеру:

– тип, що поверне запит. Головною умовою є те, щоб тип відповідав сутності або полям сутності для якої створено репозиторій. Фреймворк самостійно вміє визначити та перетворити дані запиту в сутність мови програмування на якій виконується запит;

– в назві методу слово find говорити фреймворку про те, що потрібно зробити запит SELECT у відповідну таблицю;

– слово By повідомляє, що є умови по який потрібно зробити фільтрацію під час виконання запиту;

– властивість, що знаходиться в сутності, яку шукають;

– сигнатура, що відповідає кількості та типам умов у назві методу.

Шар представлення реалізований у вигляді контролеру, що перехоплює запити на сервер по відповідних шляхах та виконує логіку, яку йому надає OrderService. Повний код в лістину В.1. Приклад коду контролеру:

```
@RestController
public class Controller {

    private static final Logger logger =
LogManager.getLogger(Controller.class);
    private final OrderService orderService;

    public Controller(OrderService orderService) {
        this.orderService = orderService;
    }

    @PostMapping(
        path = "orderTake",
        produces = MediaType.APPLICATION_JSON_VALUE,
        consumes = MediaType.APPLICATION_JSON_VALUE
    )
    public RestApiResponse takeOrder(@RequestBody @Valid
TakeOrderDto order) {
        logger.info("Request to api/orderTake with info - " +
order.toString());
        return orderService.takeTakenOrder(order);
    }
}
```

Анотація RestController існує для зручного створення RESTful веб сервісів. Вона поєднує в собі анотації Controller та ResponseBody, що дозволяє не помічати всі методи за для вміння відповідати на всі запити [14] Приклад контролеру в.

Controller анотація використовується для помітки класів представлення або бізнес логіки. Коли виконується запит на сервер, сповіщається DispatcherServlet для сканування класів з анотацією Controller. Відбувається перевірка на наявність методів з анотаціями Post(та інші типи запитів)Mapping або RequestMapping.

У нашому випадку PostMapping виконує зіставлення запиту POST по HTTP до методу, що виконує зміну статусу замовлення. Для всіх інших запитів існує відповідна анотація. Такі анотації мають перелік важливих параметрів таки як:

- Path – текстове поле відповідає за уточнення шляху виконання запиту;
- Produces – поле де вказується тип серіалізації даних при відповіді;
- Consumes – поле де вказується тип серіалізації даних при запиті;

Анотація ResponseBody відповідає за запуск серіалізації в момент повернення відповіді з методу. Тобто при поверненні певного DTO алгоритми серіалізації перетворюють його в той тип що був вказаний в Produces.

Для виконання логування використовується бібліотека log4j. Даний спосіб є дуже зручний за рахунок великого спектру налаштувань рівнів логування та простоти використання. Приклад налаштувань логів:

```
//тут визначені поля що будуть використовуватись як патерни при
записі логів
<Properties>
        <Property name="LOG_PATTERN_CONSOLE">%clr{%d{yyyy-MM-dd
HH:mm:ss.SSS}}{faint} %clr{%5p} %clr{${sys:PID}}{magenta} %clr{---
}{faint} %clr{[%5.5t ]}{faint} %clr{%-40.40c{1.}}{cyan}
%clr{:}{faint} %m%n%xwEx</Property>
        <Property name="LOG_PATTERN_FILE">[%-5level] %d{yyyy-MM-dd
HH:mm:ss.SSS} [%t] %c{1} - %msg%n</Property>
        <Property name="LOG_PATH">logs</Property>
</Properties>
// налаштування логу для рівню DEBUG
<RollingFile name="DebugLog-Appender"
                     fileName="${LOG_PATH}/application-debug.log"
                     filePattern="${LOG_PATH}/application-debug-
%d{yyyy-MM-dd}-%i.log">

            <!-- Matches only DEBUG level -->
            <LevelRangeFilter minLevel="DEBUG" maxLevel="DEBUG"
onMatch="ACCEPT" onMismatch="DENY"/>

            <PatternLayout pattern="${LOG_PATTERN_FILE}"/>
            <Policies>
                <SizeBasedTriggeringPolicy size="19500KB" />
            </Policies>
            <DefaultRolloverStrategy max="10"/>
</RollingFile>
```

Головною особливістю проекту є розподілення замовлення. Розміщене воно в класі OrderService методу takeOrderForSearchingdDriver. За логікою виконується перетворення тіла DTO в сутність для збереження в базу даних. Так звані трансформатори або мапери є залежністю класу OrderService. Весь код в лістингу В.2. Приклад алгоритму перетворення об’єкту передавачі даних до сутності:

```
public CurrentOrder toEntity(CurrentOrderDto order) {
    CurrentOrder currentOrder = new CurrentOrder();
    currentOrder.setCreatedDt(order.getCreatedDt());
    ...
    return currentOrder;
}
```

Після збереження інформації про замовлення та сервісу таксі, що є провайдером виконується запит на знаходження налаштувань розподілення. Спершу виконується запит на знаходження сервісів кому можна передавати а після сервіси що можуть приймати. З двох множин знаходиться кон'юнкція. У висновку є колекція з інформацією про сервіси таксі, які готові прийняти замовлення від провайдера. Приклад алгоритму знаходження налаштувань:

```
List<Transfers> transfers =
transfersRepository.findByTaxiServiceIdAndEnabledIsTrue(taxiService.
getId());//список таксі кому можна передати
List<Integer> receivers =
receiversRepository.findByTaxiReceiveFromIdAndEnabledIsTrue(taxiServ
ice.getId())
   .stream()
   .map(receiver -> receiver.getTaxiService().getId())
   .collect(Collectors.toList());//список таксі що можуть приймати
List<Transfers> transferTo = transfers.stream()
   .filter(transfer ->
receivers.contains(transfer.getTaxiTransferTo().getId())
         )
   .collect(Collectors.toList());//фільтрація сервісів шляхом
знаходження кон'юнкції
```

Для роботи з колекціями використовується так зване Stream API [15]. Під собою воно має набір методів та алгоритмів, що спрощують виконання різних дій над елементами колекцій. Основні методи, що використовуються:

– stream – перетворює колекцію в потік елементів типу колекції;

– filter – на вхід приймає лямбду з об'єктом колекції а на вихід певну умову, що говорить чи підходить об'єкт;

– skip – дає можливість пропустити певну кількість елементів;

– distinct – повертає stream без дублікатів;

– map – дає можливість перетворити тип stream;

– peek – повертає stream того ж типу але виконує для кожного об'єкту певну дію, що прописана в функції;

– limit – дозволяє сортувати за певним простим порядком або за написаним Comparator функціоналом.

Кожен з вище описаних методів повертає stream і з кожного методу можна викликати інший. А існують методи, що називаються термінальними. Вони приводять потік в певний оговорений тип. До прикладу:

− findFirst – повертає перший елемент з stream, повертає Optional;

− findAny – повертає любий елемент з stream. Якщо stream має один потік то цим елементом буде перший, якщо stream паралельний – елемент буде випадковим. Повертає Optional;

− collect – виконує представлення результату роботи в вигляді колекцій;

− anyMatch – повертає true, якщо умова справджується хоча б для одного елементу колекції;

− noneMatch – повертає true, якщо умова не виконується ні для одного елементу колекції;

− min – повертає мінімальний елемент використовуючи компаратор;

− max – повертає максимальний елемент використовуючи comparator;

− forEach – виконує функцію до кожного елементу потоку. Порядок виконання не гарантується;

− forEachOrderd – виконує функцію до кожного елементу потоку. Порядок виконання гарантується;

− toArray – повертає масив елементів з stream;

− reduce – дозволяє виконувати агрегатні функції на всіх елементах.

Для реалізації розподілення використовується бібліотека Concurrent. Даний пакет пропонує різноманітний функціонал для роботи з багатопотоковістю. Бібліотека поділяється на декілька під пакетів:

− Concurrent collections;

− Queues;

− Synchronizers;

− Executors;

− Locks;

− Atomics.

Concurrent collections призваний замінити звичайні колекції в роботі з

багатопотоковістю. Звичайна колекція не дозволяє одночасно читати та видаляти дані в різних потоках. Що означає з колекцією може працювати лише один потік.

Колекції з пакету concurrent можуть працювати з різними потоками одночасно. Ітератори, що використовуються для колекцій цього пакету не викликають ConcurrentModificationException. Кожна операція створює новий маси даних, що дозволяє не викликати виключення.

Queues пропонує черги, які підтримують багатопотоковість. Черги поділяються на блокуючі та неблокуючі. Неблокуючі потоки налаштовані для швидкого виконання без можливості блокування. Блокуючі потоки потрібні для більш чіткого регулювання виконання дій з можливістю паузи або зупинки по певних умовах або по інтервалу.

Synchronizers надає певний набір утиліт для контролю та синхронізації потоків. До прикладу, вони корисні в складних обрахунках.

Locks потрібен для реалізації обмеження певних блоків ресурсів. Як альтернатива для notify, notifyAll, wait, synchronized.

Atomic набір класів з підтримкою атомарних операцій з примітивами.

Executors потрібен для організації набору задач з регулюванням послідовності виконання та планування, що використовуватиметься в алгоритмах розподілення замовлення. Executors працює з такими інтерфейсами, як Future, Delayed, Callable.

Future має ціль отримати результати асинхронної операції. Ключовим методом тут являється get(). Він зупиняє потік в якому запущена асинхронна операція та очікує на завершення по timeout або без нього.

Callable аналог Runnable для асинхронних дій. Дозволяє повертати типізовані дані з тіла асинхронної операції.

Delayed інтерфейс потрібен для асинхронних задач, що мають почати виконуватись в майбутньому. Має можливість встановлення часу до початку асинхронної задачі.

Для роботи з потоками пропонується такий набір Executor services:

− Executor – корінний інтерфейс для запуску Runnable задач у методі run;

– ExecutorService – інтерфейс описує запуск як Runnable так і Callable задач. Метод виконання задач, submit, приймає певну задачу та повертає Future. Метод shutdown припиняє прийом задач та завершує роботу.

– ScheduledExecutorService – доповнює інтерфейс ExecutorService методами відкладеного запуску задач.

– AbstractExecutorService – базова реалізація всіх Executors.

Для створення реалізацій executor використовується клас фабрика Executors. Він пропонує реалізацію ThreadPoolExecutor та ScheduledThreadExecutor. Через додаткові getters та setters можна міняти ключові параметри.

ThreadPoolExecutor використовується для виконання асинхронних задач в пулі потоків. За рахунок точно визначеної кількості потоків можна визначити завантаженість та продуктивність.

ScheduledThreadPoolExecutor унаслідує всі властивості ThreadPoolExecutor та дає можливість встановлювати затримку виконання задач. А також встановлювати періодичність запуску [16].

Для складеного списку сервісів таксі, що був описаний вище ми описуємо задачі в список Callable де кожна задача це процес надсилання замовлення до певного сервісу таксі. Приклад складання списку задач:

```
CurrentOrderDto orderTransfer = orderMapper.toDto(order);
transferTo.forEach(transfer ->
tasksForSendRequestToOtherServices.add( () -> {
        TaxiService taxi = transfer.getTaxiTransferTo();
        if (taxi.getProtocolId() == 0) {
            TaxiServiceDto taxiServiceDto;
            taxiServiceDto =
scheduledExecutorServiceToTaxi.schedule(() -> {
                if
(!taxi.getToken().equals(orderTransfer.getTaxiTokenFrom())) {
            orderTransfer.setTaxiToken(taxi.getToken());
return webClientService.sendOrderToTaxiService(
                    orderTransfer,
                    taxi.getUrl(),
                    taxi.getPort(),
                   "aggregator/searchDriver"
                    );
              } else {
                    TaxiServiceDto provider = new TaxiServiceDto();
```

```
                    provider.setConfirmed(false);
                    return provider;}
                }, transfer.getDelay(), TimeUnit.SECONDS).get();
          TimeUnit.SECONDS.sleep(transfer.getDelay());
          return taxiServiceDto;
          }
        TaxiServiceDto provider = new TaxiServiceDto();
        provider.setConfirmed(false);
        return provider;
    })
);
```

Формується об’єкт для сервісу таксі. Запускається потік з найдених сервісів таксі. На кожен сервіс таксі створюється задача, що записується в список задач. В задачі описано визначення протоколу по якому виконується надсилання. Протокол описаний лише один, задля полегшення в розумінні алгоритму. В даному способі надсилання закладений scheduledExecutorService. Він виконуватиме надсилання через відведену затримку з налаштувань. В налаштуваннях також знаходиться доменне ім’я серверу таксі та порт. Шлях надсилання декларується в протоколі і з підключеним сервісом таксі.

В момент виконання завдань обробляється відповідь від сервісів. Приклад обробки відповіді:

```
responsesFromServices =
executor.invokeAll(tasksForSendRequestToOtherServices)
      .stream()
      .map(pairFuture -> {
             try {return pairFuture.get();
             } catch (InterruptedException e) {
                 log.error(e.getMessage());
             }
             return new TaxiServiceDto();
       })
.collect(Collectors.toList()
);
```

Усі відповіді збираються в єдиний список де потім проходить перевірку на наявність відповіді. Якщо немає відповіді то вона рахується негативною інакше перевіряється чи сервіс взяв замовлення в обробку. Якщо ніхто в обробку не взяв замовлення видаляється інакше запускається таймер після якого замовлення без

водія видалиться. Приклад обробки відповідей:

```
if (responsesFromServices != null &&
!responsesFromServices.isEmpty())
{List<TaxiServiceDto> checkForResponse =
responsesFromServices.stream().filter(Objects::nonNull).filter(taxiS
erviceDto -> taxiServiceDto.isConfirmed() != null &&
taxiServiceDto.isConfirmed()).collect(Collectors.toList());
            boolean driverSearching = !checkForResponse.isEmpty();
            restApiResponse.setConfirmed(driverSearching);
            if (!driverSearching)
currentOrderRepository.delete(order);
            else deleteOrderByTimeout(orderDto, order);}
```

Отже, було розроблено систему, що відповідає поставленій меті. Пояснені всі основні модулі програмного комплексу та визначені особливості реалізації, які вказують на унікальність сформованого рішення.

3.3 Керівництво користувача

Сервіс таксі, що хоче отримати доступ до системи розподілення замовлень має підключити API та отримати ключ доступу. Ключ доступу генерується власником системи.

Для роботи з агрегатором потрібно обумовити формат даних в яких буде передаватись інформація. Усі запити мають певний набір полів які обов’язково мають бути. Кожен запит має конкретний тип та шлях. Список запитів представлено в таблиці 3.1.

При надсиланні замовлення для розподілення потрібно запевнитись, що є уся інформація про замовлення та дані доступу. Замовлення має містити інформацію про клієнта, таку як:

− особистий номер в системі;

− номер телефону;

− ім’я.

Також має бути інформація про маршрут:

− початкова точка з координатами;

− опціонально можна надіслати проміжні точки та кінцеву.

У випадку якщо є інформація про кінцеву точку потрібні такі дані:

− прорахована вартість поїздки;

− уточнення що до кінцевої точки.

Обов’язковим передається тариф за яким клієнт замовляв послугу. Від цього залежить машина, що зможе взяти замовлення на виконання. В тарифі мають бути враховані погодні умови та характеристики які налаштовано в сервісі таксі засобами адміністрування. Також мета-інформація, така як час пошуку, комісія, токени, номер замовлення.

При виконанні запиту на взяття замовлення потрібно передати інформацію про водія, який буде виконувати замовлення. Обов’язковими полями є:

− ім’я водія;

− номер телефону;

− номер автомобіля;

− чи є шашка на криші автомобіля;

− колір автомобіля;

− тип автомобіля;

− рейтинг водія;

− ранг водія.

Таблиця 3.1 – API агрегатора

Назва запиту	HTTP метод	URL	Вхідні дані	Вихідні дані
1	2	3	4	5
Надіслати замовлення для розподілення	POST	/sendOrder	Модель замовлення Б.1	`Boolean isTakenForSearching Driver;`

Кінець таблиці 3.1

1	2	3	4	5
Взяти замовлення на виконання	POST	/orderTake	Модель Б.2	`Boolean isTaken;`
Водій приїхав на замовлення	POST	/orderAtAddress	`String apiKey; String taxiToken; String taxiTokenFrom; Integer aggregatorId;`	`Boolean isConfirmed;`
Водій завершив поїздку	POST	/finishTrip	Модель Б.3	`Boolean isConfirmed;`
Клієнт відмінив замовлення	POST	/cancelOrder	`String apiKey; String taxiToken; String taxiTokenFrom; Integer aggregatorId; String cancelReason; Long orderId; Integer statusId;`	`Boolean isConfirmed;`
Водій почав поїздку з пасажиром	POST	/startTrip	`String apiKey; String taxiToken; String taxiTokenFrom; Integer aggregatorId;`	`Boolean isConfirmed;`

Для того, щоб запустити систему потрібно мати комп'ютер з операційною системою, що підтримує java 11. В командному рядку системи виконати спеціальну команду для запуску архіву java. Також можна скористатись вбудованими утилітами операційної систем якщо такі є в наявності. Також потрібно запустити базу даних Postgres, що йде в комплекті з системою.

Отже, були сформовані інструкції для установки та використання системи. А саме з цілю використання системи, як стороннього сервісу таксі. Сформульовано процес виконання замовлення.

3.4 Вимоги до технічних та програмних засобів

Систему розраховано на високе навантаження але для цього потрібна висока продуктивність комп'ютера. Агрегатор можна запустити на простій системі при таких мінімальних параметрах:

- ОС Linux/Windows (на Windows продуктивність Агрегатора менша через те, що ресурси комп'ютера також витрачаються на підтримку операційної системи на відміну від Linux).
- будь-який двоядерний процесор з частотою на ядро 2.4 ГГц.
- 4 ГБ ОЗП;
- 20 МБ постійної пам'яті.

Рекомендовані вимоги спираються на те, що до системи буде 50 запитів на хвилину або 20 одночасних користувачів на 10 замовлень в годину:

- ОС Linux;
- будь-який шестиядерний процесор з частотою на ядро 3 ГГц;
- 4 ГБ ОЗП;
- 100 МБ постійної пам'яті (залежить від типу диску).

Спираючись на тип продукту було визначено технічні вимоги та сформульовані характеристики для комфортного користування системою. Було визначено на якій операційній системі оптимально використовувати програмний продукт та вказані мінімальні та оптимальні характеристики техніки.

У результаті реалізації програмного продукту для автоматизації розподілення замовлень між різними сервісами таксі було описано всі функціональні модулі та оптимальні алгоритми, що виконують поставлену задачу. Код основних модулів описано в Додатку В. На основі описаного програмного забезпечення була побудована чітка інструкція для користувача, якого представляє сервер таксі сервісу. Та для адміністратора, що є власником системи.

4 ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1 Аналіз та вибір методів тестування додатку

Для вибору методу тестування потрібно визначити ціль тестування. У нашому випадку від цілі тестування залежить результат реалізації. Загалом існує три види тестування, що залежать від цілі:

- функціональне;
- нефункціональне;
- пов’язане з змінами.

Функціональні види тестування оцінюють зовнішню поведінку системи або іншими словами перевіряється те як система реагує на взаємодію з різними чинниками. Використовуються для визначення не правильної поведінки системи. Такі тести керуються особливостями функцій системи та взаємодією з іншими системами. Реалізувати такі тести можна на всіх рівнях:

- модульне;
- інтеграційне;
- системне.

Модульне тестування призначене для визначення коректності роботи мінімально можливої частини програми. Це може бути метод або клас, загалом такі компоненти можливо тестувати ізольовано.

Інтеграційне тестування має ціль визначити коректність роботи окремих модулів у взаємодії. Такі модулі були, попередньо, протестовані за допомогою модульних тестів.

Системне тестування виконує перевірку програмного продукту, як одного цілого. Таке тестування проводить після завершення роботи над основним фунціоналом програмного комплексу.

Нефункціональне тестування під собою має ціль охарактеризувати систему в різних вимірюваних величинах. Іншими словами такі тести перевіряють якість роботи системи. Загалом це такі види тестування:

− тестування на навантаження;

− тестування на стабільність;

− тестування на пропускну здатність;

− тестування зручності.

Тести, що пов’язані з змінами мають ціль перевірити робото здатність системи після виправлення певних помилок або внесення покращень. Для того щоб правильно перевірити зміни, потрібно провести один або декілька з таких видів тестування:

− димове тестування;

− регресивне тестування;

− перевірка узгодженості.

Спосіб тестування обирається в залежності від підходу. Існують, загалом, загалом такі підходи:

− білий ящик;

− чорний ящик.

Методом білого ящику розробник може написати код, що буде напряму тестувати окремі компоненти програми. Іншими словами є доступ до вихідного коду. Таким чином можна описати модульні або інтеграційні тести.

Методом чорного ящику немає доступу до вихідного коду. Тестувальник має доступ до програми у вигляді інтерфейсу, який пропонується користувачу, іншій системі або комп’ютеру. До прикладу, спеціальний модуль може симулювати натиски миші або клавіатури по окремим областям додатку, що автоматизує ручне тестування коду. Усі тестування проводяться згідно з специфікаціями та інструкціями, що йдуть до програмного продукту [17].

Звертаючи увагу на інтерфейс системи, а ним являється API. Ми можемо зрозуміти, що ручне тестування буде досить складним але воно необхідне для такого типу доступу до системи. Адже на кожен елемент інтерфейсу може виконуватись не санкціонованим запитом, що потрібно передбачити.

У системі є модулі, що потребують тести, які перевірять правильність виконання бізнес-логіки. Такі тести є функціональними, а саме модульні та

інтеграційні. З цього ми можемо зрозуміти, що потрібно провести як ручне так і автоматизоване тестування, що покаже реакцію системи на взаємодію компонентів разом і окремо.

4.2 Тестування програмної системи

Першим модулем для тестування є шар представлення за який відповідає клас Controller, що пропонує інтерфейс взаємодії. Набір тестових випадків для даного модулю зображено в таблиці 4.1.

Таблиця 4.1 – Набір тестів для шару представлення

№	Функція	Вхідні дані	Очікуваний результат
1	2	3	4
С-М-1	Передача замовлення	Інформація про замовлення, дані доступу	Якщо дані доступу вірні буде виконано відповідну бізнес логіку. Якщо дані замовлення вірні то буде виконано відповідну бізнес логіку. Якщо дані не вірні то буде відмовлено у виконанні подальших дій.
С-М-2	Помітка замовлення про взяття	Інформація про водія, дані доступу.	Якщо дані доступу вірні буде виконано відповідну бізнес логіку. Якщо дані водія вірні то буде виконано відповідну бізнес логіку. Якщо дані не вірні то буде відмовлено.
С-М-3	Помітка замовлення про те що водій по адресу	Інформація про позицію водія, дані доступу.	Якщо дані доступу вірні буде виконано відповідну бізнес логіку. Якщо дані водія вірні то буде виконано відповідну бізнес логіку. Якщо дані не вірні то буде відмовлено.

Кінець таблиці 4.1.

1	2	3	4
С-М-4	Помітка замовлення про те що водій почав поїздку	Дані доступу.	Якщо дані доступу вірні буде виконано відповідну бізнес логіку. Якщо дані доступу не вірні буде відмова у виконанні дії.
С-М-5	Помітка замовлення про те що водій завершив поїздку	Дані доступу, дані про завершену поїздку.	Якщо дані доступу вірні буде виконано відповідну бізнес логіку. Якщо дані про поїздку вірні то буде виконано відповідну бізнес логіку. Інакше відміна.
С-М-6	Помітка замовлення про те що воно відмінене	Дані доступу, дані про відміну замовлення	Якщо дані доступу вірні буде виконано відповідну бізнес логіку. Якщо дані про відміну вірні то буде виконано відповідну бізнес логіку. Якщо дані не вірні то буде відмовлено у виконанні подальших дій.

Шар бізнес-логіки представляє клас OrderService та всі допоміжні сервіси. Тут усі випадки тестування перевірятимуть логіку виконання дій по замовленню та їх розподілення. Кожен тестовий випадок виконує дію над конкретною зміною статус замовлення а також відтворює схему розподілення та відміну. Тестові випадки зображено в таблиці 4.2.

Таблиця 4.2 – Набір тестів шару бізнес-логіки

№	Функція	Вхідні дані	Очікуваний результат
1	2	3	4
OS-M-1	Знайти сервіс таксі, що буде шукати водія	Інформація про замовлення, дані доступу	Якщо дані замовлення вірні буде виконано логіку збереження замовлення, фільтрування налаштувань, розподілення, розсилання та обробка відповідей від сервісів таксі, що активні та в стані обробити замовлення. Якщо дані доступу вірні то буде виконана логіка

Кінець таблиці 4.2.

1	2	3	4
OS-M-2	Зберегти інформацію про водія, що буде виконувати замовлення	Інформація про водія, дані доступу	Якщо дані доступу вірні буде дозволено виконати логіку. Якщо дані водія вірні то буде збережено водія до бази даних та буде сповіщений сервіс провайдера про те що водій знайшовся. Водію стане доступно виконання замовлення. Якщо дані не вірні то буде відмовлено.
OS-M-3	Зберегти інформацію про те що водій по адресу	Інформація про позицію водія, дані доступу.	Якщо дані доступу вірні буде дозволено виконати логіку. Якщо дані водія вірні буде оновлено інформацію в базі даних та буде сповіщений сервіс провайдера про те що водій приїхав за адресою. Якщо дані не вірні то буде відмовлено у виконанні подальших дій.
OS-M-4	Зберегти інформацію про те що водій почав поїздку	Дані доступу.	Якщо дані доступу вірні буде дозволено виконати логіку. Буде сповіщений сервіс провайдера про те що водій розпочав поїздку. Якщо дані не вірні то буде відмовлено у виконанні подальших дій.
OS-M-5	Зберегти інформацію про те що водій завершив поїздку	Дані доступу, дані про завершену поїздку.	Якщо дані доступу вірні буде дозволено виконати логіку. Якщо дані завершення поїздки вірні буде оновлено інформацію в базі даних та буде сповіщений сервіс провайдера про те що водій завершив поїдку. Якщо дані не вірні то буде відмовлено у виконанні подальших дій.
OS-M-6	Зберегти інформацію про те що замовлення відмінили	Дані доступу. Замовлення	Якщо дані доступу вірні буде дозволено виконати відміну замовлення. Буде сповіщений сервіс провайдера про те що замовлення відмінено. Якщо дані не вірні то буде відмовлено у виконанні подальших дій.

Для ручного тестування API використовується фреймворк Swagger. Він надає можливість автоматичної генерації документації та специфікацій по REST API. Також можливо надсилати запити до задокументованого API з можливістю перегляду коректності відповіді. При тестуванні усі тестові випадки повторюють випадки при автоматизованому тестуванні.

Така документація буде корисною як для майбутніх розробників так і для користувача, що захоче скористатись API. До всіх запитів є опис, що допоможе розібратись без втручання власника системи. На рисунку 4.1 зображено приклад запиту до системи. На рисунку 4.2 показані типи відповідей.

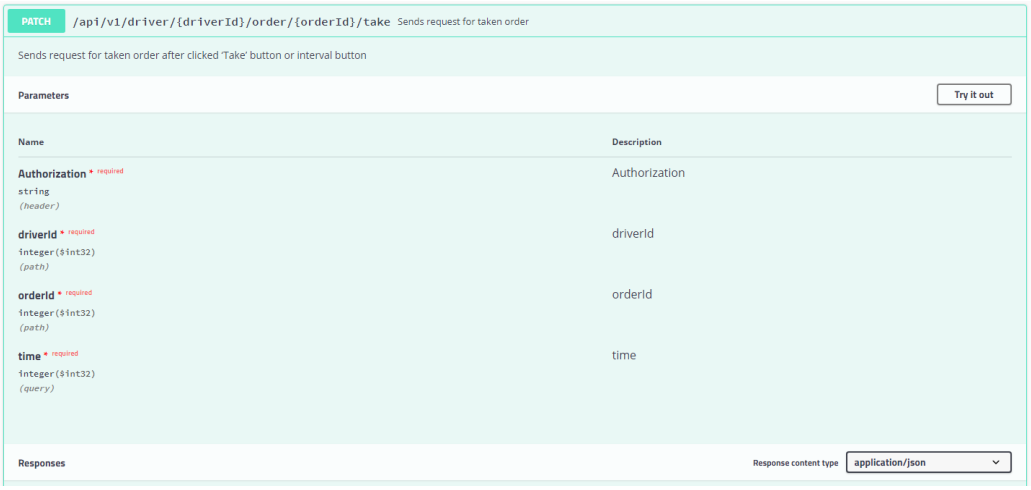


Рисунок 4.1 – Запит у Swagger

Рисунок 4.2 – Відпоіді на запит у Swagger

В результаті були побудовані тестові сценарії для системи, що покажуть ефективність та правильність роботи у різних випадках. Описані визначені способи тестування та відображено підходи реалізації тестування.

4.3 Аналіз результатів тестування системи

Для того, щоб система була протестована максимально ефективно було застосовано автоматичне та ручне тестування. Для всіх запитів були описані сценарії, що відтворювали дії користувача. При ручному тестуванні були проведені менш очевидні дії, що допомогли виправити роботу логіки. Результати варіантів тестування зображено на таблиці 4.3.

Таблиця 4.3 – Набір тестів шару бізнес-логіки

№	Вхідні дані	Очікуваний результат	Відмінність від очікуваного результату
1	2	3	4
C-M-1	Інформація про замовлення, дані доступу	Дані доступу та дані про замовлення являються валідними.	Не виявлено
C-M-2	Інформація про водія, дані доступу.	Дані доступу та дані про водія являються валідними.	Не виявлено
C-M-3	Інформація про позицію водія, дані доступу.	Дані доступу та дані про позицію являються валідними.	Не виявлено
C-M-4	Дані доступу.	Дані доступу являються валідними.	Не виявлено
C-M-5	Дані доступу, дані про завершену поїздку.	Дані доступу та дані про завершення поїздки являються валідними.	Не виявлено
C-M-6	Дані доступу, дані про відміну замовлення	Дані доступу та дані про відміну замолвення являються валідними.	Не виявлено
OS-M-1	Інформація про замовлення, дані доступу	Замовлення збережене та знайдено сервіс таксі, що шукає автомобіль.	Не виявлено
OS-M-2	Інформація про водія, дані доступу	Водія збережено до бази даних та провайдер замовлення був сповіщений про знайденого водія.	Не виявлено
OS-M-4	Дані доступу.	До бази збережена помітка про те, що водій почав поїздку по замовленню та провайдер був сповіщений про це.	Не виявлено

Кінець таблиці 4.3

1	2	3	4
OS-M-3	Інформація про позицію водія, дані доступу.	В базу даних збережено помітка про те, що водій прибув по адресу.	Не виявлено
OS-M-5	Дані доступу, дані про завершену поїздку.	До бази збережено те, що водій завершив поїздку по замовленню та провайдер був сповіщений про це.	Не виявлено
OS-M-6	Дані доступу. Замовлення	До бази збережена помітка про те, що замовлення відмінене та всі сервіси, що його отримали були сповіщені.	Не виявлено

В результаті етапу тестування були проаналізовані методики тестування програмного забезпечення. Визначені способи тестування для наявного дипломного проекту. Протестовані основні модулі представлення та бізнес-логіки. На основі створених тестових випадків можна зробити висновки про те, що програмний комплекс працює за заявленими вимогами.

ВИСНОВКИ

Сьогодні сфера обслуговування людей у галузі таксі має велику популярність, яка росте за рахунок якісного обслуговування та легкого доступу. Адже на сучасному етапі розвитку людства існує різноманітне програмне забезпечення, що спрощує багато процесів та бізнес-потреб таксі.

Загалом таксі сервіс, в технічному плані, складається з серверу, мобільного додатку водія, мобільного додатку клієнта, панель адміністратора та робочої частини оператора. Зараз, щоб увійти у ринок таксі послуг, потрібен саме такий набір. Але яке б нове, інноваційне, оригінальне програмне забезпечення не було у сервісу таксі завжди присутній людський вплив. Сучасний таксі сервіс як і десять років тому не може існувати без людського контролю. Завжди є присутній адміністратор або оператор який контролює хід роботи ну і звісно ж водії.

Спираючись на присутність людського фактору можна виокремити водіїв, що впливають на бізнес таксі найбільше. Наразі вони є основною одиницею роботи любої служби. І з цього виходить, що вони мають, порівняно, високий вплив на компанію та прибуток. Що створює головну проблему, таку як нехватка кадрів серед водії, що популярно не лише в таксі.

Темою дипломного проекту являється «Програмне забезпечення для автоматизації обміну замовленнями між різними сервісами таксі», що має на меті об’єднати сервіси таксі в єдину мережу, для пришвидшення автоматичного пошуку водія для клієнта в межах конкретного міста. Такий підхід покликаний вирішити проблему нехватки кадрів в бізнесі таксі.

Згідно з проведеним аналізом предметної області було встановлено, що відкриті сервіси автоматичного розподілення замовлень таксі є досить тонко направленою сферою. Структура таких застосунків залежить від існуючих сервісів таксі та інших факторів. Аналіз існуючого програмного забезпечення показав, що є багато різного роду програмних комплексів за кордоном та ні одного відкритого в Україні. Або такі комплекси мають обмежений доступ для

окремих лиць. Усі такі програмні комплекси маю свої позитивні сторони але і більш явні мінуси, а саме застарілість технологій та підходу до вирішення проблеми. Кожен такий сервіс хоче, в першу чергу, заробити а потім виконати поставлену мету.

Під час етапу проектування були проаналізовані типи архітектури а саме клієнт-серверна. Аналіз показав, що сервер є певним комп'ютером, що має високу потужність. Програмним забезпеченням серверу регламентується протокол передачі даних. В нашому випадку було проаналізовано та обрано HTTP. Для інтерфейсу було обрано такий архітектурний підхід, як REST. В залежності від обраної архітектури проаналізували шаблони проектування. А саме монолітний та мікросервісний шаблон. Обраним був монолітний шаблон проектування, так як сервіс, який буде результатом завдання не є великого розміру. Він легко масштабується та не потребує складних тестів.

При аналізі баз даних було визначено, що для поставленої мети реляційна база даних підходить найбільше. Даний вибір впав на такий тип СКБД за рахунок зручності доступу, швидкістю фільтрації та організації даних. Аналізуючи наявні реляційні бази даних вибір впав на Postgres за рахунок великої кількості якісної документації та великого спектру налаштувань.

На етапі опису декомпозиції була визначена трирівнева архітектура системи. Вибір впав на таку архітектуру за рахунок можливостей розділення різних типів функціоналу та захист від SQL ін'єкцій. Також спроектовані пакети з відповідним функціоналом.

Під час опису залежностей та інтерфейсів визначені всі функціональні властивості до програмної системи. Побудована ієрархія залежностей між усіма компонентами. Встановлені інтерфейси взаємодії компонентів.

Згідно з проаналізованими методами та технологіями реалізації подібних систем були обрані інструменти реалізації серверного програмного забезпечення. А саме мова програмування java у взаємодії з Spring.

У процесі реалізації було створено базу даних. Відповідно проаналізованій предметній області база даних має містити набір таблиць з мінімальною кількістю

посилань один на одну для швидшого виконання запитів. Після чого було реалізовано API з урахуванням функціональних та нефункціональних вимог разом з модельним рядом доступу до даних.

Після реалізації було описано керівництво користувача системи з поясненням підключення API. Визначені технічні вимоги.

Під час етапу тестування було проаналізовано методи тестування, що залежать від цілі. Визначені характеристики окремо кожного методу. Також описані тести, що пов’язані з змінами. В залежності від підходу та можливостей були обрані такі способи тестування, як «білий ящик» та «чорний ящик». У результаті аналізу були побудовані тестові випадки для шару представлення та бізнес логіки. Виконані тести допомогли вирішити ключові проблеми системи та був сформований звіт по результатам тестування.

Таким чином, результатом виконання дипломного проекту є проведений аналіз існуючих систем по темі «Програмне забезпечення для автоматизації обміну замовленнями між різними сервісами таксі». Визначені функціональні та нефункціональні вимоги, спроектовано архітектуру програмного комплексу. Реалізовано програмний продукт, що об’єднує служби таксі в єдину мережу для пришвидшення пошуку водія по замовленню в межах певного міста.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Агрегатор Матеріал з Вікісловника [Електронний ресурс] / Wikitionary – Режим доступу: https://uk.wiktionary.org/wiki/агрегатор (дата звернення: 15.04.2021).

2. Что такое биржа Up&Up и как она повысит доход службы такси? [Электронный ресурс] / Up&Up – Режим доступа: https://ligataxi.com/blog/chto-takoye-birzha-UpUp (дата обращения: 15.04.2021).

3. Биржа заказов такси SeDi [Электронный ресурс] / SeDi – Режим доступа: https://sedi.ru/биржа-такси/ (дата обращения: 15.04.2021).

4. Общая информация по работе с заказами из TMMarket [Электронный ресурс] / TMMMARKET – Режим доступа: https://www.taximaster.ru/instrukciya-tmmarket/ (дата обращения: 17.04.2021).

5. Биржа такси [Электронный ресурс] / Taxis – Режим доступа: https://taksi.bz/news/birzha-taksi (дата обращения: 17.04.2021).

6. Агрегаторы-такси: чем недовольны водители и как убивается рынок такси в Украине [Електронний ресурс] / TaxiService – Режим доступу до ресурсу: https://taxiservice.com.ua/ukraine/2522-agregator-taksi-chem-nedovolny-voditeli-i-kak-ubivaetsya-rynok-taksi-v-ukraine (дата обращения: 17.04.2021).

7. Уніфікована мова моделювання UML [Електронний ресурс] / Znannya – Режим доступу до ресурсу: http://www.znannya.org/?view=uml (дата звернення: 25.04.2021).

8. Chuck Groom. On Writing Tech Specs [Online] / C. Groom // CodeBurst – Available: https://codeburst.io/on-writing-tech-specs-6404c9791159. Accessed on: 25.04.2021.

9. Operating systems specs [Online] / Tech republic – Available: https://www.techrepublic.com/article/ubuntu-server-the-smart-persons-guide/. Accessed on: 25.04.2021.

10. Sql and NoSql database models [Электронный ресурс] / TProger – Режим доступа: https://tproger.ru/translations/sql-nosql-database-models/. Accessed

on: 25.04.2021.

11. PostgreSQL vs MySQL [Электронный ресурс] / Habr – Режим доступа: https://habr.com/ru/company/mailru/blog/248845/ (дата обращения: 29.04.2021).

12. Язык C# и .NET Framework [Электронный ресурс] / Professorweb – Режим доступа: https://professorweb.ru/my/csharp/charp_theory/level1/infonet.php (дата обращения: 29.04.2021).

13. Working with Spring Data Repositories [Online] / Spring documentation – Available: https://docs.spring.io/spring-data/datacommons/docs/1.6.1.RELEASE/reference/html/repositories.html. Accessed on: 29.04.2021.

14. The Spring @Controller and @RestController Annotations [Online] / Baeldung – Available: https://www.baeldung.com/spring-controller-vs-restcontroller. Accessed on: 29.04.2021.

15. Interface Stream [Online] / Oracl documentation – Available: https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html. Accessed on: 05.05.2021.

16. Многопоточный пакет util.concurrent [Електронний ресурс] / Java online – Режим доступу до ресурсу: http://java-online.ru/concurrent.xhtml #synchronizers (дата обращения: 05.05.2021).

17. Подходы (методы) тестирования [Электронный ресурс] / QAstart – Режим доступа: https://qastart.by/class/vidi-testirovaniya-m/47-podkhody-metody-testirovaniya (дата обращения: 07.05.2021).

ДОДАТОК А
(обов’язковий)

ТЕХНІЧНЕ ЗАВДАННЯ

Введення

Робота виконується в рамках проекту автоматизації обміну замовлень між різними сервісами таксі, що надає можливість передавати та отримувати замовлення у рамках певного міста.

1 Підстава для розробки

Підставою для розробки є «Завдання на дипломний проект», затверджене завідувачем кафедри інженерії програмного забезпечення.

Найменування розробки: Програмне забезпечення для автоматизації обміну замовленнями між різними сервісами таксі.

2 Призначення розробки

Програмне забезпечення для автоматизації обміну замовленнями між різними сервісами таксі призначене для поєднання різних служб таксі.

Користувачами програми є адміністратори власника системи та служби таксі, що зареєструвались.

У функціональних можливостях передбачено: додавання нового замовлення, взяття замовлення на виконання, відміна замовлення, сповіщення про виконання кожного етапу.

Редагування способів розподілу між сервісами таксі, реєстрація служб таксі, регулювання грошових потоків у зареєстрованих сервісів таксі. Деактивація розподілу для певних або всіх служб таксі.

Програмне забезпечення призначено для використання на спеціалізованому серверному обладнанні з достатньою потужністю. Обладнання не обмежується операційною системою, так як програмне забезпечення є кросплатформове.

3 Вимоги до програми

3.1 Вимоги до функціональних характеристик

Система повинна забезпечувати наступний функціонал:

- Оперативне розподілення замовлення та знаходження водія;
- Взяття замовлення на виконання;
- Можливість вказати, що водій прибув до клієнта;
- Можливість вказати, що водій почав поїздку з клієнтом;
- Можливість вказати, що водій завершив поїздку;
- Можливість відмінити замовлення;
- Реєстрація нових сервісів таксі з генерацією унікального ключа;
- Налаштування розподілення замовлень та додавання нових налаштувань;

3.2 Вимоги до надійності

Розроблюване ПЗ повинно мати:

– Кожне вхідне замовлення перевіряється на унікальність щоб уникнути дублювання в базі даних;

– При виконанні функцій перевіряється ключ доступу провайдера замовлення до API;

– Система має підтримувати протокол HTTPS;

3.3 Вимоги до складу та параметрів технічних засобів

Систему розраховано на високе навантаження але для цього потрібна висока продуктивність комп’ютера. Агрегатор можна запустити на простій системі при таких мінімальних параметрах:

– ОС Linux/Windows (на Windows продуктивність Агрегатора менша через те, що ресурси комп’ютера також витрачаються на підтримку операційної системи на відміну від Linux).

– Будь який двоядерний процесор з частотою на ядро 2.4 ГГц.

– 4 ГБ ОЗП;

– 20 МБ постійної пам’яті;

Рекомендовані вимоги:

– ОС Linux;

– Будь який чотириядерний процесор з частотою на ядро 3 ГГц;

– 4 ГБ ОЗП;

– 50 МБ постійної пам’яті;

3.4 Вимоги до інформаційної та програмної сумісності

Технічні рішення розроблюваної системи:

– Мова програмування: Java;

– Мова запитів до даних: Sql;

– Технології розробки: Spring Framework (включає в собі функціонал для доступу до даних, організації API та встановлення структури сутностей в коді);

– Сховище даних: PostgreSQL;

Структура програмного комплексу ділиться на дві частини:

– Сервер програмного комплексу (REST API), який поділяється на три під шари, що виконують окрему логіку:

а) API

б) Бізнес логіка

в) Доступ до даних

– База даних;

3.5 Спеціальні вимоги

Програма повинна виконувати операції максимально швидко. У випадку технічних неполадок система має мати можливість відновити хід роботи перед аварійним вимкненням.

4 Вимоги до програмної документації

В ході розробки програми повинні бути підготовлені:

– структурна схема додатку;

– текст програми – запис програми з необхідними поясненнями і коментарями;

– опис програми – відомості про логічну і фізичну модель, відомості про функціонування програми;

– програма і методика випробувань – вимоги, що підлягають перевірці при випробуванні програми, також порядок і методи контролю;

– технічне завдання – цей документ;

– записка пояснення – схема алгоритму, загальний опис алгоритму або функціонування програми, а також обґрунтування ухвалених технічних і техніко-економічних рішень.

5 Стадії та етапи розробки

Розробка проекту поділяється на декілька стадій. В свою чергу стадії розробки базуються на відомостях, що наведено у таблиці 2.4.1.

Таблиця 2.4.1 – стадії та етапи розробки системи

Стадії розробки	Етапи розробки
1. Технічне завдання (02.01.21 – 31.01.21)	Розробка технічного завдання
	Затвердження технічного завдання
2. Ескізний проєкт (01.02.21 – 14.02.21)	Розробка ескізного проєкту
	Затвердження ескізного проєкту
3. Технічний проєкт (15.02.21 – 28.02.21)	Розробка технічного проєкту
	Затвердження технічного проєкту
4. Робочий проєкт (01.03.21 – 25.04.21)	Розробка першої релізної версії системи
	Розробка програмної документації
	Тестування системи на тестовому комп’ютері
5. Впровадження (26.04.21 – 31.05.21)	Установка сервера або оренда
	Тестування на реальному сервері та правки
	Запуск системи

6 Порядок контролю та приймання

Контроль здійснюється ініціатором після завершення кожної із проміжних стадій розробки та може тривати не довше п’яти днів. Тестування на завершальному етапі буде здійснюватися протягом чотирнадцяти робочих днів не лише замовником, але й кінцевими користувачами системи. У разі виявлення помилок чи зауважень проєкт відправляється на доопрацювання, яке може тривати до семи робочих днів.

ДОДАТОК Б
(довідковий)

МОДЕЛІ ДАНИХ

Б.1 Модель замовлення:

```
Integer id;
String deliveryDt;
Integer statusId;
String taxiName;
Boolean isBonusOrder;
Boolean isAdvanceOrder;
Integer paymentTypeId;
List<OrderAddress> addresses;
String entrance;
String apartment;
String comments;
String commentsForDriver;
Double surcharge;
Double discount;
Double totalSurcharge;
Double totalCoefficient;
Double distance;
Integer tripTime;
Double price;
Double commission;
Integer driverId;
String phone;
String clientNum;
String clientName;
Double distanceToOrder;
FareDto fare;
Integer waitTime;
Boolean waitFromAtAddressPressed;
Boolean addWaitTimeToTaximeter;
Boolean roundTaximeterDuringTrip;
Boolean roundTaximeterAfterTrip;
Double distanceCoefficient;
Boolean showTariffInTaximeter;
Boolean showTaxiNameInTaximeter;
Double rollbackAmount;
Boolean allowRecalculatePriceAuto;
Boolean allowRecalculatePriceManual;
Double objectsSurcharge;
Double intermediateFirstPointSurcharge;
Double intermediateOtherPointsSurcharge;
OffsetDateTime endOfSearchTime;
String driverPhone;
```

```
Integer driverCall;
String taxiToken;
String taxiTokenFrom;
String apiKey;
```

Б.2 Модель з інформацією про водія:

```
String apiKey;
String taxiToken;
Integer aggregatorId;
String taxiTokenFrom;
Integer time;
String checker;
String driverCarNumTaken;
String driverNameTaken;
String driverPhoneTaken;
String driverCarModelTaken;
Integer driverCallTaken;
Double driverRankTaken;
Integer driverRatingCountTaken;
String driverPhotoTaken;
Integer driverCarIdTaken;
Integer driverColorIdTaken;
Double aggregatorFixedCommission;
Double aggregatorPercentCommission;
```

Б.3 Модель інформації про завершення поїздки:

```
String apiKey;
String taxiToken;
String taxiTokenFrom;
Integer aggregatorId;
Double distanceIn;
Double distanceOut;
Integer stayTime;
Integer tripTime;
Double price;
Double calculated;
```

ДОДАТОК В
(обов'язковий)

ПРОГРАМНИЙ КОД ОСНОВНИХ МОДУЛІВ

В.1 Програмний код класу Controller

```
@RestController
public class Controller {
    private static final Logger logger = LogManager.getLogger(Controller.class);
    private final OrderService orderService;
    private final OrderMapper orderMapper;
    public Controller(OrderService orderService, OrderMapper orderMapper) {
        this.orderService = orderService;
        this.orderMapper = orderMapper;
    }
    @PostMapping(
        path = "sendOrder",
        produces = MediaType.APPLICATION_JSON_VALUE,
        consumes = MediaType.APPLICATION_JSON_VALUE
    )
    public RestApiResponse getNewOrder(@Valid @RequestBody CurrentOrderDto order)
{
        logger.info("Request to api/sendOrder with info - " + order.toString());
        return orderService.takeOrderForSearchingDriver(order);
    }
    @PostMapping(
        path = "orderTake",
        produces = MediaType.APPLICATION_JSON_VALUE,
        consumes = MediaType.APPLICATION_JSON_VALUE
    )
    public RestApiResponse takeOrder(@RequestBody @Valid TakeOrderDto order) {
        logger.info("Request to api/orderTake with info - " + order.toString());
        return orderService.takeTakenOrder(order);
    }
    @PostMapping(
        path = "orderAtAddress",
        produces = MediaType.APPLICATION_JSON_VALUE,
        consumes = MediaType.APPLICATION_JSON_VALUE
    )
    public RestApiResponse orderAtAddress(@RequestBody @Valid AtAddressDto order)
{
        logger.info("Request to api/orderAtAddress with info - " +
order.toString());
        return orderService.orderAtAddress(order);
    }
    @PostMapping(
        path = "startTrip",
        produces = MediaType.APPLICATION_JSON_VALUE,
        consumes = MediaType.APPLICATION_JSON_VALUE
    )
    public RestApiResponse sendOrderStartTrip(@RequestBody @Valid StartTripDto
order) {
        logger.info("Request to api/startTrip with info - " + order.toString());
        return orderService.orderStartTrip(order);
    }
    @PostMapping(
```

```
        path = "finishTrip",
        produces = MediaType.APPLICATION_JSON_VALUE,
        consumes = MediaType.APPLICATION_JSON_VALUE
    )
    public RestApiResponse sendOrderFinishTrip(@RequestBody @Valid FinishTripDto
order) {
        logger.info("Request to api/finishTrip with info - " + order.toString());
        return orderService.orderFinishTrip(order);
    }
    @PostMapping(
        path = "cancelOrder",
        produces = MediaType.APPLICATION_JSON_VALUE,
        consumes = MediaType.APPLICATION_JSON_VALUE
    )
    public RestApiResponse sendOrderCancelled(@RequestBody @Valid CancelOrderDto
order) {
        logger.info("Request to api/cancelOrder with info - " + order.toString());
        return orderService.orderCancel(order);
    }
    @PostMapping(
        path = "askConference",
        produces = MediaType.APPLICATION_JSON_VALUE,
        consumes = MediaType.APPLICATION_JSON_VALUE
    )
    public RestApiResponse askConference(@RequestBody @Valid AskConferenceDto
order) {
        logger.info("Request to api/askConference with info - " +
order.toString());
        return orderService.askConference(order);
    }
}
```

В.2 Програмний код класу OrderService

```
@Service
public class OrderService {
    private static final Logger log = LogManager.getLogger(OrderService.class);
    private final WebClientService webClientService;
    private final FireBirdService evosFireBirdService;
    private final TaxiServiceRepository taxiServiceRepository;
    private final OrderMapper orderMapper;
    private final TransfersRepository transfersRepository;
    private final ReceiversRepository receiversRepository;
    private final CurrentOrderRepository currentOrderRepository;
    public OrderService(WebClientService webClientService,
                        FireBirdService fireBirdDAO,
                        TaxiServiceRepository taxiServiceRepository,
                        OrderMapper orderMapper,
                        TransfersRepository transfersRepository,
                        ReceiversRepository receiversRepository,
                        CurrentOrderRepository currentOrderRepository) {
        this.webClientService = webClientService;
        this.evosFireBirdService = fireBirdDAO;
        this.taxiServiceRepository = taxiServiceRepository;
        this.orderMapper = orderMapper;
        this.transfersRepository = transfersRepository;
        this.receiversRepository = receiversRepository;
        this.currentOrderRepository = currentOrderRepository;
    }
    public RestApiResponse askConference(AskConferenceDto orderDto) {
        CurrentOrder currentOrder = currentOrderRepository
```

```
            .findByOrderProviderIdAndTaxiTokenFrom(orderDto.getAggregatorId(),
orderDto.getTaxiTokenFrom());
        TaxiService taxiService = currentOrder.getTaxiService();

        orderDto.setApiKey(null);

        log.info("Ask conference by orderId: " + currentOrder.getId());
        Boolean confirmation = webClientService.sendRequestForAskConference(
            orderDto, taxiService.getUrl(), taxiService.getPort(),
"aggregator/askConference"
        );
        return new RestApiResponse(confirmation);
    }
    @Transactional
    public RestApiResponse takeTakenOrder(TakeOrderDto orderDto) {
        RestApiResponse orderResponse;
        try {
            String endPoint = "aggregator/takeOrder";
            CurrentOrder order =
currentOrderRepository.findById(orderDto.getAggregatorId()).orElseThrow();
            if (order != null && order.getStatusId() != 3 && order.getStatusId()
!= 4 && order.getStatusId() != 5 && order.getStatusId() != 6) {
                TaxiService taxiWhoTakeOrder =
taxiServiceRepository.findByTokenAndApiKey(orderDto.getTaxiToken(),
orderDto.getApiKey());
                TaxiService orderProvider = order.getTaxiService();
                Transfers transfer =
transfersRepository.findByTaxiServiceIdAndTaxiTransferToId(orderProvider.getId(),
taxiWhoTakeOrder.getId());
orderDto.setAggregatorFixedCommission(transfer.getCommissionFixed());
orderDto.setAggregatorPercentCommission(transfer.getCommissionPercent());
                TakeOrderDto requestInfo = orderDto.copyForRequest();
                order = orderMapper.modifyTakeOrder(order, orderDto);
                order.setTaxiServiceTaken(taxiWhoTakeOrder);
                TaxiServiceDto responseFromOrderProvider =
webClientService.sendOrderStatus(
                    requestInfo,
                    orderProvider.getUrl(),
                    orderProvider.getPort(),
                    endPoint
                );
                if (responseFromOrderProvider == null) {
                    log.error(
                        "Taxi service " + orderProvider.getUrl() + ":" +
orderProvider.getPort() + endPoint +
                            " not responding"
                    );
                    orderResponse = setStatusForOrderResponse(order.getStatusId(),
order.getId());
                    orderResponse.setConfirmed(false);
                    return orderResponse;
                }
                if (responseFromOrderProvider.isConfirmed()) {
                    log.info(
                        "Taxi service " + orderProvider.getUrl() + ":" +
orderProvider.getPort() + endPoint +
                            " responded"
                    );
                    orderResponse = setStatusForOrderResponse(3, order.getId());
                    orderResponse.setConfirmed(true);
                    orderDto.setOrderId(order.getId());
                    log.info(
```

```
                        "Update order " + order.getId() + " and set status 3 -
taken"
                    );
                    order.setOrderConsumerId(orderDto.getOrderId());
                    order.setStatusId(3);
                    currentOrderRepository.saveAndFlush(order);
                    return orderResponse;
                }
                log.error(
                    "Taxi service " + orderProvider.getUrl() + ":" +
orderProvider.getPort() + endPoint +
                        " responded but did not confirm the action"
                );
                orderResponse = setStatusForOrderResponse(order.getStatusId(),
order.getId());
            } else {
                log.error(
                    "Order not saved for " + orderDto.getTaxiToken() + " - " +
orderDto +
                        ", because of order does not exist or is taken"
                );
                orderResponse = new RestApiResponse();
            }
            orderResponse.setConfirmed(false);
            return orderResponse;
        } catch (Exception e) {
            log.error(
                "Order not saved for " + orderDto.getTaxiToken() + " - " +
orderDto +
                    ", because of order does not exist or is taken \n EXCEPTION:
\n" + e.getMessage()
            );
            orderResponse = new RestApiResponse();
            orderResponse.setConfirmed(false);
            return orderResponse;
        }
    }
    public RestApiResponse orderAtAddress(AtAddressDto orderDto) {
        String endPoint = "aggregator/orderAtAddress";
        CurrentOrder order =
currentOrderRepository.findById(orderDto.getAggregatorId()).orElseThrow();
        TaxiService orderProvider = order.getTaxiService();
        orderDto.clearForRequest();
        order = orderMapper.modifyAtAddressOrder(order, orderDto);
        log.info(
            "Send request to order: " + order.getId() +
                ", provider" + orderProvider.getUrl() + ":" +
orderProvider.getPort() + endPoint +
                " for change order status to 4 - at_address"
        );
        TaxiServiceDto responseFromOrderProvider =
webClientService.sendOrderStatus(
            orderDto, orderProvider.getUrl(), orderProvider.getPort(), endPoint
        );
        RestApiResponse orderResponse;
        if (responseFromOrderProvider == null) {
            log.error(
                "Taxi service" + orderProvider.getUrl() + ":" +
orderProvider.getPort() + endPoint +
                    " not responding"
            );
            orderResponse = setStatusForOrderResponse(order.getStatusId(),
order.getId());
```

```
            return orderResponse;
        }
        if (responseFromOrderProvider.isConfirmed()) {
            log.info(
                "Taxi service" + orderProvider.getUrl() + ":" +
orderProvider.getPort() + endPoint +
                    " responded"
            );
            orderResponse = setStatusForOrderResponse(4, order.getId());
            log.info(
                "Update order " + order.getId() + " and set status 4 - at_address"
            );
            currentOrderRepository.save(order);
            return orderResponse;
        }
        log.error(
            "Taxi service" + orderProvider.getUrl() + ":" +
orderProvider.getPort() + endPoint +
                " responded but did not confirm the action"
        );
        orderResponse = setStatusForOrderResponse(order.getStatusId(),
order.getId());
        return orderResponse;
    }
    public RestApiResponse orderStartTrip(StartTripDto orderDto) {
        String endPoint = "aggregator/orderStartTrip";
        CurrentOrder order =
currentOrderRepository.findById(orderDto.getAggregatorId()).orElseThrow();
        TaxiService orderProvider = order.getTaxiService();
        RestApiResponse orderResponse;
        StartTripDto requestToOrderProvider = new StartTripDto();
        requestToOrderProvider.setTaxiTokenFrom(orderDto.getTaxiTokenFrom());
        requestToOrderProvider.setAggregatorId(order.getId());
        order = orderMapper.modifyStartTripOrder(order, orderDto);
        log.info(
            "Send request to order: " + order.getId() +
                ", provider" + orderProvider.getUrl() + ":" +
orderProvider.getPort() + endPoint +
                " for change order status to 5 - start_trip"
        );
        TaxiServiceDto taxiServiceDto = webClientService.sendOrderStatus(
            requestToOrderProvider, orderProvider.getUrl(),
orderProvider.getPort(), endPoint
        );
        if (taxiServiceDto == null) {
            log.error(
                "Taxi service" + orderProvider.getUrl() + ":" +
orderProvider.getPort() + endPoint +
                    " not responding"
            );
            orderResponse = setStatusForOrderResponse(order.getStatusId(),
order.getId());
            return orderResponse;
        }
        if (taxiServiceDto.isConfirmed()) {
            log.info(
                "Taxi service" + orderProvider.getUrl() + ":" +
orderProvider.getPort() + endPoint +
                    " responded"
            );
            orderResponse = setStatusForOrderResponse(5, order.getId());
            log.info(
                "Update order " + order.getId() + " and set status 5 - start_trip"
```

```
            );
            currentOrderRepository.save(order);
            return orderResponse;
        }
        log.error(
            "Taxi service" + orderProvider.getUrl() + ":" +
orderProvider.getPort() + endPoint +
                " responded but did not confirm the action"
        );
        orderResponse = setStatusForOrderResponse(order.getStatusId(),
order.getId());
        return orderResponse;
    }
    public RestApiResponse orderFinishTrip(FinishTripDto orderDto) {
        String endPoint = "aggregator/orderFinishTrip";
        CurrentOrder order =
currentOrderRepository.findById(orderDto.getAggregatorId()).orElseThrow();
        TaxiService taxiWhoTakeOrder =
taxiServiceRepository.findByTokenAndApiKey(orderDto.getTaxiToken(),
orderDto.getApiKey());
        TaxiService orderProvider = order.getTaxiService();
        Transfers transfer =
transfersRepository.findByTaxiServiceIdAndTaxiTransferToId(orderProvider.getId(),
taxiWhoTakeOrder.getId());
        Double percent = transfer.getCommissionPercent();
        Double fixed = transfer.getCommissionFixed();
        if (percent != null) {
            orderProvider.setBalance(orderProvider.getBalance() -
(orderDto.getCalculated() * (percent / 100)));
        }
        if (fixed != null) {
            orderProvider.setBalance(orderProvider.getBalance() - fixed);
        }
        RestApiResponse orderResponse;
        FinishTripDto finishTripDto = orderDto.copyForRequest();
        order = orderMapper.modifyFinishTripOrder(order, orderDto);
        log.info(
            "Send request to order: " + order.getId() +
                ", provider " + orderProvider.getUrl() + ":" +
orderProvider.getPort() + endPoint +
                " for change order status to 6 - finish_trip.  with info - " +
finishTripDto.toString()
        );
        TaxiServiceDto taxiServiceDto = webClientService.sendOrderStatus(
            finishTripDto, orderProvider.getUrl(), orderProvider.getPort(),
endPoint
        );
        if (taxiServiceDto == null) {
            log.error(
                "Taxi service" + orderProvider.getUrl() + ":" +
orderProvider.getPort() + endPoint +
                    " not responding"
            );
            orderResponse = setStatusForOrderResponse(order.getStatusId(),
order.getId());
            return orderResponse;
        }
        if (taxiServiceDto.isConfirmed()) {
            log.info(
                "Taxi service" + orderProvider.getUrl() + ":" +
orderProvider.getPort() + endPoint +
                    " responded"
            );
```

```
            orderResponse = setStatusForOrderResponse(6, order.getId());
            log.info(
                "Update order " + order.getId() + " and set status 6 -
finish_trip"
            );
            currentOrderRepository.save(order);
            return orderResponse;
        }
        log.error(
            "Taxi service" + orderProvider.getUrl() + ":" +
orderProvider.getPort() + endPoint +
                " responded but did not confirm the action"
        );
        orderResponse = setStatusForOrderResponse(order.getStatusId(),
order.getId());
        return orderResponse;
    }
    public RestApiResponse orderCancel(CancelOrderDto orderDto) {
        CurrentOrder order =
currentOrderRepository.findById(orderDto.getAggregatorId()).orElseThrow();
        order.setStatusId(12);
        order.setCanceledDt(OffsetDateTime.now());
        order.setCancelReason(orderDto.getCancelReason());
        currentOrderRepository.save(order);
        if (order.getTaxiServiceTaken().getId() != 0) {
            TaxiService orderProvider = order.getTaxiService();
            List<Transfers> transfers =
transfersRepository.findByTaxiServiceIdAndEnabledIsTrue(orderProvider.getId());
            List<Integer> receivers =
receiversRepository.findByTaxiReceiveFromIdAndEnabledIsTrue(orderProvider.getId())
                .stream()
                .map(receiver -> receiver.getTaxiService().getId())
                .collect(Collectors.toList());
            List<Transfers> transferTo = transfers.stream()
                .filter(transfer -> receivers.contains(transfer.getId()))
                .collect(Collectors.toList());
            final List<Callable<TaxiServiceDto>>
tasksForSendRequestToOtherServices = new ArrayList<>();
            final ExecutorService executor = Executors.newWorkStealingPool();
            List<TaxiServiceDto> responsesFromServices;
            CancelOrderDto cancelOrderDto = orderDto.copyForRequest();
            transferTo.forEach(transfer -> tasksForSendRequestToOtherServices.add(
                () -> {
                    TaxiService taxi = transfer.getTaxiTransferTo();
                    if (!taxi.getToken().equals(orderProvider.getToken())) {
                        log.info("Sending request to " + taxi.getUrl() + ":" +
                            taxi.getPort() + "aggregator/cancelOrder - with info -
" +
                            orderDto);
                        return webClientService.sendOrderStatus(
                            cancelOrderDto,
                            taxi.getUrl(),
                            taxi.getPort(),
                            "aggregator/cancelOrder"
                        );
                    } else {
                        TaxiServiceDto provider = new TaxiServiceDto();
                        provider.setConfirmed(false);
                        return provider;
                    }
                })
            );
            try {
```

```
                executor.invokeAll(tasksForSendRequestToOtherServices)
                    .forEach(pairFuture -> {
                        try {
                            if (pairFuture.get() != null) {
                                log.info(
                                    "Receive response from taxi: " +
pairFuture.get().getTaxiToken() +
                                        " by order cancel. "
                                );
                            }
                        } catch (InterruptedException | ExecutionException e) {
                            log.error(e.getMessage());
                        }
                    });
                //responsesFromServices.forEach(log::info);
            } catch (InterruptedException e) {
                log.error(e.getMessage());
            }
        }
        return setStatusForOrderResponse(order.getStatusId(), order.getId());
    }
    public RestApiResponse updateCoordinates(UpdateCoordinatesDto orderDto) {
        String endPoint = "aggregator/updateCoordinates";
        CurrentOrder order =
currentOrderRepository.findById(orderDto.getAggregatorId()).orElseThrow();
        TaxiService orderProvider = order.getTaxiService();
        RestApiResponse orderResponse;
        UpdateCoordinatesDto updateCoordinatesDto = new UpdateCoordinatesDto();
        updateCoordinatesDto.setTaxiTokenFrom(orderDto.getTaxiTokenFrom());
        updateCoordinatesDto.setAggregatorId(order.getId());
        updateCoordinatesDto.setBearing(orderDto.getBearing());
        updateCoordinatesDto.setDriverLat(orderDto.getDriverLat());
        updateCoordinatesDto.setDriverLon(orderDto.getDriverLon());
        order.setDriverLat(orderDto.getDriverLat());
        order.setDriverLon(orderDto.getDriverLon());
        order.setDriverBearing(orderDto.getBearing());
        log.info(
            "Send request to order: " + order.getId() +
                ", provider" + orderProvider.getUrl() + ":" +
orderProvider.getPort() + endPoint +
                " for change order status to 5 - start_trip"
        );
        TaxiServiceDto taxiServiceDto = webClientService.sendOrderStatus(
            updateCoordinatesDto, orderProvider.getUrl(), orderProvider.getPort(),
endPoint
        );
        if (taxiServiceDto == null) {
            log.error(
                "Taxi service" + orderProvider.getUrl() + ":" +
orderProvider.getPort() + endPoint +
                    " not responding"
            );
            orderResponse = setStatusForOrderResponse(order.getStatusId(),
order.getId());
            return orderResponse;
        }
        if (taxiServiceDto.isConfirmed()) {
            log.info(
                "Taxi service" + orderProvider.getUrl() + ":" +
orderProvider.getPort() + endPoint +
                    " responded"
            );
            orderResponse = setStatusForOrderResponse(5, order.getId());
```

```
            log.info(
                "Update order " + order.getId() + " and set status 5 - start_trip"
            );
            currentOrderRepository.save(order);
            return orderResponse;
        }
        log.error(
            "Taxi service" + orderProvider.getUrl() + ":" +
orderProvider.getPort() + endPoint +
                " responded but did not confirm the action"
        );
        orderResponse = setStatusForOrderResponse(order.getStatusId(),
order.getId());
        return orderResponse;
    }
    private RestApiResponse setStatusForOrderResponse(Integer statusId, Integer
orderId) {
        return new RestApiResponse(statusId, orderId);
    }
    public RestApiResponse takeOrderForSearchingDriver(CurrentOrderDto orderDto) {
        final TaxiService taxiService =
taxiServiceRepository.findByTokenAndApiKey(orderDto.getTaxiToken(),
orderDto.getApiKey());
        CurrentOrder orderForSave = orderMapper.toEntity(orderDto);
        orderForSave.setTaxiService(taxiService);
        orderForSave.setOrderProviderId(orderDto.getId());
        CurrentOrder order = currentOrderRepository.saveAndFlush(orderForSave);
        List<Transfers> transfers =
transfersRepository.findByTaxiServiceIdAndEnabledIsTrue(taxiService.getId());
        List<Integer> receivers =
receiversRepository.findByTaxiReceiveFromIdAndEnabledIsTrue(taxiService.getId())
            .stream()
            .map(receiver -> receiver.getTaxiService().getId())
            .collect(Collectors.toList());
        List<Transfers> transferTo = transfers.stream()
            .filter(transfer ->
receivers.contains(transfer.getTaxiTransferTo().getId()))
            .collect(Collectors.toList());
        final ExecutorService executor = Executors.newWorkStealingPool();
        final List<Callable<TaxiServiceDto>> tasksForSendRequestToOtherServices =
new ArrayList<>();
        final ScheduledExecutorService scheduledExecutorServiceToTaxi =
Executors.newScheduledThreadPool(transferTo.size());
        List<TaxiServiceDto> responsesFromServices = null;
        CurrentOrderDto orderTransfer = orderMapper.toDto(order);
        log.debug("REQUESTING order - " + orderTransfer);
        transferTo.forEach(transfer -> tasksForSendRequestToOtherServices.add(
            () -> {
                TaxiService taxi = transfer.getTaxiTransferTo();
                if (taxi.getProtocolId() == 0) {
                    TaxiServiceDto taxiServiceDto;
                    taxiServiceDto = scheduledExecutorServiceToTaxi.schedule(() ->
{
                        if
(!taxi.getToken().equals(orderTransfer.getTaxiTokenFrom())) {
                            log.info("Sending request to " + taxi.getUrl() + ":" +
                                taxi.getPort() + "/aggregator/searchDriver - with
info - " +
                                orderTransfer);
                            orderTransfer.setTaxiToken(taxi.getToken());
                            return webClientService.sendOrderToTaxiService(
                                orderTransfer,
                                taxi.getUrl(),
```

```
                                taxi.getPort(),
                                "aggregator/searchDriver"
                            );
                        } else {
                            TaxiServiceDto provider = new TaxiServiceDto();
                            provider.setConfirmed(false);
                            return provider;
                        }
                    }, transfer.getDelay(), TimeUnit.SECONDS).get();
                    TimeUnit.SECONDS.sleep(transfer.getDelay());
                    return taxiServiceDto;
                }
                if (taxi.getProtocolId() == 1) {
                    TaxiServiceDto taxiServiceDto = new TaxiServiceDto();
                    int delay = 20;
                    log.info("Sending request to EVOS FireBird - with info - " +
                        orderDto);
                    evosFireBirdService.setOrderViaRemoteService(order, taxi,
delay);
                    taxiServiceDto.setTaxiToken("EVOS");
                    taxiServiceDto.setConfirmed(true);
                    return taxiServiceDto;
                }
                TaxiServiceDto provider = new TaxiServiceDto();
                provider.setConfirmed(false);
                return provider;
            })
        );
        try {
            responsesFromServices =
executor.invokeAll(tasksForSendRequestToOtherServices)
                .stream()
                .map(pairFuture -> {
                    try {
                        log.info(
                            "Receive response from taxi: " +
pairFuture.get().getTaxiToken() +
                                " - with info - " + pairFuture.get().isConfirmed()
                        );
                        return pairFuture.get();
                    } catch (InterruptedException e) {
                        log.error(e.getMessage());
                    } catch (ExecutionException e) {
                        log.error("Connection refused - " + e.getMessage());
                    }
                    return new TaxiServiceDto();
                })
                .collect(Collectors.toList());
        } catch (InterruptedException e) {
            log.error(e.getMessage());
        }
        RestApiResponse restApiResponse = new RestApiResponse();
        if (responsesFromServices != null && !responsesFromServices.isEmpty()) {
            log.debug("Responded services " + responsesFromServices);
            List<TaxiServiceDto> checkForResponse = responsesFromServices
                .stream()
                .filter(Objects::nonNull)
                .filter(taxiServiceDto -> taxiServiceDto.isConfirmed() != null &&
taxiServiceDto.isConfirmed())
                .collect(Collectors.toList());
            boolean driverSearching = !checkForResponse.isEmpty();
            restApiResponse.setConfirmed(driverSearching);
            if (!driverSearching) currentOrderRepository.delete(order);
```

```
            else deleteOrderByTimeout(orderDto, order);
        } else {
            log.debug("Not exists services for order - " + orderTransfer);
            deleteOrderByTimeout(orderDto, order);
        }
        return restApiResponse;
    }
    private void deleteOrderByTimeout(CurrentOrderDto orderDto, CurrentOrder
order) {
        new Thread(() -> {
            while (!OffsetDateTime.now().isAfter(orderDto.getEndOfSearchTime())) {
                try {
                    TimeUnit.SECONDS.sleep(1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            if (OffsetDateTime.now().isAfter(orderDto.getEndOfSearchTime())) {
                CurrentOrder orderForDelete =
currentOrderRepository.findById(order.getId()).orElseThrow();
                if (orderForDelete.getStatusId().equals(1))
currentOrderRepository.deleteById(orderForDelete.getId());
            }
        }).start();
    }

}
```

ДОДАТОК Г
(обов’язковий)

ПРЕЗЕНТАЦІЙНІ МАТЕРІАЛИ

1

Хмельницький Національний Університет
Факультет програмування та комп'ютерних
і телекомунікаційних систем
Кафедра інженерії програмного забезпечення

Дипломний проект, на тему:

«Програмне забезпечення для автоматизації обміну замовленнями між різними сервісами таксі»

Студент: Мисик Іван Сергійович
Керівник: Бедратюк Л. П. доктор фізико-математичних наук, професор

2

Вступ

Сьогодні сфера обслуговування людей у галузі таксі має велику популярність, яка росте за рахунок якісного обслуговування та легкого доступу. Адже на сучасному етапі розвитку людства існує різноманітне програмне забезпечення, що спрощує багато процесів. Загалом таксі сервіс, в технічному плані, складається з серверу, мобільного додатку водія, мобільного додатку клієнта, панель адміністратора та робочої частини оператора. Зараз, щоб увійти у ринок таксі послуг, потрібен саме такий набір.

Але яке б нове, інноваційне, оригінальне програмне забезпечення не було у сервісу таксі завжди присутній людський вплив. Сучасний таксі сервіс як і десять років тому не може існувати без людського контролю. Завжди є присутній технічний адміністратор або оператор який контролює хід роботи ну і звісно ж водії.

Актуальність та мета дипломного проекту

3

- Актуальність теми полягає в тому, що кожен сервіс, коли виходить на ринок з часом стикається з різними факторами, що погано впливають на популярність компанії. Одним з таких факторів є нехватка кадрів. Водії постійно у пошуках комфортніших умов, через що міняють компанію на більш зручних правилах. Дане програмне забезпечення дозволить об'єднати служби таксі в єдину мережу, за рахунок чого не буде важливо в якій службі таксі знаходиться водій.
- Метою проекту є створення сервісу для обміну таксі замовлень, який дозволятиме різним таксі компаніям, абсолютно автоматично, підтримувати зв'язок для підвищення швидкості знаходження машини для клієнта в межах певного міста.

Завдання дипломного проекту

4

Для вирішення поставленої проблеми потрібно виконати наступні задачі:

- Провести аналіз сучасних систем обміну замовленнями.
- Визначити модель роботи обміну замовленнями та сервісу таксі.
- Розробити модель сервісу для обміну замовленнями на основі таксі сервісів.
- Розробити архітектуру сервісу для обміну замовленнями.
- Розробити алгоритм розподілення замовлення між сервісами.
- Виконати програмну реалізацію сервісу для обміну замовленнями.

Наявне програмне забезпечення 5

- Up&Up

Особливості обмінника:

Обмінник можливо підключити в любому місті Росії але мінімальні вимоги – два робочих сервіси таксі, які хочуть обмінюватись замовленнями. Також є можливим запустити обмінник в іншій країні але обов'язковими умовами є десять таксі сервісів які готові обмінюватись замовленнями. Для роботи біржі потрібно провести інтеграцію з диспетчерським робочим місцем.

Недоліки обмінника:
Купити замовлення можуть тільки водії в додатку LigaTaxi. Після купівлі, замовлення відразу ж створюється на робочому місці у оператора.

Наявне програмне забезпечення 6

- SeDi

Особливості біржі:

Автоматична біржа замовлень таксі пропонує програмний комплекс, який включає в себе водійський додаток та диспетчерську робочу частину для керуванням обміном замовлень.

SeDi
Service Direct

Біржа створена в тому числі і для партнерів, які вже працюють на будь-якому ПО і/або поки не бажають переходити на роботу в системі SeDi. У компаній, що використовують програму SeDiManager, біржовий механізм використовується більш ефективно, і всі операції максимально автоматизовані.

Висновки аналізу наявного програмного забезпечення

7

Більшість існуючих бірж направлені на те, щоб заробити швидше та більше, що погано впливає на швидкість обробки замовлень. Також через те, що стати таксистом стає все простіше завдяки таким агрегторам погіршує статистику якості обслуговування.

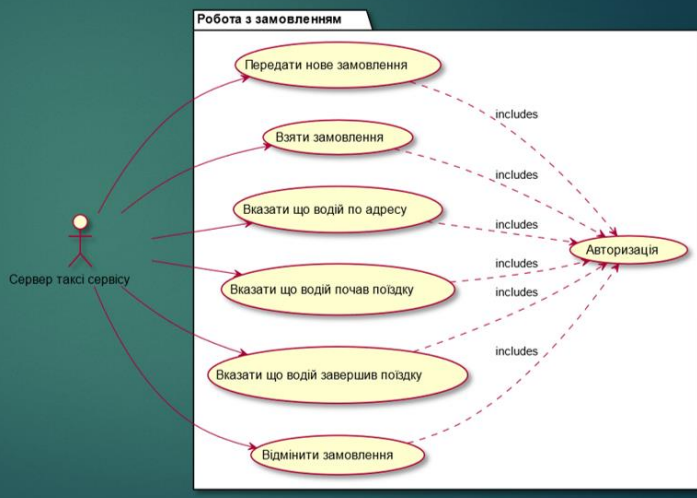
Розуміючи нинішню ситуацію відкритих аггрегаторів таксі можемо вивести такий перечень характеристик майбутньої програмної системи:

- Відкритий для всіх існуючих служб таксі, що мають своє власне програмне забезпечення;
- Абсолютна автоматизація всіх етапів виконання замовлення;
- Замовлення, в першу чергу, не продається а розподіляється між всіма користувачами системи;
- Кожен етап виконання замовлення фіксується в електронному журналі;
- Захист доступу до системи від сторонніх служб;
- Налаштування всіх процесів розподілення замовлення;

Проектування

8

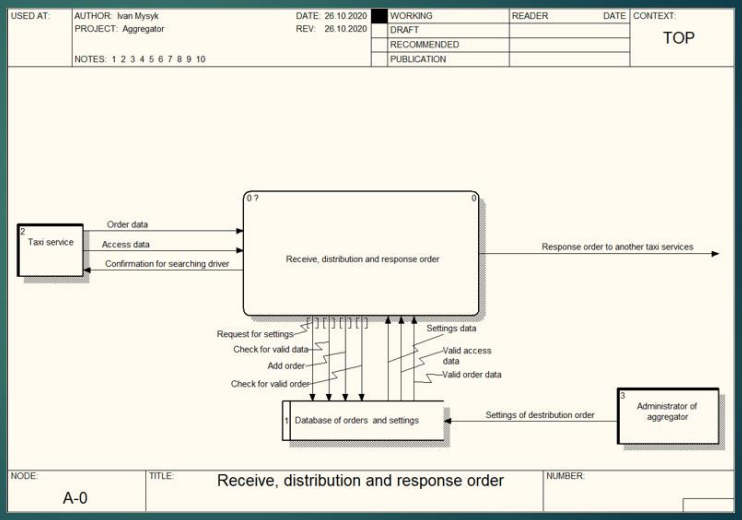
У результаті проектування була побудована така діаграма використання:



Проектування

9

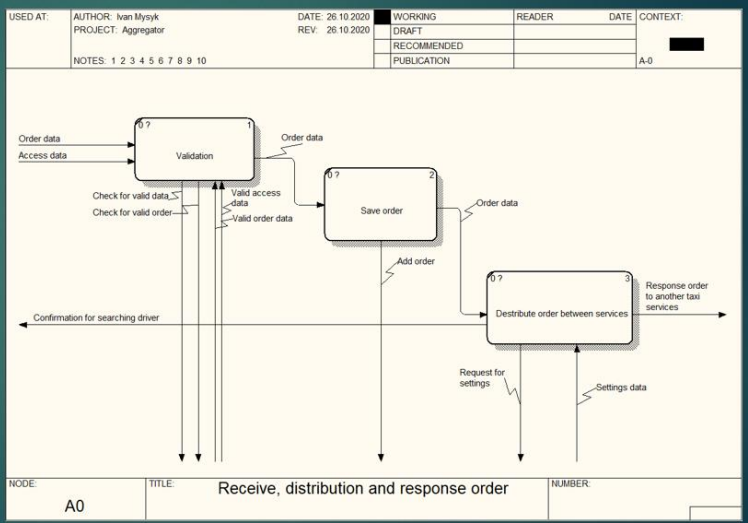
Для опису обробки замовлення побудовано діаграму потоків даних першого рівня



Проектування

10

Діаграма потоків даних другого рівня



Принцип роботи реалізації

11

Висновок

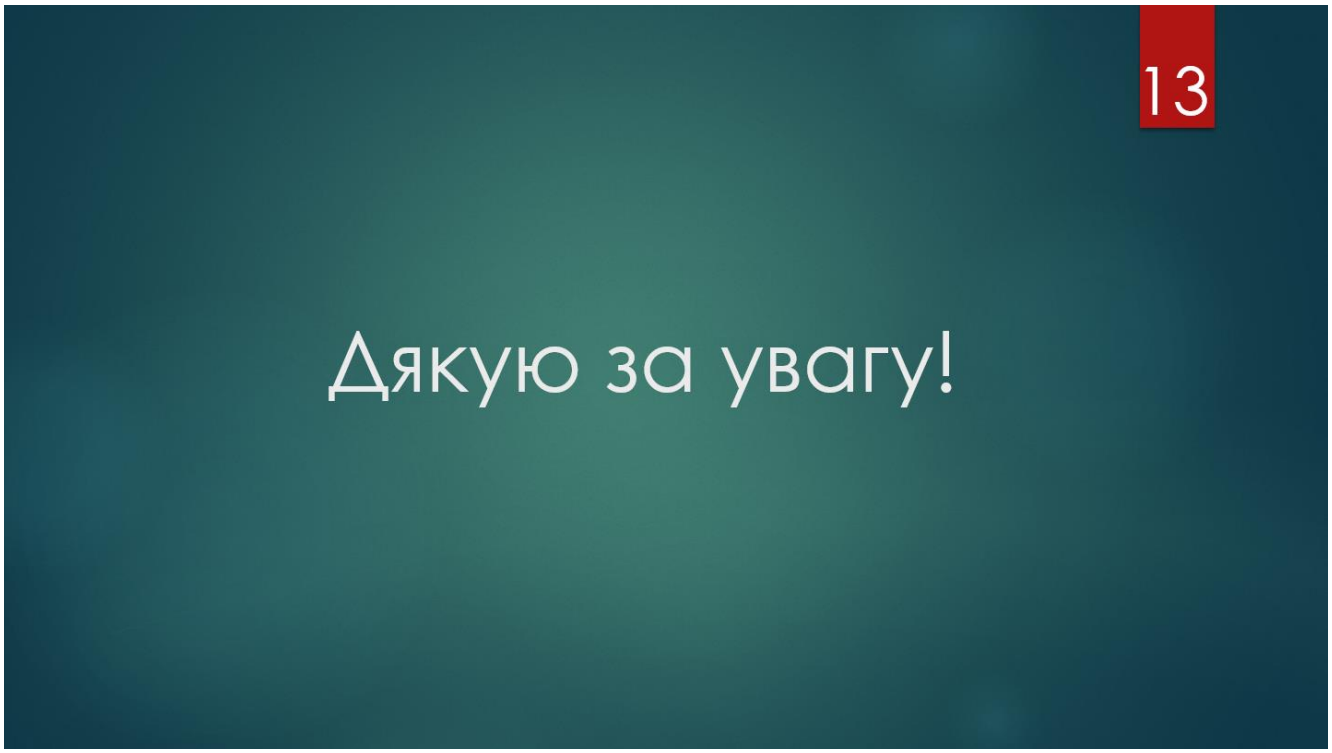
12

В процесі виконання дипломного проекту за темою «програмне забезпечення для автоматизації обміну замовленнями між різними сервісами таксі» було проаналізовано предметну область та усі функціональні та не функціональні вимоги.

Виконано аналіз наявного програмного забезпечення та визначені усі недоліки та переваги. Зібрано загальну інформацію про необхідний набір функціоналу. Встановлено, що відкриті сервіси автоматичного розподілення замовлень таксі є досить тонко направленою сферою. Структура таких застосунків залежить від існуючих сервісів таксі та інших факторів.

Після проведеного проектування був обраний клієнт-серверний тип архітектури з поправкою на те, що клієнтами виступають сервери інших сервісів таксі. Визначено монолітний шаблон проектування. Обрана база даних з цілю швидкого сортування та фільтрації даних.

В результаті роботи було реалізоване програмне забезпечення, що може об'єднати різні сервіси таксі в одну мережу з цілю підвищення ефективності виконання замовлень та покращення іміджу сервісів таксі.


13
Дякую за увагу!

Завідувачу кафедри інженерії програмного забезпечення проф. Бедратюку Л. Н.

здобувача вищої освіти
Мисика І. С.
Прізвище, ініціали

факультет ПКТС, 4 курс, група ПЗ-17-1

ЗАЯВА

З правилами чинного Положення «Про дотримання академічної доброчесності в Хмельницькому національному університеті» від 26.09.2020 (зі змінами від 26.11.2020), згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування заходів дисциплінарної та академічної відповідальності, ознайомлений (а). Про використання програмно-технічних засобів для перевірки кваліфікаційних робіт здобувачів вищої освіти на плагіат оповіщений (а) та надаю свою згоду на обробку та збереження університетом моєї роботи в інституційному репозитарії університету.

Також надаю університету право на передачу моєї роботи для обробки та збереження в базах даних програмно-технічних засобів (Unicheck та Anti-Plagiarism) та використання роботи для виявлення плагіату в інших роботах, які перевіряються програмно-технічними засобами та користувачами, що мають доступ до цих програмно-технічних засобів, виключно в обмежених цілях для виявлення плагіату в текстах робіт.

Робота для перевірки університетом надасться в друкованому та електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

08.06.2021р
дата

підпис

UNICHECK

Ім'я користувача:
Кафедра ІПЗ

ID перевірки:
1008234500

Дата перевірки:
08.06.2021 19:18:59 EEST

Тип перевірки:
Doc vs Internet + Library

Дата звіту:
08.06.2021 19:25:24 EEST

ID користувача:
100005589

Назва документа: **ІПЗ_17_1_Мисик_Іван_Дипломний_проект1**

Кількість сторінок: **93** Кількість слів: **13836** Кількість символів: **113255** Розмір файлу: **15.83 MB** ID файлу: **1008307484**

Виявлено модифікації тексту (можуть впливати на відсоток схожості)

4.24%
Схожість

Найбільша схожість: **1.45%** з джерелом з Бібліотеки (ID файлу: **1008265198**)

2.52% Джерела з Інтернету	152	Сторінка 95
2.19% Джерела з Бібліотеки	48	Сторінка 96

0% Цитат

Вилучення цитат вимкнене

Вилучення списку бібліографічних посилань вимкнене

0%
Вилучень

Немає вилучених джерел

Модифікації

Виявлено модифікації тексту. Детальна інформація доступна в онлайн-звіті.

Замінені символи	2
Підозріле форматування	17 сторінок

Tue Jun 08 18:14:25 EEST 2021, Хіврич Володимир Русланович, Хмельницький національний університет, ХНУ

Anti-Plagiarism v-15.257

Максимальне співпадіння з одним документом 20.0%

Словники перевірки: en_US, ru_RU, ua_UA. **Помилок в документах: 14%**

ID: 92766 Назва: Програмне забезпечення для автоматизації обміну замовленнями між різними сервісами таксі Додано в БД: 2021-06-08 Автора: І. С. Мисик Керівники: Л.П. Бедратюк Консультанти: Опоненти:	Документ		Сумарний збіг по Базі Даних	
	Символи	Лексеми	Символи	Лексеми
	89597	939	18556 (21%)	192 (20%)

Джерело плагіату

ID	Опис	Наявність плагіату в документі	
		Символи	Лексеми
90219	Назва: Звіт з переддипломної практики Мисик І.С. Додано в БД: 2021-05-11 Автора: Мисик І.С. Керівники: Бедратюк Л.П. Консультанти: Опоненти:	17798 (20.0%)	188 (20.0%)

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА ДИПЛОМНИЙ ПРОЕКТ
освітнього ступеня «Бакалавр»

Дипломник Мисик Іван Сергійович

Тема «Програмне забезпечення для автоматизації обміну замовленнями між різними сервісами таксі»

Спеціальність 121 – Інженерія програмного забезпечення

Обсяг дипломного проекту:

Кількість листів креслень _____ кількість сторінок записки 64

1.Короткий зміст пояснювальної записки та прийнятих рішень У бакалаврській роботі виконано аналіз сучасних сервісів таксі. Побудовано модель роботи сервісу таксі. Розроблено модель сервісу для обміну замовленнями на основі роботи таксі сервісів. Розроблено архітектуру сервісу для обміну замовленнями. Побудовано алгоритм розподілення замовлень між сервісами. Виконано програмну реалізацію сервісу для обміну замовленнями.

2. Висновок про відповідність проекту поставленому завданню Дипломна робота освітнього ступеня «бакалавр» в повній мірі відповідає поставленому завданню як у теоретичній, так і у практичній її частині.

3. Характеристика виконання кожного розділу проекту, ступінь використання останніх досягнень науки і техніки та передових методів роботи У вступі обґрунтовується актуальність теми проекту, проводиться попередній аналіз досліджуваної проблеми, встановлюється мета та завдання на виконання проекту. У першому розділі в повній мірі досліджено структуру сервісів таксі, проаналізоване наявне програмне забезпечення, визначені вимоги до проектування та реалізації системи. В другому розділі проведене аналіз існуючої архітектури та підходів до реалізації. В третьому розділі проведена реалізація та повний опис реалізованої системи. В четвертому розділі проведений аналіз методів тестування та застосовані конкретні підходи в тестуванні програмного забезпечення.

4. Позитивні сторони проекту Дипломний проект містить детальний аналіз роботи сучасних сервісів таксі. Сформована специфіка роботи обмінників замовлень, що вказує на низку рішень які покращать обслуговування клієнтів у сфері таксі. Побудований ефективний алгоритм розподілення замовлень з гнучким налаштуванням фільтрування та розсилання замовлень.

5. Негативні сторони проекту Програмне забезпечення, що пропонується в дипломному проекті не протестоване в інтеграції з реальним сервісом таксі, на реальних умовах, через що, ефективність роботи залишається на теоретичному рівні.

6. Оцінка графічного оформлення та пояснювальної записки проекту Графічне оформлення виконано відповідно до теми дипломного проекту з дотриманням вимог стандартів. Рівень графічного оформлення достатній для ілюстрування всіх принципів та процесів, що описані в дипломному проекті.

7. Відгук про дипломний проект в цілому Загалом дипломний проект повноцінно описує вирішення поставленої задачі. Весь матеріал проекту послідовний та чіткий. Всі розділи логічні та повноцінно описані, що дозволяє зрозуміти весь проект. Графічні матеріали наочно показують всі особливі моменти, що спрощує розуміння вирішення задачі. В цілому проект заслуговує позитивної оцінки.

8. Інші зауваження

9. Оцінка дипломного проекту Розглянувши всі позитивні та негативні сторони представленого дипломного проекту можна зробити висновок, що він заслуговує оцінки «відмінно» (4.75/A).

РЕЦЕНЗЕНТ (прізвище, ім'я, по-батькові, посада, місце роботи)

Говорущенко Тетяна Олександрівна, доктор технічних наук, професор, завідувач кафедри комп'ютерної інженерії та системного програмування Хмельницького національного університету

«29» 05 2021 р.

(підпис)

РІШЕННЯ ЕКСПЕРНОЇ КОМІСІЇ

КАФЕДРИ ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Підтверджуємо ознайомлення з результатом звіту подібності щодо роботи, генерованого системою виявлення текстових збігів/ідентичності/схожості:

Назва: Програмне забезпечення для автоматизації обміну замовленнями між різними сервісами таксі

Автор: Мисик Іван Сергійович

Спеціальність: 121 – Інженерія програмного забезпечення

Освітня програма: Освітньо-професійна програма «Інженерія програмного забезпечення»

Науковий керівник: Бедратюк Леонід Петрович д-р фіз-мат. наук, професор

Після аналізу звіту подібності зроблено такий висновок:

№	Висновок	Позначка про відповідність
1	Запозичення, виявлені в роботі, є законними і не є плагіатом. Робота приймається до захисту.	відповідає
2	Виявлені запозичення не є плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи. Робота приймається до захисту, але має бути відкоригована. Відкоригований варіант має бути поданий на кафедру за 2 дні до захисту, разом із заявою щодо самостійності виконання письмової роботи та ідентичності друкованої та електронної версії роботи	
3	Виявлені запозичення не є плагіатом, але частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. В зв'язку з цим мета роботи та поставлені завдання не були досягнені. Робота може бути допущена до захисту (наступного року) після того як буде відкоригована та допрацьована і успішно пройде повторну перевірку на академічний плагіат.	
4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	
5	Інше:	

Підтвердження:

Запозичення, виявлені в роботі, є законними і не є плагіатом, оскільки:

1) у тексті дипломного проекту системами перевірки на плагіат виявлено схожість з деякими документами в частині загальновживаних обов'язкових словосполучень в бланках (титулка, бланк завдання, в структурі підрозділів ВСТУПУ) та в назвах публікацій джерел;
2) В якості запозичень системою було зафіксовано послідовність вихідного коду, які є спільними для великої кількості задач і не можуть розглядатися як об'єкт авторських прав і, відповідно, їх порушення;
3) усі запозичення фрагментарні, або мають належним чином оформлені посилання.

Сумарний обсяг всіх запозичень, визначений системою виявлення збігів ідентичності / схожості, складає 6,07% і адресується до 604 першоджерел, що з урахуванням наведених обґрунтувань, відповідає характеру теми і свідчить на користь дипломного проекту.

Керівник _____ Л.П. Бедратюк

Гарант ОП _____ Л.П. Бедратюк

Завідувач кафедри _____ Л.П. Бедратюк