Хмельницький національний університет
Факультет інформаційних технологій
Кафедра інженерії програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА

Серверна частина REST API для кросплатформного фріланс-сервісу LiveFreelance
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 КвРІПЗ.200128.01.11.ПЗ

Виконав студент III курсу група ІПЗс-20-1 _____ Підпис Д. А. Ямборко Ініціали, прізвище

Керівник канд. техн. наук, доцент Науковий ступінь, звання _____ Підпис Г. І. Радельчук Ініціали, прізвище

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

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

5 червня 2023 р.

Хмельницький 2023

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Інформаційних технологій
Кафедра Інженерії програмного забезпечення
Рівень вищої освіти Перший (бакалаврський)
Галузь знань 12 «Інформаційні технології»
Спеціальність 121 «Інженерія програмного забезпечення»
Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Л. П. Бедратюк
02 01 2023 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Ямборку Дмитру Анатолійовичу
Прізвище, ім'я, по батькові студента

1. Тема кваліфікаційної роботи Серверна частина REST API для кросплатформного фріланс-сервісу LiveFreelance

Керівник кваліфікаційної роботи Радельчук Галина Іванівна, канд. техн. наук, доцент
Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 01.03.2023 р. № 5
2. Строк подання студентом проекту (роботи) на кафедру 01.06.2023 р.
3. Вихідні дані до роботи Матеріали переддипломної практики
4. Зміст пояснювальної записки (перелік питань, які потрібно розробити) Дослідження предметної області та постановка задачі, проектування програмного забезпечення, програмна реалізація та тестування програмного забезпечення
5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень) 1. Три креслення у форматі А3 (UML-діаграма варіантів використання. Інфологічна модель бази даних. UML-діаграма компонентів)

6. Консультанти розділів кваліфікаційної роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Радельчук Г. І., доцент кафедри ІПЗ	25.05.2023	01.06.2023
Антиплагіат	Гурман І. В., доцент кафедри ІПЗ	3.06.2023	5.06.2023

7. Дата видачі завдання « 02 » січня 2023 р.

КАЛЕНДАРНИЙ ПЛАН

Назва етапів (розділів) кваліфікаційної роботи	Строк виконання етапів кваліфікаційної роботи	Примітка
1 Ознайомлення з тематикою кваліфікаційних робіт (КвР), визначення та узгодження індивідуальної теми КвР	01.12 – 31.12.2022	
2 Дослідження предметної області, в якій планується використання програмного засобу (ПЗ), визначення задач та вимог, розробка технічного завдання	02.01 – 31.01.2023	
3 Проектування програмного забезпечення. Розробка графічної частини КвР	01.02 – 28.02.2023	
4 Програмна реалізація	01.03 – 10.04.2023	
5 Тестування програмного забезпечення	11.04 – 30.04.2023	
6 Написання вступу, загальних висновків, оформлення джерел посилання та додатків. Оформлення пояснювальної записки КвР згідно вимог стандартів	01.05 – 25.05.2023	
7 Попередній захист КвР	Травень 2023 (згідно графіка)	
8 Перевірка КвР на плагіат, нормконтроль, отримання відгуків та рецензій. Брошурування (зшиття) пояснювальної записки	26.05 – 30.05.2023	
9 Підготовка до захисту та захист КвР	з 01.06.2023	

Студент _____ Підпис Д. А. Ямборко Ініціали, прізвище

Керівник роботи _____ Підпис Г. І. Радельчук Ініціали, прізвище

АНОТАЦІЯ

Тема кваліфікаційної роботи: Серверна частина REST API для кросплатформного фріланс-сервісу LiveFreelance.

Автор роботи: Ямборко Дмитро Анатолійович.

Керівник роботи: Радельчук Галина Іванівна.

Пояснювальна записка: 67 с., 26 рис., 4 табл., 3 дод., 40 джерел.

Графічна частина: три креслення у ф. А3.

ФРІЛАНС-СЕРВІС, СЕРВЕРНА ЧАСТИНА, REST API, КРОСПЛАТФОРМА.

Метою кваліфікаційної роботи є розробка серверної частини для фріланс-сервісу, який дозволить особам різних категорій зайнятості знаходити роботу без зайвих складнощів, а роботодавцям знаходити висококваліфікованих фахівців, які можуть виконати необхідну роботу.

В ході роботи було проаналізовано предметну область, проведено оцінку актуальності та доцільності розробки, розглянуто програми-аналоги. Виконана поста-новка завдання та визначені вимоги до програми, розроблено специфікацію програм-ного продукту, а також спроектовано архітектуру додатку та структуру бази даних.

Серверну частину REST API було реалізовано з використанням мови програмування TypeScript/Javascript, платформи Node.js, фреймворку Nest.js та системи керування базами даних MySQL.

В результаті було розроблено серверну частину REST API для кросплатформного фріланс-сервісу LiveFreelance.

01.06.2023
Дата

Підпис

ВІДОМІСТЬ ДОКУМЕНТІВ

№ рядка	Формат	Позначення документа	Найменування документа	К-сть аркушів	№ екз.	Примітка
			Текстові документи			
1	А4	КвРІПЗ.200128.01.11.ПЗ	Пояснювальна записка	67		
2	А4		Завдання на	1		
			кваліфікаційну роботу			
3	А4		Анотація	1		
			Графічні документи			
4	А3	КвРІПЗ.200128.01.11.Е8	UML-діаграма варіантів			
			використання			
5	А3	КвРІПЗ.200128.01.11.Е8	Інфологічна модель бази	1		
			даних			
6	А3	КвРІПЗ.200128.01.11.Е8	UML-діаграма	1		
			компонентів			

Змн.	Арк.	№ докум.	Підпис	Дата	КвРІПЗ.200128.01.11.ВД		
Виконав		Ямборко Д.А.		01.06	Серверна частина REST API для кросплатформного фріланс-сервісу LiveFreelance	Літ.	Арк. 1 / Акрушів 1
Керівник		Радельчук Г.І.		01.06			
Н. Контр.		Радельчук Г.І.		01.06		ХНУ, ІПЗс-20-1	
Зав. Каф.		Бедратюк Л.П.		5.06	Відомість документів		

ЗМІСТ

ВСТУП

У наш час досить важко, без досвіду, знайти висококваліфікованого спеціаліста в певній галузі, так само спеціалісту досить важко знайти роботу, тому що конкуренція на ринку праці досить велика. Тому цей процес пошуку фахівця чи роботи потребує спрощення та автоматизації.

Програмне забезпечення використовують для автоматизації різних процесів. Найпоширенішими засобами автоматизації сьогодні, є веб додатки, кросплатформені додатки, веб сайти, десктопні програми, боти в популярних мессенджерах тощо.

Однією з поширених сфер, в яких активно ведеться автоматизація процесів, є сфера надання різних послуг, сфери пошуку роботи та працевлаштування. Біржа праці, центр зайнятості, різноманітні дошки оголошень, газети, всі ці засоби не завжди встановлюють прямий зв'язок між виконавцем і замовником і не кожен з цих засобів може бути автоматизованим, тому є досить неефективними та застарілими у такому вигляді як є зараз.

Кожна людина, коли потребує певну послугу або фахівця в певній області, наприклад сантехніка чи вантажні перевезення, стикається з певними труднощами, так само людина яка надає послуги, може мати потребу в певній клієнтській базі. Використання газет, оголошень, реклами та листівок не є ефективним для вирішення подібних задач, тому з’являється необхідність розробки автоматизованих систем.

Наявність фріланс-сервісу дійсно має багато переваг як для фахівців, так і для клієнтів, які шукають виконавців для своїх проектів. Ось кілька переваг, які надає фріланс-сервіс:

– широкий вибір фахівців (фріланс-сервіси надають доступ до великої бази даних фахівців з різних галузей; це означає, що клієнти можуть знайти саме тих спеціалістів, які потрібні для їхніх проектів, з різними навичками та досвідом);

– зручність та ефективність (фріланс-сервіси надають зручний і ефективний спосіб комунікації між клієнтом та виконавцем; вони забезпечують інструменти для обміну повідомленнями, обговорення деталей проекту, взаємного відстеження прогресу та вирішення потенційних питань);

– зменшення посередників (фріланс-сервіси дозволяють клієнтам та виконавцям прямо спілкуватися між собою, обходячи посередників; це зменшує витрати та складнощі, пов'язані зі звичайними посередницькими послугами, і дозволяє обом сторонам встановлювати пряму комунікацію та узгоджувати умови співпраці без зайвих обмежень);

– огляди та рейтинги (багато фріланс-сервісів надають систему оглядів та рейтингів, яка дозволяє клієнтам оцінювати роботу виконавців, а також ділитися своїм досвідом з іншими користувачами; це допомагає забезпечити високу якість послуг та підтримувати довіру між сторонами);

– глобальний доступ (фріланс-сервіси дозволяють знайти виконавців та замовників з усього світу; це дає можливість працювати з фахівцями з будь-якої країни, що розширює географію можливостей та забезпечує доступ до найкращих талантів та творчих людей).

Загалом, фріланс-сервіси спрощують пошук фахівців та надають зручний спосіб спілкування та співпраці між клієнтом та виконавцем. Вони підвищують ефективність робочого процесу, допомагають знайти відповідних спеціалістів та забезпечують високу якість виконання проектів.

Ця система покликана автоматизувати процес та спростити процес надання послуг, формування певних відгуків та рейтингу фахівців, виведення певної статистики, зменшення кількості ресурсних затрат.

Наявність фріланс-сервісу значно спрощує пошук роботи фахівцям та пошук виконавця певної послуги чи роботи клієнту. Цей сервіс забезпечує ефективну комунікацію між клієнтом та замовником, що суттєво підвищить якість організації робочого процесу, та ліквідує посередників.

Мета кваліфікаційної роботи – розробка серверної частини для фріланс-сервісу, який дозволить особам різних категорій зайнятості знаходити роботу без

зайвих складнощів, а роботодавцям знаходити висококваліфікованих фахівців, які можуть виконати необхідну роботу.

Завдання, які необхідно вирішити для досягнення мети:

– проаналізувати предметну область для визначення її головних особливостей та характеристик;

– проаналізувати аналогічне програмне забезпечення;

– визначити вимоги до програми та розробити технічне завдання;

– спроектувати програмний продукт;

– виконати програмну реалізацію;

– протестувати програму.

1 ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Змістовний аналіз предметної області, її структурних та функціональних особливостей

Пошук фахівця в певній галузі та пошук роботи є однією з актуальних проблем сьогодення. Замовнику важко знайти людину, яка якісно виконає роботу, так само фахівцю, який немає досвіду, досить важко знайти роботу. Тому для цього створені фріланс-сервіси, які забезпечують прямий контакт замовника з виконавцем без певних посередників.

Сьогодні існує досить мало сервісів та додатків які допомагають вирішити проблеми пошуку виконавця певної послуги та знаходження роботи, особливо в невеликих містах. Самостійно майже неможливо знайти виконавця чи замовника, без попереднього досвіду, чи досвіду знайомих. Буває вкрай складно вибрати з усього того, що пропонують існуючі методики.

Фріланс-сервіси стали все більш популярними останнім часом, так як це місце де клієнт, який має потребу в певній послузі, може знайти виконавця, з яким домовиться за спрятливими для них умовами, зазвичай ціна буде нижчою, ніж при звернені за такою ж послугою до певної компанії, в свою чергу для виконавця досить просто знайти якусь роботу або підробіток, маючи кваліфікацію в певній галузі, зазвичай щоб влаштуватись працювати на певне підприємство, роботодавець вимагає від працівника певного досвіду, але досить часто молоді фахівці немають цього досвіду і можуть його отримати саме на подібних сервісах та спеціальних майданчиках.

Типовий фріланс-сервіс, це місце де замовник і виконавець напряму контактують один з одним, подібні сервіси мають зручний та доступний інтерфейс, просту реєстрацію, перегляд фахівців, за їх місцерозташуванням, відгуки та рейтинги виконавців для підтвердження своєї кваліфікації та досвіду взаємодії з іншими користувачами системи.

1.2 Аналіз наявного програмно-технічного забезпечення предметної області

Сьогодні існує безліч способів та видів реалізації фріланс-сервісів, від невеликих веб-сторінок до повноцінних кросплатформених додатків.

Численні підходи виявляються неефективними через недостатню кількість інформації, яку надає сервіс, що перешкоджає встановленню контакту між замовниками та підрядниками. Низькофункціональні системи не мають комунікаційних можливостей і не надають вичерпної інформації про угоди, укладені між замовниками та спеціалістами. Як наслідок, загальна ефективність системи знижується.

Наприклад, розглянемо ситуацію, коли людині потрібен висококваліфікований автомеханік. Вона натрапляє на численні оголошення та пропозиції, але в системі відсутня інформація про наявних спеціалістів, що призводить до невизначеності та вагань. Більше того, у багатьох системах відсутня важлива інформація про виконавця, наприклад, його місцезнаходження, фотографія або підтверджені дипломи та сертифікати в певній галузі, що робить процес пошуку ще більш складним. Така нестача інформації ускладнює процес пошуку та відбору. Аналогічно, з точки зору підрядника, недостатня інформація про його місцезнаходження, рейтинг та відгуки на платформі може зробити його вразливим до потенційних шахрайських дій під час пошуку клієнтів. Щоб усунути ці недоліки та вдосконалити систему, необхідно розробити рішення, яке надаватиме вичерпну інформацію як про замовників, так і про фахівців, включаючи їхню кваліфікацію, місцезнаходження, облікові дані та відгуки користувачів, які з ним взаємодіяли.

Для вирішення цих проблем використовують повноцінні системи, які містять в собі розгорнуту інформацію, як про клієнта, так і про виконавця і допомагають встановити контакт між ними. Такі рішення інтегрують можливості вище перелічених неповноцінних сервісів в одну єдину систему.

Провівши пошук існуючих методів та рішень для реалізації фріланс-сервісу, було виявлено декілька систем, які частково вирішують проблему.

Фріланс-сервіс «Freelance.ua» (рисунок 1.1)

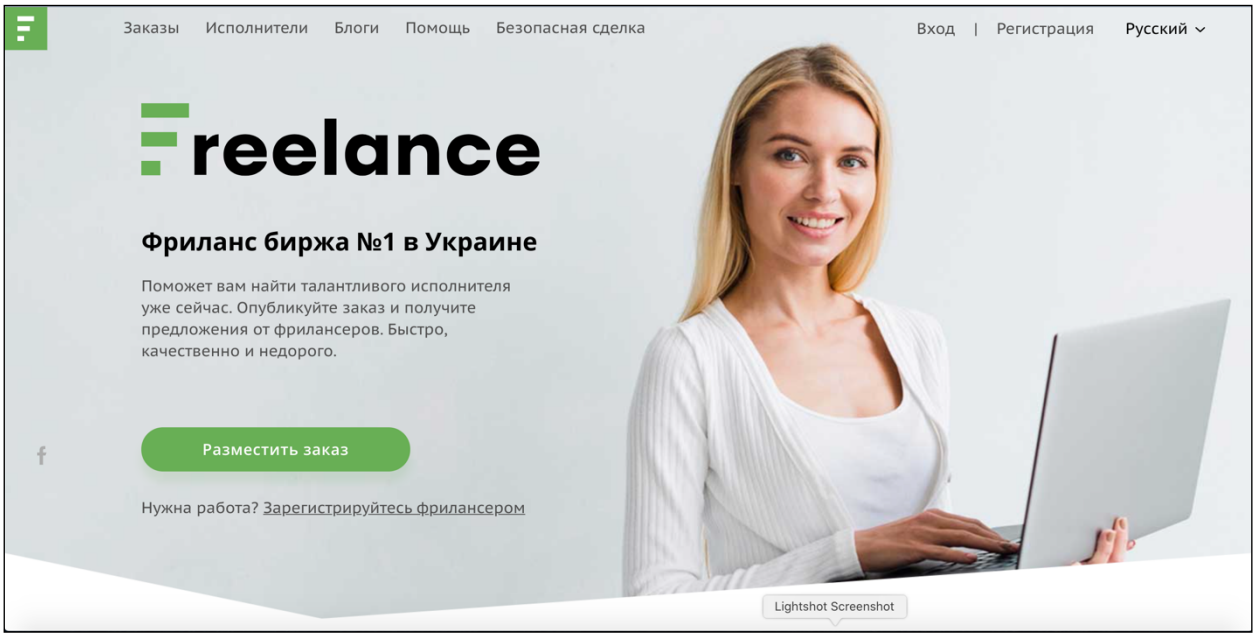


Рисунок 1.1 – Фріланс-сервіс «Freelance.ua» [1]

Український фріланс-сервіс «Frelance.ua» [1] – це онлайн-сервіс, який допоможе знайти талановитого виконавця за короткий термін. Система дозволяє зареєструватись як виконавцем, так і замовником, а також переглядати різні замовлення, фільтрувати їх по спеціалізації, даті, регіону, ціні і створити замовлення самому, якщо ви є замовником.

Сервіс підтримує функцію «Безпечна угода», яка мінімізує ризики шахрайства як з боку замовника, так і з боку виконавця.

Серед недоліків системи можна виділити те, що існує досить велика конкуренція серед виконавців і виконавців без досвіду, дуже тяжко знайти замовлення. Безкоштовна версія обмежена у функціоналі, а повна pro-версія є платною. Суттєвим недоліком також є вузькість спеціалізацій, які базуються більше на онлайн-послугах.

Фріланс-сервіс «Kabanchik.ua» (рисунок 1.2)

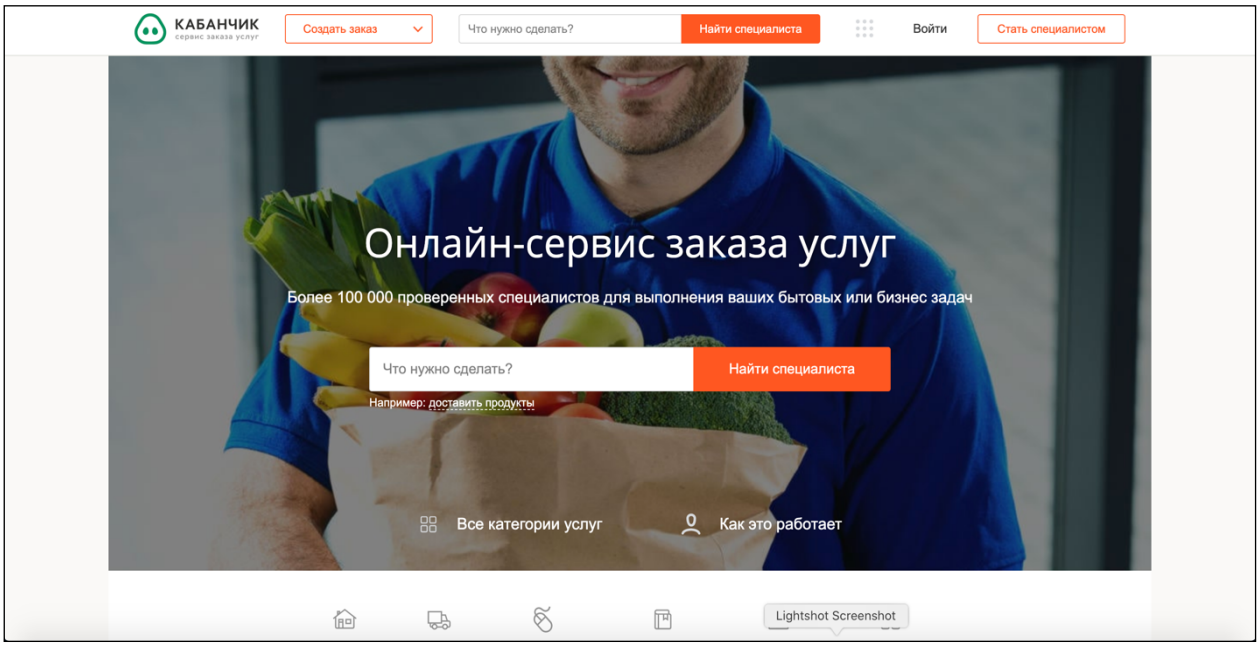


Рисунок 1.2 – Фріланс-сервіс «Kabanchik.ua» [2]

Фріланс-сервіс «Kabanchik.ua» [2] – веб-сервіс, який допомагає знайти компетентного фрілансера для виконання онлайн-роботи. Система надає такий функціонал: особисті аккаунти клієнтів, створення замовлень і пошук по них. До переваг системи можна віднести достатню кількість спеціалізацій, безліч функціоналу, та можливість безкоштовного використання.

До недоліків «Kabanchik.ua» можна віднести складність у використанні і налаштуванні. Через велику кількість налаштувань та опцій, які не завжди необхідні, новачку досить важко зорієнтуватись. Це також український сервіс, тому не весь функціонал буде доступним в інших країнах.

Фріланс-сервіс «Freelancehunt.ua» (рисунок 1.3)

«Freelancehunt.ua» [3] на сьогодні є провідним сервісом в Україні та займає високі позиції в Казахстані, Білорусі. Близько 70% користувачів сервісу живуть і працюють в Україні. Решта приєднується до нього із СНД та інших країн світу з різних континентів.

До переваг сервісу можна віднести зручний та інтуїтивний інтерфейс, простоту у використані, наявність бази знань, де користувач може отримати додаткову інформацію про сервіс. Система має різноманітні конкурси та бонусні програми, на сайті регулярно проводяться акції.

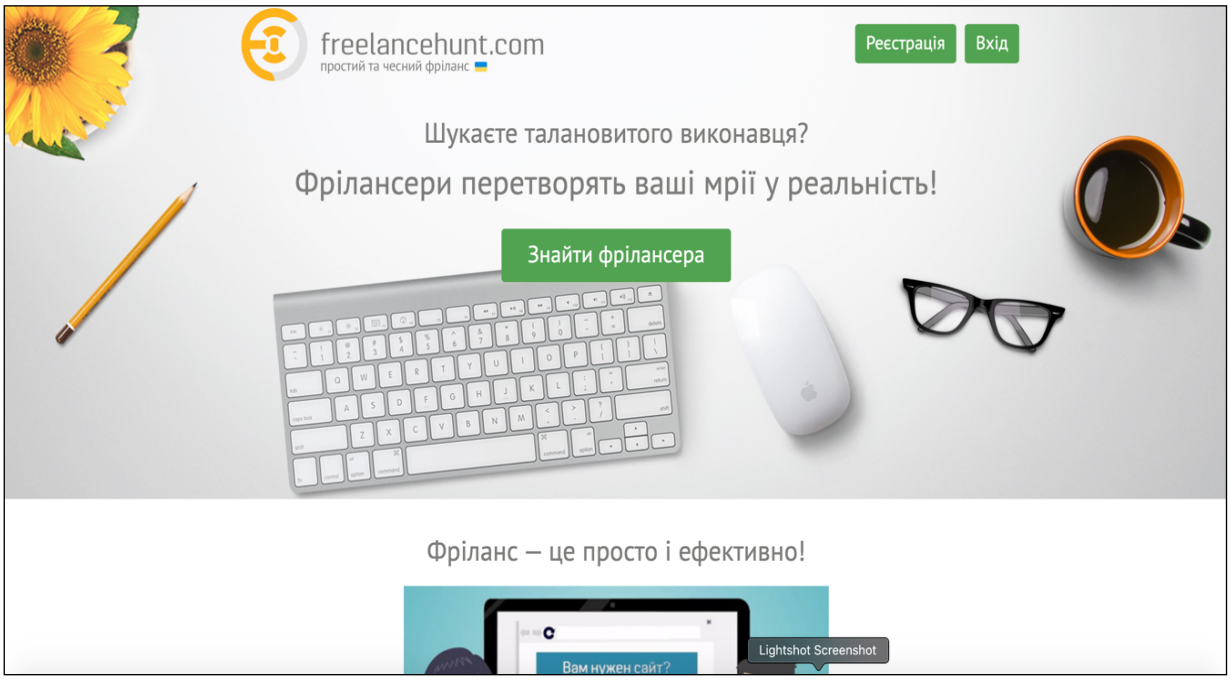


Рисунок 1.3 – Фріланс-сервіс «Freelancehunt.ua» [3]

До недоліків можна віднести дуже малу кількість спеціальностей, заточених під онлайн-послуги, відсутність додатків на IOS та Android, закритий API та досить нечитабельну документацію.

Фріланс-сервіс «Weblancer.net» (рисунок 1.4)

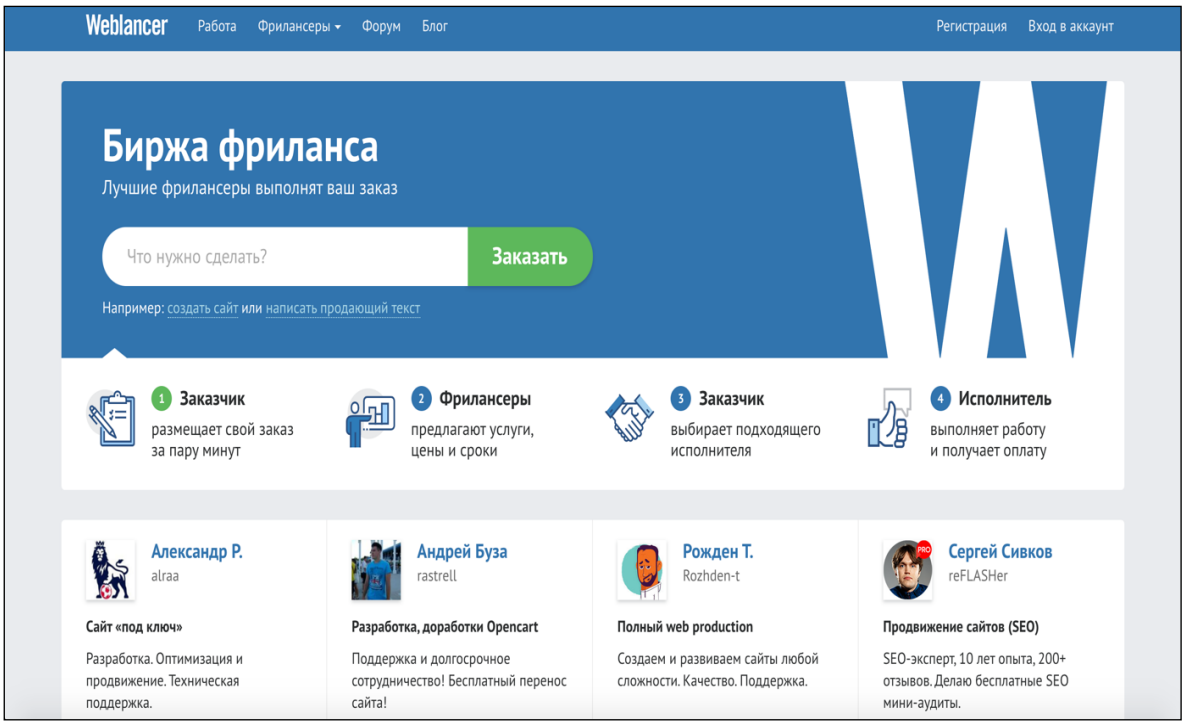


Рисунок 1.4 – Фріланс-сервіс «Weblancer.net» [4]

Система «Weblancer.net» [4] – це фріланс-біржа, яка стала піонером у своїй області в СНГ-сегменті. Сталося це у 2003 році, коли віддалена зайнятість практично не розглядалася у контексті окремого виду трудової діяльності.

Як і подібні сервіси, система містить можливість реєстрації виконавців та замовників, створення замовлень, поділених на категорії.

Серед переваг сервісу, можна визначити довгостроковість його роботи, простота інтерфейсу.

Суттєвим недоліком є платна версія, невелика кількість спеціалізацій, відсутність мобільних додатків.

У таблиці 1.1 показано порівняння додатків за перерахованими критеріями.

Таблиця 1.1 – Порівняльна таблиця характеристик додатків

Назва додатку	Засоби форматування тексту	Зручність інтерфейсу	Можливість кастомізації	Реклама	Наявність мобільного додатку	Бекап та експорт	Пароль	Безкоштовний
Freelance.ua	Обмежені в безкоштовній версії	6/10	Так, обмежена в безкоштовній версії	Дуже багато, займає значну площу	−	Тільки в платній версії	Так	Так
Kabanchik	Відсутні	8/10	Зміна головного кольору	Помірно	−	Архів у безкоштовній версії, експорт у PDF та хмару в платній	Так	Ні
Freelancehunt	Відсутні	5/10	Відсутня	Помірно	−	Відсутня	Так	Так
Weblancer	Відсутні	5/10	Відсутня	Дуже багато, займає значну площу	+	Синхронізація через Google	Ні	Ні

Цей розділ зосереджується на предметній області фрілансу та висвітлює завдання і проблеми, які можуть бути вирішені за допомогою онлайн-систем.

Розглядаються різні існуючі рішення, що полегшують пряму комунікацію між клієнтами та фрілансерами. Однак багато з розглянутих прикладів пропонують платні версії, пристосовані до конкретних онлайн-сервісів і не мають еквівалентних мобільних додатків з аналогічним функціоналом.

Тому було прийнято рішення розробити API для сервісу, який би усунув обмеження, знайдені в існуючих системах.

1.3 Визначення вимог до серверної частини

Дане програмне забезпечення призначене для реалізації фріланс-сервісу та для допомоги встановлення трудових відносин між замовником і виконавцем.

Сервіс спрощує взаємодію виконавця і замовника, автоматизує безліч процесів, які виникають під час надання чи виконання певної роботи. Також сервіс надає безліч можливостей як виконавцю, так і замовнику. І надає безліч інформації про кожну зі сторін, що значно спрощує пошук як клієнта, так і виконавця. Провівши пошук аналогів програмних сервісів, було виявлено декілька систем, що вирішують проблематику предметної області, проте мають свої недоліки. Знайдені програмні продукти вказані у таблиці 1.1.

Автоматизована система повинна формально мати три типи користувачів, але так як замовник може, у свою чергу, бути і виконавцем і навпаки, то по факту сервіс має дві ролі – користувач та адміністратор.

Для адміністратора слід реалізувати такий функціонал:

– облік користувачів;
– авторизація;
– облік документів користувачів;
– статистика;
– модерування відгуків користувачів;
– оновлення власного профілю.

Для користувача слід передбачити такі можливості:

– авторизація і реєстрація;

– перегляд всіх користувачів;

– редагування профілю;

– додання сфер діяльності, в який. користувач може надавати послуги;

– додання документів, які підтверджують спеціалізація користувача;

– надсилання заявок на певну послугу іншим користувачам;

– написання відгуків іншим користувачам, після виконаного замовлення.

При роботі системи використовується два типи користувачів – користувач і адміністратор. Кожен користувач має обмежений набір можливостей, які необхідні даному класу. Незареєстровані користувачі не мають доступу до системи.

API для фріланс-сервісу передбачає наявність серверної частини.

Для забезпечення стабільної та швидкої роботи системи до серверної частини висуваються такі апаратні та програмні вимоги:

– операційна система: Linux Ubuntu 18.4 і вище;

– процесор: Intel/AMD з тактовою частотою від 2ГГц, x64;

– оперативна пам’ять: від 2 Гб;

– дисковий простір: від 3 Гб;

– встановлене ПЗ: Docker;

– наявність підключення до мережі Інтернет.

ПЗ повинно забезпечити можливість здійснювати пошук та перегляд інформації про виконавців, за містом, сферою діяльності, додавати інформацію про себе, редагувати або видаляти існуючі поля.

Також користувач має мати можливість додавати сферу послуг, яка додастmся в базу даних. Користувач може додати документи, сертифікати чи інші речі, які підтверджують кваліфікацію цього користувача у його сфері діяльності. До кожного документа має бути прикріплено фото та опис, після завантаження та додання документів, документ автоматично відобразиться адміністратору, який в свою чергу може прийняти або відхилити документ.

Система має дозволяти адміністратору переглядати статистику роботи сервісу. До статистики повинна входити інформація про кількість користувачів

згрупованих за різними полями чи властивостями, інформація про найпопулярніші сервіси, тощо.

Також має бути реалізований функціонал, який дозволяє користувачам ставити оцінки та писати відгуки один - одному, після завершеної роботи.

Інтерфейс програми повинен бути інтуїтивно - зрозумілим, забезпечувати швидку навігацію - по основних функціональних можливостях, та не вводити в оману користувача. Інтерфейс системи повинен не дозволяти вводити некоректні дані, де це можливо, та видавати інформаційні повідомлення про помилки. В процесі розробки кваліфікаційної роботи було реалізовану тільки серверну частину без графічного інтерфейсу.

Для зберігання даних використано реляційну СУБД PostgreSQL від компанії PostgreSQL Global Development Group. Зв’язок серверної частини системи та бази даних відбувається за допомогою технології TypeORM.

Зв'язок між клієнтом та веб сервером здійснюється за допомогою мережевого протоколу HTTP та WebSocket. Дані передаються у форматі JSON.

Інші нефункціональні вимоги

Не зареєстровані користувачі не мають доступу до системи. Зареєстровані користувачі мають доступ до розділів системи, відповідно до класу (користувач, адміністратор). Користувач не має доступу до тих частин системи, які не визначені для даного класу. Програмний засіб повинен забезпечувати реалізацію усього функціоналу фріланс системи. Програмне забезпечення повинне легко адаптуватись та, бути достатньо гнучким для використання. Також програма повиненна бути доступною на багатьох платформах і адаптована під різних користувачів з різними можливостями.

1.4 Постановка задачі

Темою кваліфікаційної роботи є «Серверна частина REST API для кроспла–тформного фріланс-сервісу LiveFreelance». Метою кваліфікаційної роботи є

створення серверної частини (API) для кросплатформенго додатку фріланс-сервісу LiveFrelance.

Слід реалізувати REST API, який буде зберігати, обробляти та отримувати інформацію з бази даних, а саме:

– реалізувати реєстрацію та автентифікацію користувача;

– редагування профілю користувача;

– додання сфер діяльності, в яких користувач може надавати свої послуги;

– створити можливість надсилання заявок іншим користувачам стосовно замовлення роботи у певній сфері діяльності;

– реалізувати можливість додання документів, які будуть підтверджувати кваліфікацію користувача в певній послузі;

– створити рейтинг користувачів, який буде базуватись на відгукам, які користувачі будуть залишати після виконаних заявок;

– реалізувати функціонал адміністративної панелі, де адміністратор зможе підтверджувати чи відхиляти документи і контролювати користувачів.

Висновок. В даному розділі було описано предметну область сфери фрілансу, завдання та проблеми що вирішуються за допомогою онлайн систем. Розглянуто існуючі рішення, що забезпечують безпосередній контакт замовника з виконавцем, який буде виконувати відповідну роботу.

Більшість розглянутих прикладів мають платну версію, заточені під онлайн послуги і не мають мобільних додатків з таким самим функцоналом. Тому було вирішено створити API для сервісу, який вирішить більшість недоліків існуючих систем, та буде реалізовувати додатковий функціонал.

Технічне завдання на розробку подано у додатку А.

2 ПРОЕКТУВАННЯ СЕРВЕРНОЇ ЧАСТИНИ

2.1 Архітектура та функціональна структура серверної частини

Автоматизовані системи зазвичай реалізуються у вигляді настільного або веб додатку. Мобільні платформи використовуються рідше, оскільки робота з такими системами не вимагає мобільності. Було реалізовано REST api, з якими можна працювати з будь-яких клієнтів, як з мобільних додатків Android та IOS.

Архітектура клієнт-сервер є широко використовуваною моделлю для розробки автоматизованих систем. Вона дозволяє розділити функціональність системи між сервером, який надає послуги, та клієнтами, які користуються цими послугами. Одним із важливих аспектів цієї архітектури є забезпечення безпеки та захисту даних користувачів.

Клієнт-серверна архітектура є однією з найпоширеніших моделей архітектури для розробки програмних систем. Проте, існують інші моделі архітектури, які можуть бути використані залежно від потреб і вимог конкретного проекту. Ось декілька інших моделей архітектури, які можна порівняти з клієнт-серверною.

Peer-to-Peer (P2P) архітектура

У P2P-архітектурі всі компоненти, як клієнти, так і сервери, можуть виконувати як серверну, так і клієнтську роль. Вони обмінюються ресурсами, послугами або інформацією без необхідності централізованого сервера. У цій моделі кожен компонент може бути одночасно постачальником та споживачем ресурсів один одного.

Клієнт-серверна архітектура з тонким клієнтом (Thin Client)

У цій моделі більша частина обробки відбувається на сервері, а клієнтські пристрої (тонкі клієнти) виконують базові функції та відображають результати. Тонкі клієнти мають обмежені обчислювальні та зберігаючі можливості, тому вся складна обробка відбувається на сервері.

Стрічка (Pipeline) архітектура

У цій моделі обробка даних розбивається на послідовні етапи (кроки), і кожен етап обробки виконується на окремих вузлах. Кожен вузол отримує вхідні дані з попереднього етапу, обробляє їх і передає результати наступному етапу. Ця модель дозволяє ефективно розпаралелювати обробку даних і підвищити продуктивність системи.

Мікросервісна архітектура

У мікросервісній архітектурі система розбивається на набір невеликих незалежних компонентів (мікросервісів), які можуть функціонувати незалежно один від одного. Кожен мікросервіс надає конкретну функціональність та може бути розгорнутий, масштабований та оновлюваний окремо від інших. Клієнти взаємодіють з різними мікросервісами для отримання необхідних послуг.

Кожна з цих архітектур має свої переваги та обмеження і може бути вибрана залежно від вимог та цілей проекту. Клієнт-серверна архітектура залишається популярною через свою гнучкість, масштабованість та зручну взаємодію між клієнтами та серверами.

Клієнт-серверна архітектура передбачає такі основні компоненти:

– один або декілька серверів, які надають інформацію або інші послуги програмам, які звертаються до них;

– набір клієнтів, які використовують сервіси, що надаються серверами;

– мережа, яка забезпечує взаємодію між клієнтами та серверами.

Сервери є незалежними один від одного. Клієнти також функціонують паралельно і незалежно один від одного.

Немає жорсткої прив'язки клієнтів до серверів. Більш ніж типовою є ситуація, коли один сервер одночасно обробляє запити від різних клієнтів. З іншого боку, клієнт може звертатися то до одного сервера, то до іншого. Клієнти не знають про існування один одного, звертаючись лише до серверу. Така архітектура досить часто використовується в побудові сучасниз програмних систем, так, як має безліч переваг, зокрема гучність, нам достатньо вертикально

або горизонтально масштабувати сервер і це дозволить, будь якій кількості клієнтів працювати з ним. На рисунку 2.1 наведено модель клієнт-сервер.

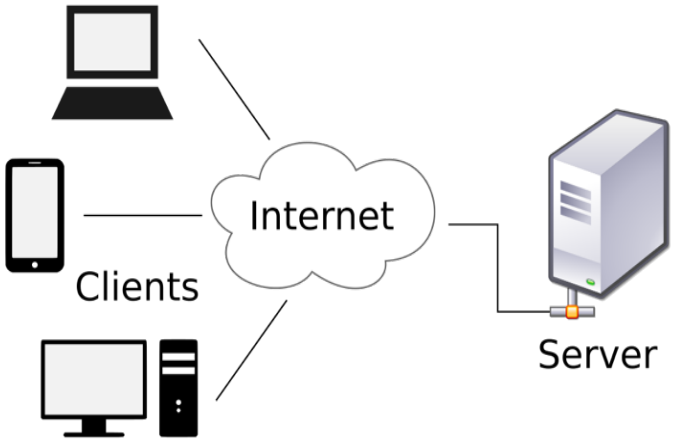


Рисунок 2.1 – Модель клієнт-сервер

Така архітектура - дозволяє взаємодіяти з додатком через ноутбук, ПК чи смартфон, в будь-який час в будь-якому місті при наявності Інтернету. Користувачам додатку не потрібно встановлювати та налаштовувати програму, достатньо зайти на адресу серверу та авторизуватись.

2.2 Визначення основних модулів серверної частини

Оскільки розроблялась серверна частина, то система буде складатись з модулів, які реалізуватимуть вказану архітектуру (модель, сервіс, репозиторій). Кожен модуль відповідатиме за роботу з окремим об’єктом або групою об’єктів предметної області. Розділимо програму на такі модулі:

- модуль керування адміністраторами;
- модуль керування користувачами;
- модуль керування містами;
- модуль керування країнами;
- модуль керування документами;

– модуль керування обраними користувачами;
– модуль репортів;
– модуль заявок;
– модуль відгуків;
– модуль сфер діяльності;
– модуль статистики;
– модуль авторизації;
– модуль кервання нотифікаціями;
– модуль керування послугами;
– модуль для обробки запитів;
– модуль авторизації та автентифікації користувача;
– модуль з допоміжними функціям.

Крім головних модулів, буде допоміжний компонент безпеки, який відповідатиме за захист додатку та авторизацію. Всі компоненти будуть взаємодіяти з модулем безпеки для перевірки, чи має поточний користувач необхідні права. Більшість з модулів будуть взаємодіяти з базою даних для виконання основних функцій. Діаграму компонентів наведено на рисунку 2.2.

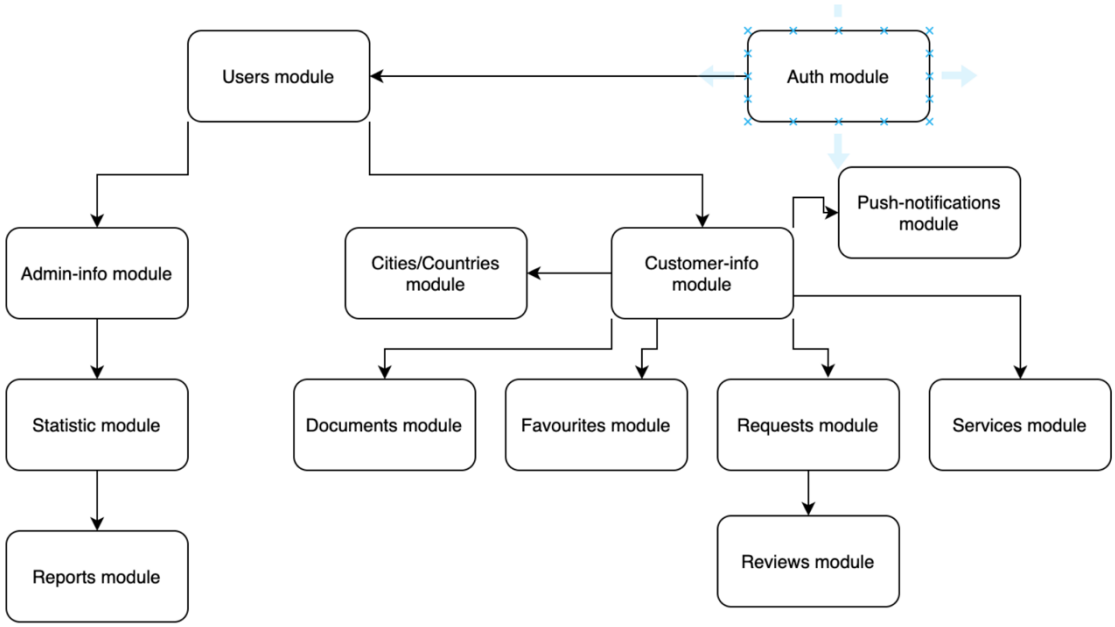


Рисунок 2.2 – Діаграма компонентів

Проектування послідовностей та станів системи

Розглянемо діаграму послідовності при додаванні документа користувачем. Користувач авторизується, після чого з’являється можливість додати документ надіславши запит з відповідними даними. Користувач в тіло запиту додає дані про документ, прикріпляє фото до запиту, після цього відбувається валідація цих даних. Якщо валідація успішна, викликається сервіс, що відповідає за керування документами. Сервіс звертається до репозиторія, а той в свою чергу до бази даних. Після додавання документа повертається відповідь з збереженим документом.

Діаграма послідовності додавання студента наведена на рисунку 2.3.

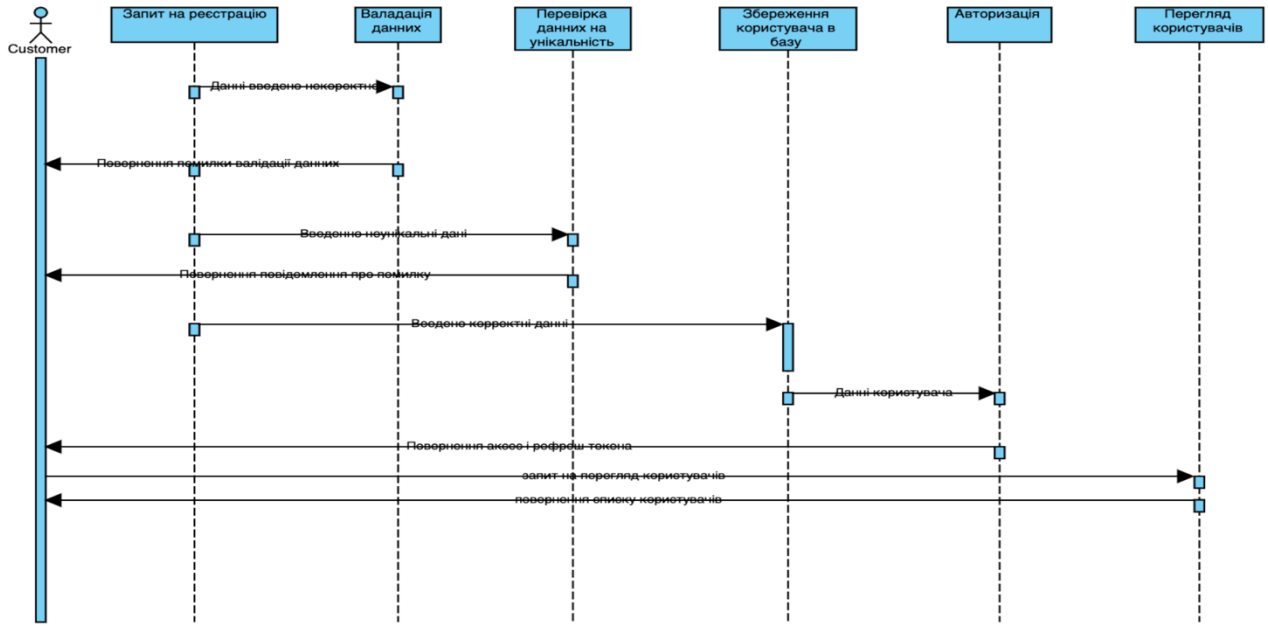


Рисунок 2.3 – Діаграма послідовності

Така ж послідовність дій застосовується при реєстрації користувача, при надсиланні заявок, відгуків. Розглянемо діаграму послідовності для реєстрації користувача. Не зареєстрований користувач, додає необхідні дані до тіла запиту.

При цьому модуль користувача виконує ряд перевірок для уникнення досить частих помилок:

– перевірка чи вільний email;

– перевірка чи немає невказаних полів;

– перевірка чи співпадають типи даних.

В перевірках враховуються всі нюанси. В разі, якщо перевірки пройдені, додається користувач в базі даних і відразу посилається відповідь з токеном, який необхідний для авторизації.

При роботі з відгуками, містами, країнами, сферами діяльності виконується та ж сама послідовність дій.

Послідовність опрацювання заявки має схожий вигляд і очікує реакції з боку іншого користувача

Діаграма станів – діаграма, що визначає зміну станів об'єкта у часі, одна з діаграм моделювання поведінки в UML. Адміністратор може керувати користувачами, містами, країнами, сферами діяльності, переглядати статистику та вносити заміни. Діаграму станів для адміністратора наведено на рисунку 2.4.

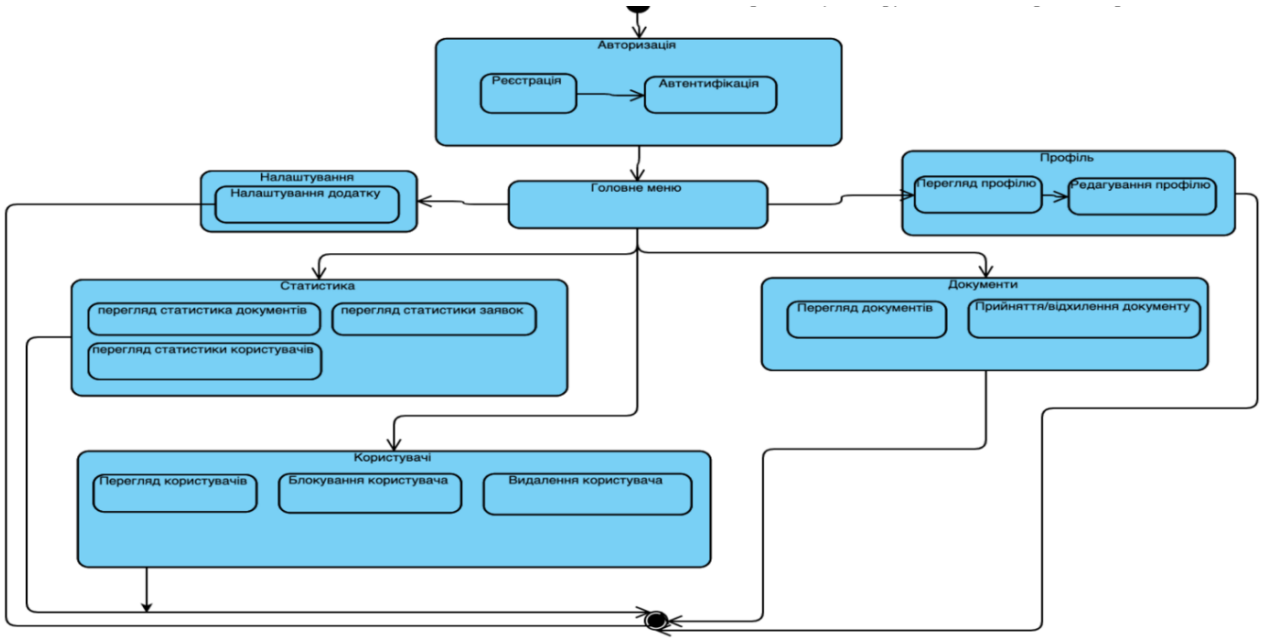


Рисунок 2.4 – Діаграма станів

Користувачі мають можливість взаємодіяти один з одним, оновлювати свій профіль, надсилати заявки, писати відгкуки, додавати документи. Діаграму діяльності для користувача та адміністратора наведено на рисунку 2.5.

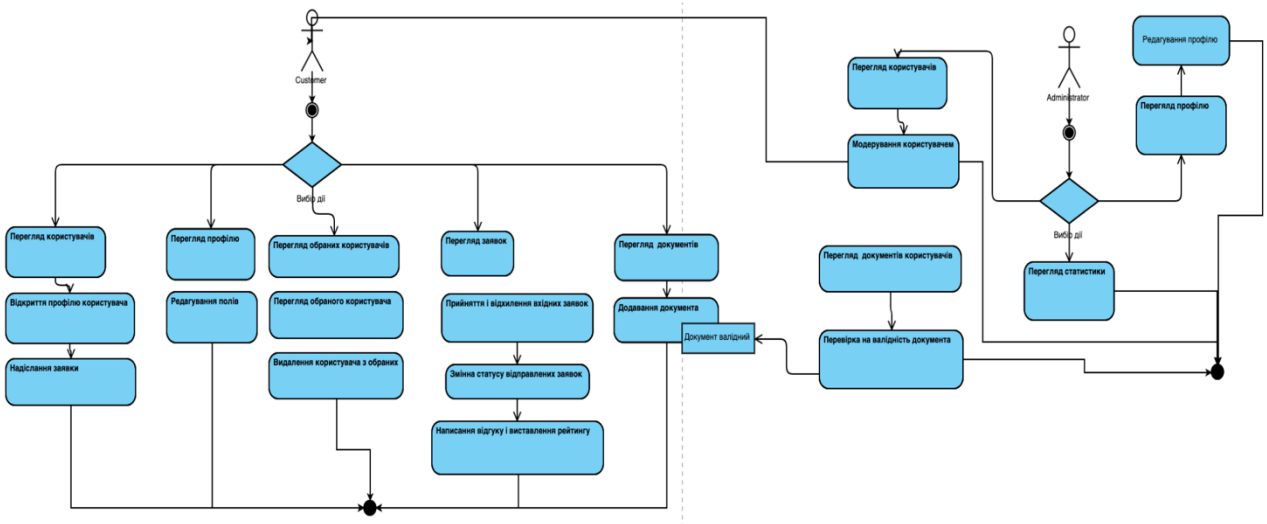


Рисунок 2.5 – Діаграма діяльності

2.3 Проектування інфологічної моделі бази даних

Для зберігання даних використаємо реляційну базу даних. Для проектування бази даних розглянемо детальніше вхідні дані. Визначимо основні структури даних, необхідні для опрацювання в програмі. Маємо такі об’єкти предметної області: користувач, адміністратор, документ, репорт, заявка, відгук, сфера діяльності, рейтинг.

Користувач характеризується іменем, прізвищем, номером телефону, адресою, має email та пароль, опис і ширину. Адміністратор містить тільки основні поля користувача.

Документ містить такі дані: назва, опис, статус, дата створення, дата оновлення та інші метадані.

Заявка містить дані про те, який користувач є замовником, який є виконавцем, дату, контакти виконавця, повідомлення і сферу діяльності, в якій здійснюється послуга.

Сфера діяльності має лише назву та ключ для пошуку.

Репорт містить інформацію про табличку і запис відповідно до якого створено репорт, користувач, повідомлення і дата створення.

Рейтинг формується на основі відгуків, відгук містить користувача якому написано відгук, та користувача який його написав, повідомлення і оцінка.

Нормалізація схеми бази даних – покроковий процес розбиття одного відношення (на практиці – таблиці) відповідно до алгоритму нормалізації на декілька відношень на базі функціональних залежностей.

Нормальна форма – властивість відношення в реляційній моделі даних, що характеризує його з точки зору надмірності, яка потенційно може призвести до логічно помилкових результатів вибірки або зміни даних. Нормальна форма визначається як сукупність вимог, яким має задовольняти відношення. Таким чином, схема реляційної бази даних переходить у першу, другу, третю і наступні нормальні форми, якщо їх вимагає логіка.

Перша нормальна форма утворює ґрунт для структурованої схеми бази даних. Кожна таблиця повинна мати основний ключ: мінімальний набір колонок, які ідентифікують запис.

Необхідно уникати повторень груп правильно визначаючи неключові атрибути. Кожен атрибут повинен мати лише одне конкретне значення, а не множину різних значень (атомарність).

Друга нормальна форма вимагає, аби дані, що зберігаються в таблицях із композитним ключем, не залежали лише від частини ключа. Схема бази даних повинна відповідати вимогам першої нормальної форми. Дані, що повторно з'являються в декількох рядках, виносяться в окремі таблиці.

Третя нормальна форма нормалізації вимагає, аби дані в таблиці залежали винятково від основного ключа.

Схема бази даних повинна відповідати всім вимогам другої нормальної форми. Будь-яке поле, що залежить від основного ключа та від будь-якого іншого поля, має виноситись в окрему таблицю.

Серед можливих зв’язків таблиць в реляційних базах даних є зв’язок один до одного, один до багатьох та багато до багатьох. Розглянемо представлення користувача в вигляді таблиці бази даних.

Інформація про користувача в базі даних наведена на рисунку 2.6.

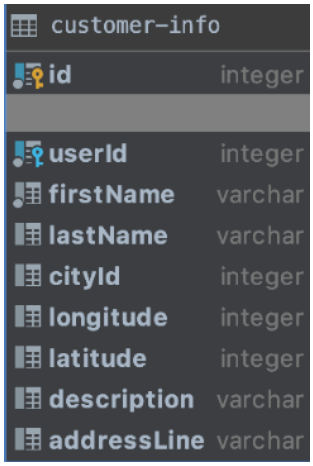


Рисунок 2.6 – Таблиця “Користувач”

До кожної таблиці додаватимемо унікальний ідентифікатор ID для відповідності першій нормальній формі. Документ відокремлюється в іншу таблицю, згідно другої нормальної форми. Один користувач може мати безліч документів, документ може мати лише одного користувача. Отже, це зв’язок один до багатьох. На рисунку 2.7 зображена таблиця користувач та документ, зв’язок між ними.

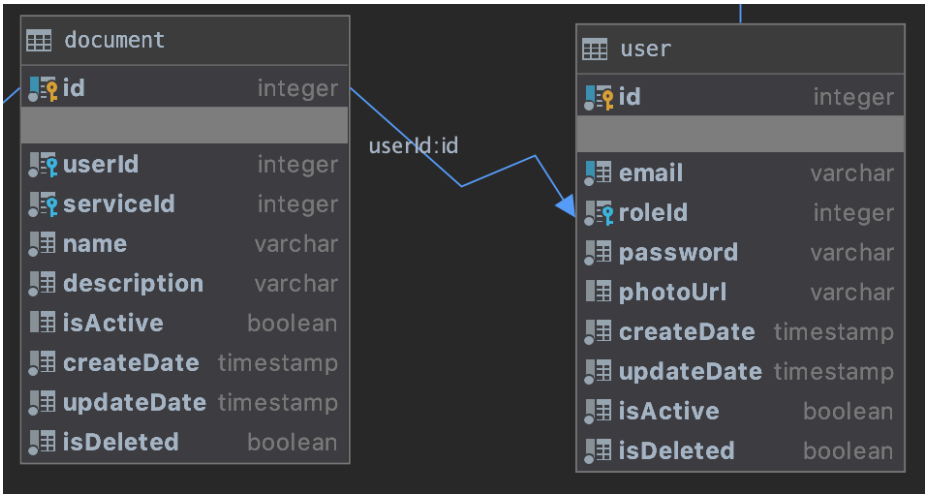


Рисунок 2.7 – Таблиці «користувач» та «документ»

Відомості про сфери діяльності утворюються за допомогою списку користувачів, які спеціалізуються в ній. Багато користувачів, може мати багато сервісів і навпаки. Зв’язок користувачів до сфер діяльності – багато до багатьох.

Тому введемо проміжну таблицю для забезпечення такого типу зв’язку. Зв’язок викладачів та мов до груп – один до багатьох.

На рисунку 2.8 зображені зв’язки між таблицями користувач, сфера діяльності, документ та користувач.

Також було досить гарним рішенням є винесення спільних даних користувача в таблиці користувачів і окремі дані адміністратора в іншу табличку. Тому отримаємо три таблички такі як: користувач, інформація користувача, роль користувача і ідентифікаційні токени користувача.

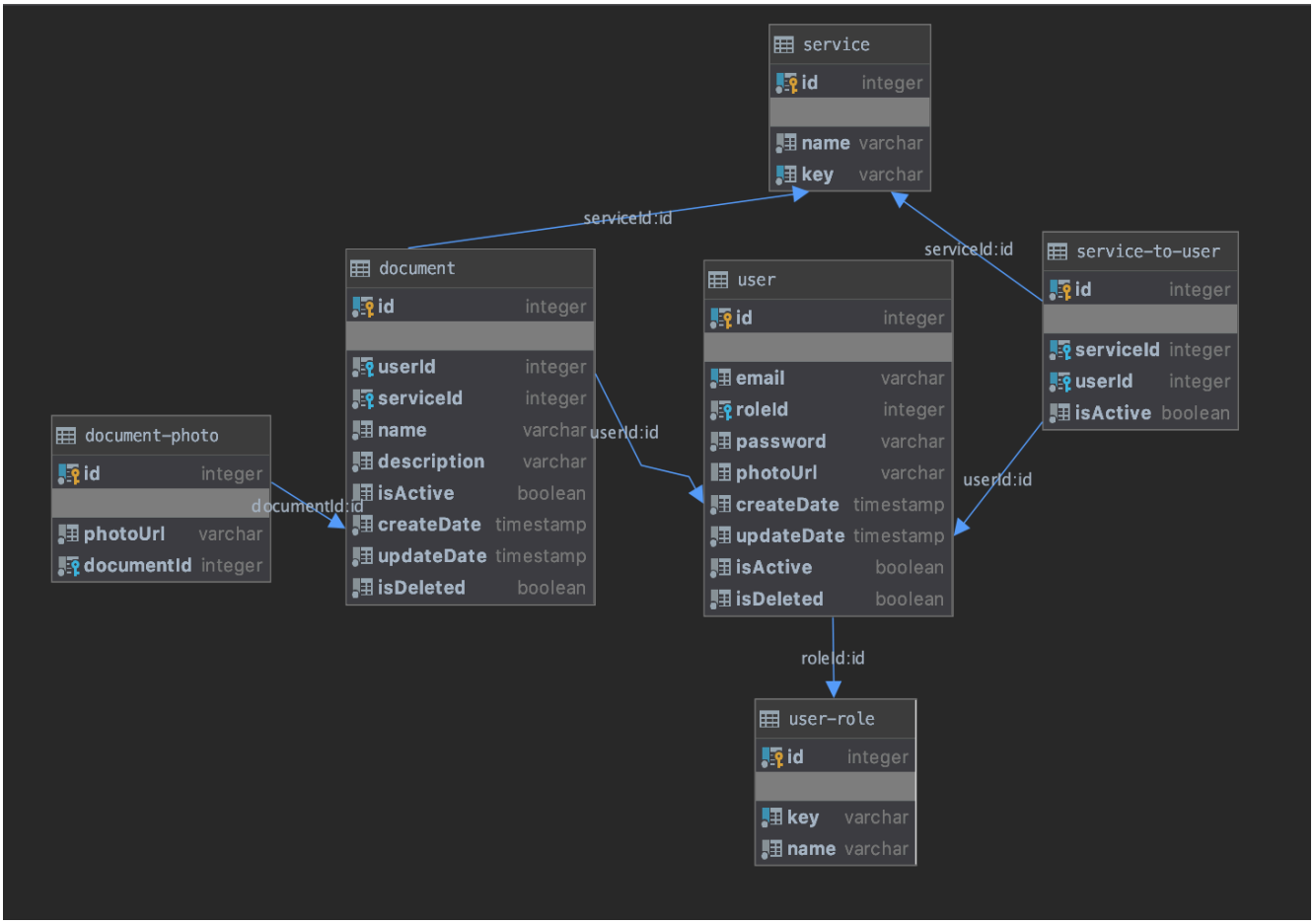


Рисунок 2.8 – Таблиці «користувач», «сфера діяльності», «документ»

Один користувач може мати лише одну інформацію користувача, так само як один користувач (адміністратор), може мати лише одну роль користувача, тому зв'язок між ними буде один до одного.

На рисунку 2.9 зображені зв’язки між таблицями користувач, інформація користувача, інформація адміністратору.

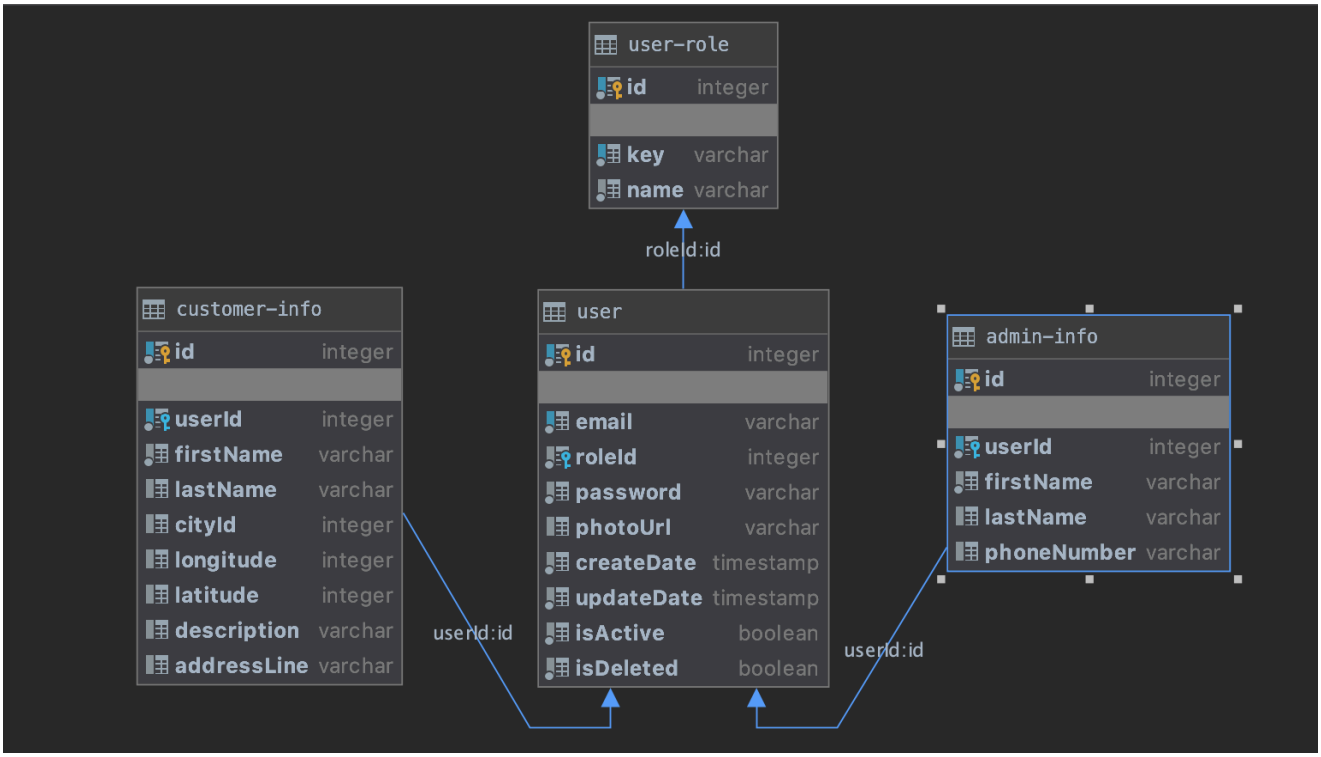


Рисунок 2.9 – Таблиці «користувач», «інформація користувача» і «інформація адміністратора» та «роль користувача»

Також майже кожна табличка має допоміжні таблички, для додаткових даних. Наприклад: табличка з посиланнями на фото документів, статут заявки, рейтинг користувача, рефреш-токен тощо. Всі таблички і повна схема бази даних зображено на рисунку 2.10.

Така структуризація даних відповідає всім нормальним формам реляційних баз даних та забезпечує відповідну нормалізацію.

Перевіримо, чи розроблені схеми відповідають основним нормальним формам. Кожна таблиця має основний ключ, атрибути є атомарними, повторення користувачів даних не виявлено, а отже спроектована база даних відповідає першій нормальній формі.

Друга нормальна форма вимагає першу нормальну форму та щоб дані, що зберігаються в таблицях із композитним ключем, не залежали лише від частини

ключа. Оскільки в структурі таблиць немає композитних ключів, база даних відповідає другій нормальній формі.

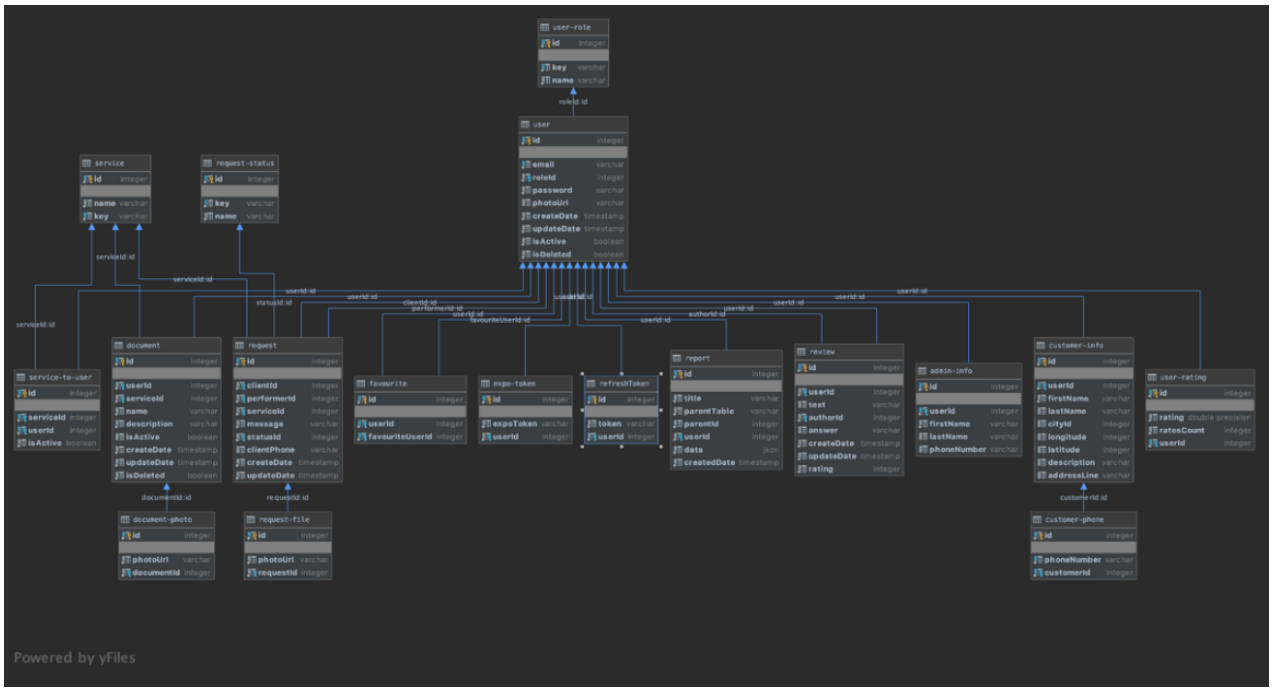


Рисунок 2.10 – Повна схема бази даних

Третя нормальна форма вимагає, аби дані в таблиці залежали винятково від основного ключа. Даних, що залежать від неключових атрибутів таблиць не виявлено – база даних нормалізована. Узагальнена інфологічна схема база даних наведена на рисунку 2.11.

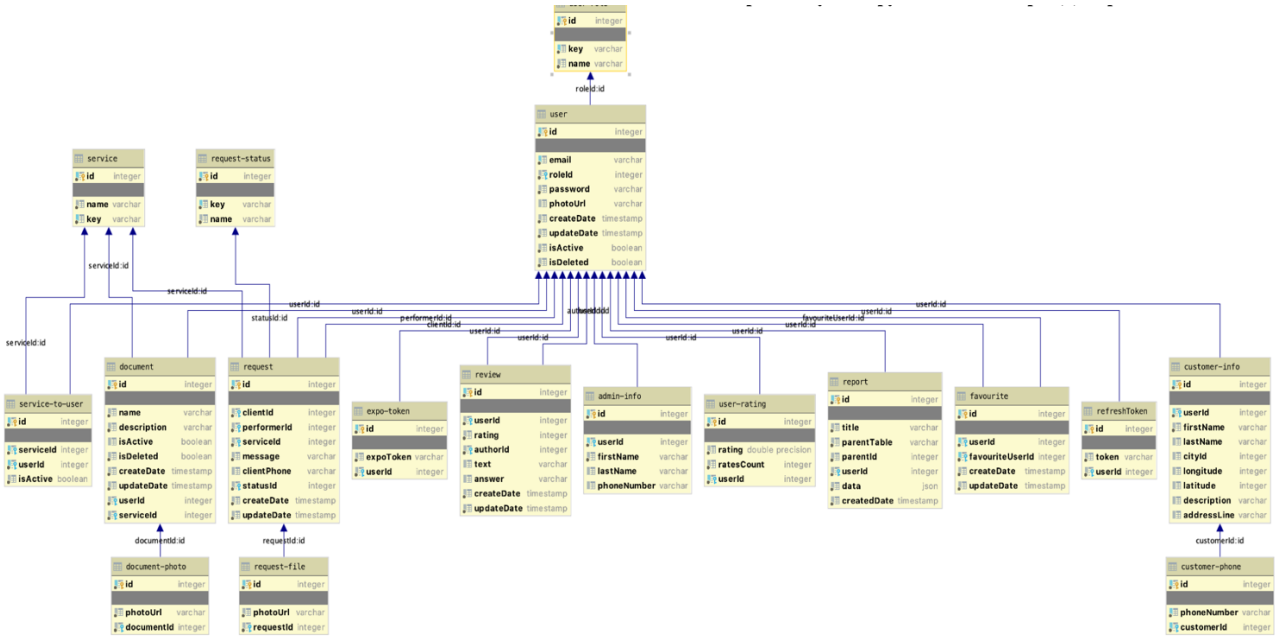


Рисунок 2.11 – Інфологічна модель бази даних

Така структура даних між таблицями з відповідними зв’язками буде перенесена в об’єктну модель проекту.

2.4 Проектування інтерфейсу клієнта

Будь який застосунок повинен мати зручний та інтуїтивний інтерфейс, щоб навіть недосвідчений користувач міг з легкістю працювати з програмним забезпеченням, розуміючи, як скористатись основними функціями програми Всі форми та поля повинні бути скомпонованими та правильно розміщеними.

Темою даної кваліфікаційної роботи була лише серверна частина, яка не включає в себе проектування та розробку графічного користувацького інтерфейсу. Тому в ході розробки кваліфікаційної роботи було створено OpenAPI документацію за допомогою бібліотеки Swagger, яка графічно зображує інтерфейс взаємодії з серверної частини, для будь-якого клієнту.

Для генерації документації було використано бібліотеку Swagger. Ця бібліотека надає безліч генераторів, якими можна промаркувати методи контролеру та визначити, що певний ендпойнт отримує, який тип даних повертає, які помилки може повертати тощо. Наприклад в нас є контроллер UsersRolesController, його ми промаркуємо декоратором ApiTags, щоб додати до нього короткий опис за допомогою тегу. В середині цього контролеру є метод get(), цей метод було позначено декоратором ApiResponse, який приємай параметри, які є характеристиками результату виклику цього ендпойнту, наприклад статус код має бути – 200, короткий опис, тип даних, який поверне ендпойнт та позначка чи цей тип буде в масиві. Також можуть бути декоратори щоб позначити характеристики баді чи параметрів але так, як цей метод не приймає ніякого інпуту, відповідно ці декоратори тут не було використано. Контролер можна розглянути на рисунку 2.12.

```
@ApiTags( tags: 'User roles')
@Controller( prefix: 'user-roles')
export class UsersRolesController {
  constructor(private readonly userRoleService: UsersRolesService) {
  }

  @ApiResponse( options: { status: 200, description: 'Return customer profile',
                type: UserRoleDto, isArray: true })
  @Get( path: '')
  async get(): Promise<UserRoleDto[]> {
    return await this.userRoleService.getRoles();
  }
}
```

Рисунок 2.12 – Контроллер «UsersRolesController» з декораторами Swagger

В прикладі вище ми мали тип, який повертає метод, цей тип називався UserRoleDto, для того, щоб відобразити всі атрибути цього типу, їх також потрібно описати декораторами бібліотеки Swagger. Кожен атрибут класу промарковано декоратором ApiProperty, який дозволяє потім візуалізувати це поле в документації. Переглянути UserRoleDto можна на рисунку 2.13.

```
import { ApiProperty } from '@nestjs/swagger';

export class UserRoleDto {
  @ApiProperty()
  id: number;

  @ApiProperty()
  key: string;

  @ApiProperty()
  name: string;
}
```

Рисунок 2.13 – Тип даних UserRoleDto з декораторами Swagger

У цьому прикладі для кожного атрибуту (id, name, role) додано декоратор ApiProperty(), який вказує бібліотеці Swagger, що ці атрибути повинні бути відображені у документації. Можна також додати додаткові параметри до декоратора ApiProperty, щоб уточнити деталі відображення атрибутів.

Таким чином потрібно описати кожен контролер, метод і типи вхідних і вихідних даних. Після цього додати конфігурцію бібліотеки в основний файл серверу. Конфігурацію зобожено на рисунку 2.14.

```
async function bootstrap() {
  const appOptions = { cors: true, bodyParser: false };
  const app = await NestFactory.create(AppModule, appOptions);

  app.useGlobalPipes(new ValidationPipe());

  app.use(bodyParser.json( options: { limit: '10mb' }));
  app.use(bodyParser.urlencoded( options: { limit: '20mb', extended: true }));

  const options = new DocumentBuilder()
    .addBearerAuth()
    .setTitle('LiveServices')
    .setDescription('The LiveServices API description')
    .setVersion('1.0')
    .build();
  const document = SwaggerModule.createDocument(app, options);
  SwaggerModule.setup( path: 'documentation', app, document);

  await app.listen( port: 3000);

  if (module.hot) {
    module.hot.accept();
    module.hot.dispose(() => app.close());
  }
}

bootstrap();
```

Рисунок 2.14 – Конфігурація серверу разом з конфігурацією бібліотеки Swagger

В результаті за шляхом http://localhost:3000/documentation/#/, в браузері можна відкрити вже готову згенеровану документацію. Основна сторінка відображена на рисунку 2.15.

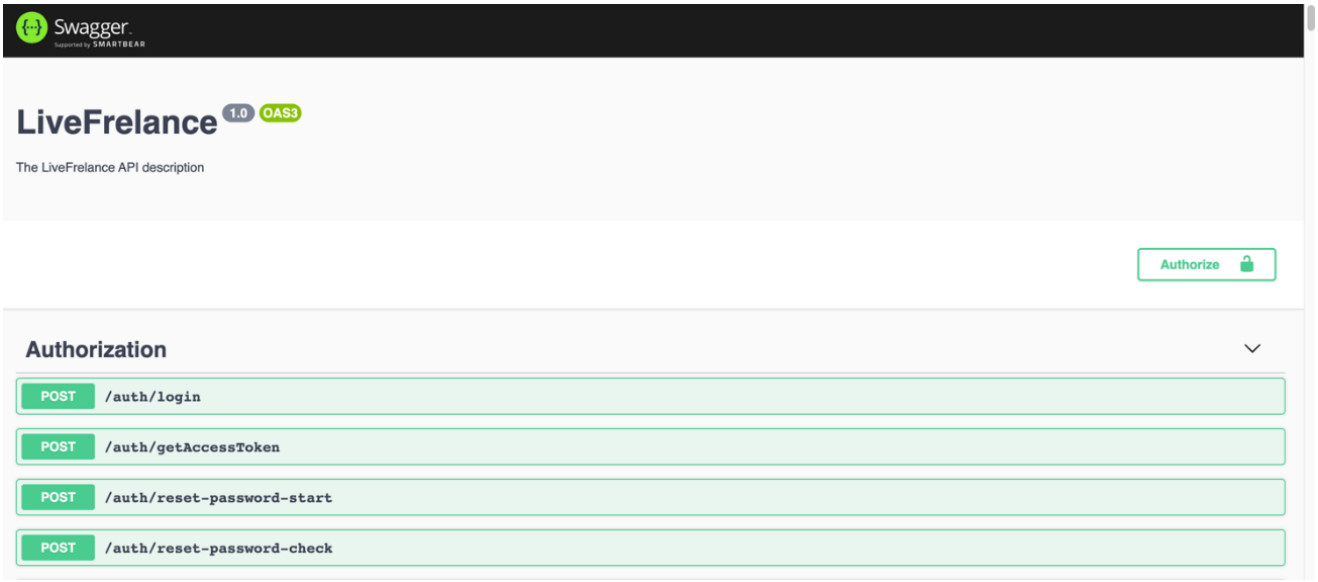


Рисунок 2.15 – На рисунку зображено головну сторінку згенерованої документації

Кожний контролер відображається в окремій секції і ця секція містить набір методів, при відкритті метода, можна побачити структуру даних яку він приймає, параметри і заголовки які потрібні, статус коди відповіді і пояснення до них, структура вихідних даних і можливі помилки і опис до них. Приклад розгорнутого методу можна побачити на рисунку 2.16.

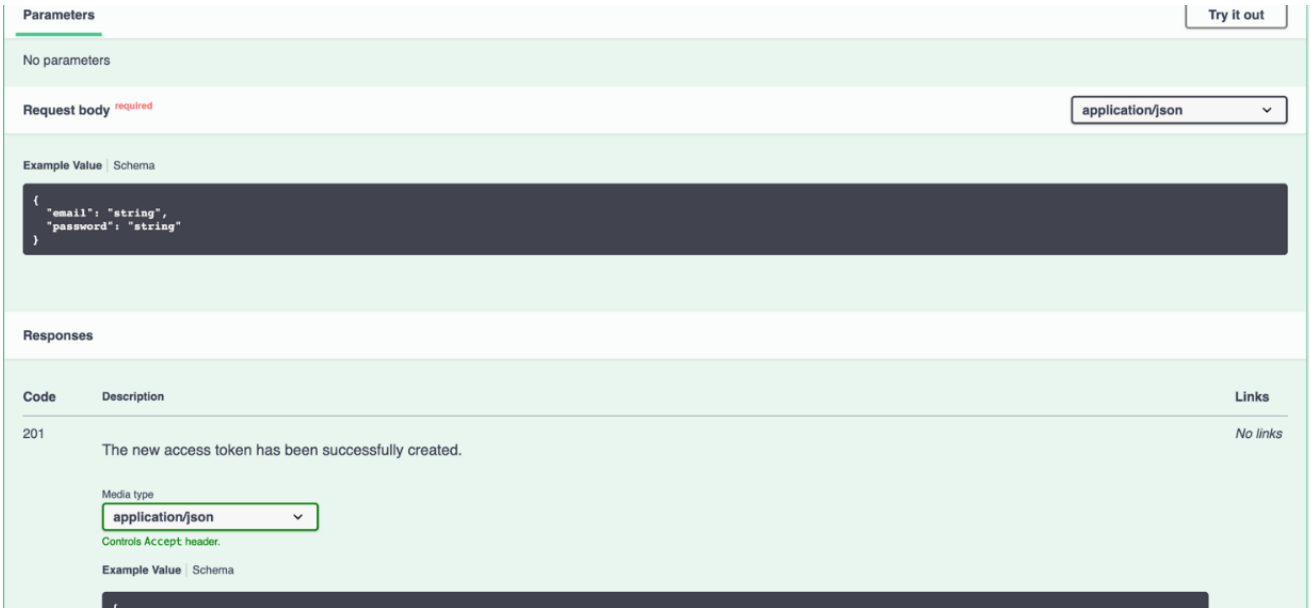


Рисунок 2.16 – Розгорнутий метод login

Перший модуль Authorization зображено на рисунку 2.17. На ньому відображено всі основні методи цього модуля.

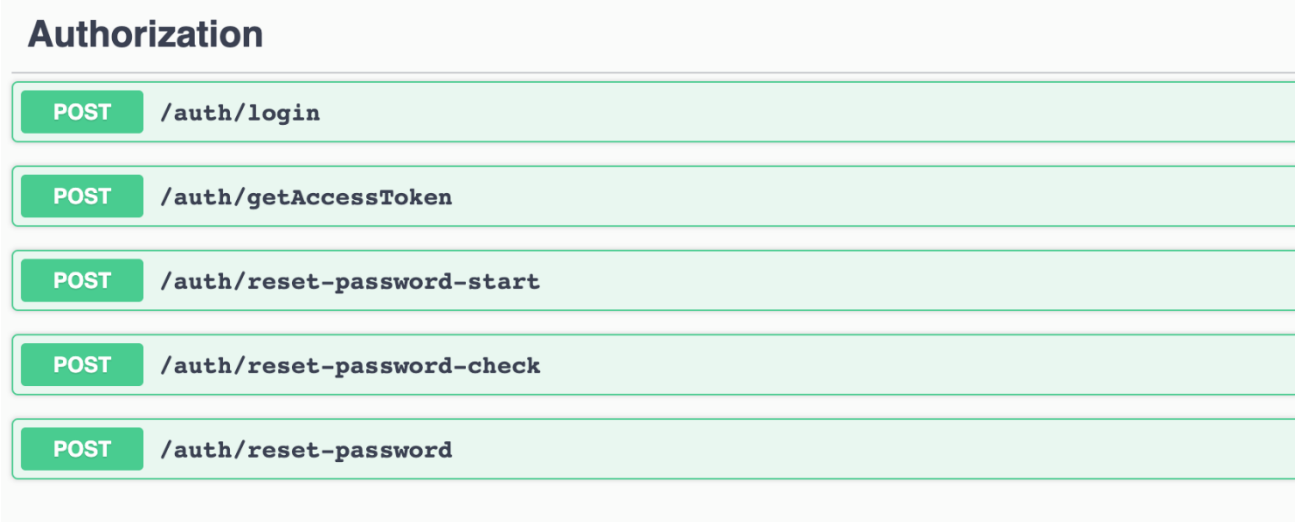


Рисунок 2.17 – Документація модуль Authorization

На рисунку 2.18 зображено модуль CustomerAccount з усіма його методами та атрибутами.

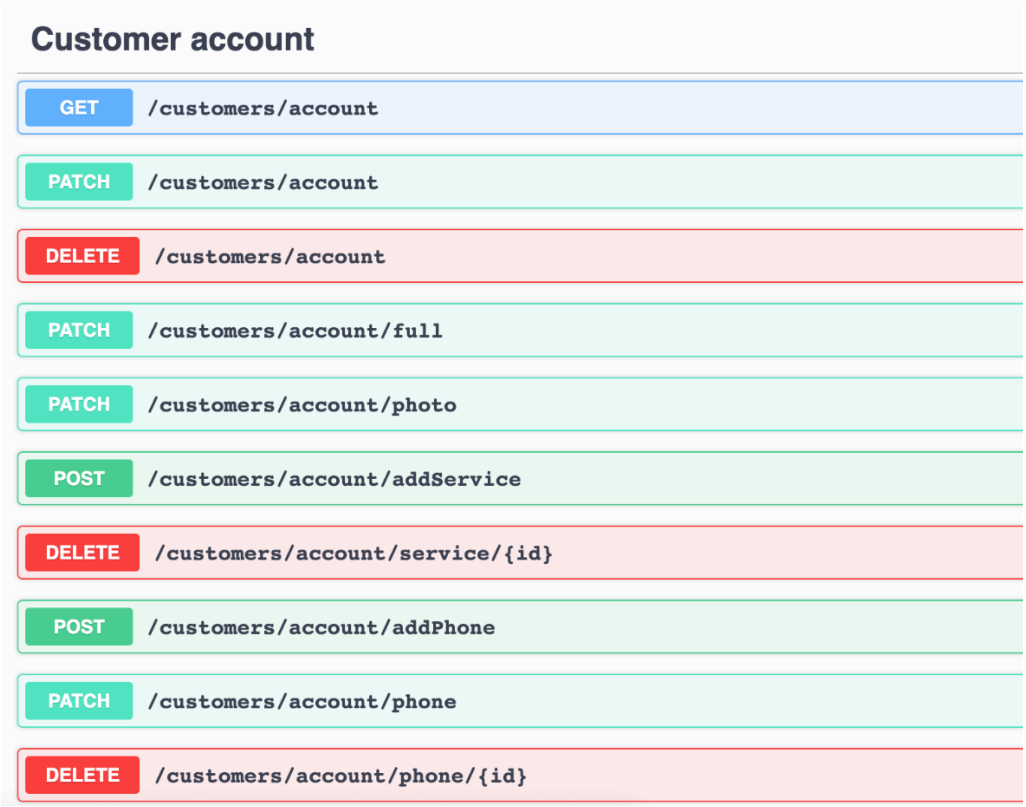


Рисунок 2.18 – Документація модуля CustomerAccount

В кінці документації в нас знаходяться описи структур різних типів даних, які використовуються в методах. Приклад перших двох типів даних можна побачити на рисунку 2.19.

Рисунок 2.19 – Опис документації для двох типів даних – LoginDto I TokenResponseDto

Отже, в ході побудови документації, було розгорнуто та описано всі деталі кожного модуля системи і кожного ендпойнту, які включають в себе всі нюанси та деталі їх використання.

2.5 Аналіз та вибір технологій і методів реалізації серверної частини

Розробка веб-застосунків ведеться на багатьох мовах програмування.

Java – чудова мова для написання серверної частини. У всякому разі, майже

вся освітня документація, всі інтернет-курси засновані на ній. А ще це найпопулярніша мова за оцінкою TIOBE, друга за кількістю початкових кодів на GitHub, та й взагалі велика красива та доступна мова. Саме тому вивчення цієї мови програмування має бути першочерговим завданням для будь-якого бекенд веб-розробника, який орієнтований на розробку серверної частини.

C# – чудова мова, що увібрала в себе все краще від Java, при цьому врахувавши і виправивши багато недоліків. Що стосується розробки веб-додатків, то тут до ваших послуг одні з найбільш функціональних середовищ Visual Studio. А ще знання C # стане для вас приємним бонусом, коли доберетеся до використання Unity 3D. З таким набором можливості будуть безмежні.

Python – мова широко використовується для написання. Любителі зміїної мови розробили безліч інструментів, що дозволяють скомпілювати код на Python в потрібний стан. Найпопулярнішим фреймворком є Django, який без праці допоможе вам створити застосунок на чистому Python. А якщо ні, то допоможуть добрі розробники в чаті.

PHP – сама по собі відмінна мова. Є одглю з найпопулярніших мов програмування протягом останніх років. З його ви не будете відчувати жодних проблем в розробці веб-додатків. Попит на фахівців які володіють цією мовою програмування досить високий.

Lua – мова, яка старше Java, куди менш популярна, але все одно затребувана. У неї є ряд переваг, на кшталт динамічної типізації, щодо простого синтаксису, але до наших днів вона дожила завдяки задіянню в іграх. Саме зручність створення програмного прошарку між движком і оболонкою відкрило перед Lua двері в світ кишенькових гаджетів.

Node.js – платформа з відкритим кодом для виконання високопродуктивних мережевих застосунків, написаних мовою JavaScript / TypeScript. Засновником платформи є Раян Дал (Ryan Dahl). Якщо раніше JavaScript/TypeScript застосовувався для обробки даних в браузері користувача, то node.js надав можливість виконувати JavaScript-скрипти на сервері та відправляти користувачеві результат їх виконання.

Платформа Node.js перетворила JavaScript/TypeScript на мову загального використання з великою спільнотою розробників.

Node.js має наступні властивості:

– асинхронна одно-нитева модель виконання запитів;

– неблокуючий ввід/вивід;

– система модулів CommonJS;

– рушій JavaScript Google V8.

– для керування модулями використовується пакетний менеджер npm (node package manager).

NestJS – це той фреймворк, який створений для полегшення життя розробника, який використовує правильні архітектурні підходи і диктує свої правила проектування та розробки.

Тому, NestJS – це не тільки фреімворк для бекенд, але і можливість увійти в світ передових концепції, наприклад таких як DDD, Event sourcing і мікросервісной архітектурі. Все упаковано в простій і легкій формі, так що вибір за вами – вирішуєте ви використовувати всю платформу або просто використовувати її компоненти.

Виходячи з потреб додатків та моєї спеціалізації, було обрано платформу NodeJs та фреймворк NestJs та мову TypeScript, оскільки обраний фреймворк використовує мову програмування TypeScript.

Для роботи додатку не обійтись без бази даних. Всі дані, які є в додатку, потрібно десь зберігати, і база даних є оптимальним рішенням. На сьогоднішній день існує два головних види баз даних: реляційні і нереляційні.

Переваги реляційних баз даних:

– простота і доступність для розуміння користувачем; єдиною використовуваною інформаційною конструкцією є таблиця;

– суворі правила проектування, які базуються на математичному апараті;

– повна незалежність даних; зміни в прикладній програмі при зміні реляційної бази даних мінімальні;

– для організації запитів і написання прикладного програмного забезпечення немає необхідності знати конкретну організацію бази даних у зовнішній пам'яті.

Недоліки реляційної моделі бази даних:

– далеко не завжди предметна область може бути представлена у вигляді таблиць;

– в результаті логічного проектування з'являється множина «таблиць»; це призводить до труднощів розуміння структури даних;

– база даних займає відносно багато зовнішньої пам'яті;

– відносно низька швидкість доступу до даних.

Переваги нереляційних баз даних:

– відсутність SQL;

– простота роботи;

– простіший синтаксис запитів;

– JSON.

Недоліки нереляційних баз даних:

– програма сильно прив’язується до конкретної СУБД;

– обмежена ємність вбудованої мови запитів;

– низька цінність і вузькопрофільність знань.

Виходячи з переваг і недоліків, було обрано реляційну базу даних та систему управління базами даних PostgreSQL.

Вибір середовища розробки є також важливим моментом у розробці програмного забезпеченя. На даний момент існує 3 найбільш популярні середовища розробки під Java. У кожної з цих IDE є свої особливості. Тому щоб вибрати найбільш відповідне рішення, слід розглянути кожну IDE більш докладно та проаналізувати їхні переваги та недоліки.

Eclipse – це безкоштовне середовище розробки від некомерційної організації Eclipse Foundation. По суті справи, сама програма - це основа, до якої підключаються різні модулі. Наприклад, Java Development Tools (для створення додатків на Java), C / C ++ Development Tools (для розробки програм на мові C

або C ++) і так далі. Завдяки активному розвитку, а також підтримки з боку компанії і сторонніх розробників, на даний момент у цій IDE є наступні переваги:

– офіційна русифікація інтерфейсу і документації;

– відмінна продуктивність на слабких машинах;

– велике число доповнень (для роботи з сервером, базою даних і т. д.);

– можливість підключення модулів (про це було сказано вище);

– можливість групової розробки.

Eclipse була дуже популярна кілька років тому і вважалася монополістом на ринку IDE для Android. Однак у зв'язку з виходом Android Studio, в 2014 р Google перестала підтримувати Eclipse як основне середовище для розробки додатків під Android.

IntelliJ IDEA – розробляється компанією JetBrains. Як і Eclipse, це середовище розробки дає можливість створювати програми на декількох мовах програмування. Крім цього, середовище має потужний двигун і величезні можливості. Якщо розглядати програмування під Java між IntelliJ IDEA і Eclipse, то перший варіант кращий. У цього середовища є незаперечні переваги щодо свого попереднього конкурента:

– більш швидке налагодження значень;

– автозаповнення методів (також реалізовано в Eclipse, але поки в тестовому варіанті);

– наявність рефакторінга (автоматичного підбору значень);

– більш зручний інтерфейс;

– дуже добре підходить для програмування на Java.

Головний недолік - це наявність платної версії.

WebStorm – так само компанією JetBrains. В данному середовищі є всі необхідні інструменти для розробки веб-додатків на будь якій веб-мові. Головним конкурентом цього середовища є Visual Studio Code. Основні переваги WebStorm IDE відносно інших середовищ:

– наявність зручного терміналу;

– можливість встановлення плагінів;

– система контрою версміями;

– красивий інтерфейс;

– автопідставнока та налагодження всіх зв’язків в проекті.

– головний недолік - це платна підписка.

Visual Studio Code – розробляється компанією Microsoft. Є досить непоганим рішенням для веб-розробки. Є універсальим середовищем в якому можна розробляти на будь-які мові, достатньо лише встановити розширення. А також наявна можливість встановлення багатьох розширень, присутні низькі системні вимоги, є безкоштовне і має зручний інтерфейс.

Отже, використовувати Eclipse краще, коли приватний комп’ютер не володіє хорошою потужністю (наприклад, має всього 1 ГБ оперативної пам'яті), програми будуть створюватися на декількох мовах програмування, а майбутній розробник абсолютно незнайомий з англійською мовою.

IntelliJ IDEA відмінно підійде, якщо розробка ведеться на декількох мовах програмування, а комп'ютер досить потужний (мінімум 2ГБ оперативної пам'яті).

WebStorm слід використовувати якщо приватний комп’ютер має достатню потужність (мінімум 2 ГБ оперативної пам'яті), буде здійснюватись веб розробка і є кошти для придбання данного програмного продукту.

VisualCode слід використовувати якщо приватний комп’ютер має слабку потужність (мінімум 1 ГБ оперативної пам'яті), буде здійснюватись веб розробка і потрібне просте та швидке середовище для розробки.

Виходячи з потреб для даної кваліфікаційної роботи, було обрано WebStorm IDE, так як це найзручніше та найефективніше середовище розробки додатків на платформі NodeJs, яке містить безліч функцій та можливостей.

2.6 Висновки до розділу 2

Висновок. Отже, у цьому розділі було проведено аналіз існуючих архітектурних рішень та обрано оптимальне рішення для створення ПЗ. Визначено основні компоненти, послідовності дій, алгоритми та стани

програмної системи. На основі вхідних даних було спроектовано та нормалізовано структуру бази даних, складено інфологічну схему. З метою досягнення поставленої мети було виконано детальне вивчення різних аспектів архітектури та виявлено найбільш підходящу структуру для потреб проекту.

На основі проведеного аналізу були ідентифіковані основні компоненти, які будуть використовуватися в програмній системі. Ці компоненти були ретельно розглянуті і детально описані, включаючи їх функції та взаємодію між ними. При визначенні послідовності дій було враховано логіку роботи системи та взаємозв'язки між компонентами.

Для забезпечення ефективності та правильності роботи програмної системи, були розроблені відповідні алгоритми. Ці алгоритми визначають логіку обробки даних, взаємодію з компонентами та забезпечують досягнення поставлених цілей. Крім того, були визначені можливі стани програмної системи та відповідні дії, що пов'язані з перехідами між цими станами.

Окрім архітектурних аспектів, в роботі було зосереджено увагу на проектуванні та нормалізації структури бази даних. З урахуванням вхідних даних та вимог до системи, була розроблена оптимальна структура бази даних, яка відповідає потребам проекту. Для зручності подальшого розроблення та роботи з даними була складена інфологічна схема, що уявляє собою детальне відображення зв'язків та взаємозв'язків між сутностями бази даних.

У результаті проведеного аналізу, визначення компонентів, алгоритмів, станів програмної системи та проектування структури бази даних були створені основні основи для подальшої розробки ПЗ, які будуть використовуватися для досягнення мети проекту та задоволення вимог користувачів.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ

3.1 Базова конфігурація збірки проекту

Для розробки програми було використано мову програмування TypeScript, платформу NodeJs та фреймоврк NestJS.

В якості реляційної бази даних обрано PostreSQL, та ORM TypeORM для відображення таблиць бази даних в об’єктну модель.

TypeORM дозволяє будувати схему бази даних на основі заданих в коді моделей та їх зв’язків.

Створимо базовий проект на NestJs за допомогою попередньо встановленої nest-cli, використовуючи відповідні команди.

Для збірки проекта використано npm, запускаємо команду npm i -g @nestjs/cli, після чого запустимо генератор проекту командою nest new project-name, також встановимо додаткові пакети командою bnpm i --save @nestjs/core @nestjs/common rxjs reflect-metadata nodemoon @types/node ts-node ts-lint, es-lint та інші, серед яких:

- NestJs core ядро фреймворку;
- NestJs common додаткова бібліотека фреймоврку;
- Rxjs додаткова бібліотке для зрозробки;
- reflect-metadata додаткова бібліоткека;
- nodemoon бібліотека для перезапуску проекту після змін;
- @types/node типи даних node;
- ts-node біліотека для запуску TypeScript;
- ts-lint бібліотека для підтримки чистоти коду.

Зазначені пакети є лише прикладами, і список може бути розширений залежно від потреб вашого проекту. Після встановлення пакетів ви можете використовувати їх у вашому проекті, додавши їх у файл package.json та імпортуючи їх у вашому коді.

На рисунку 3.1 зображений вид файлу package.json.

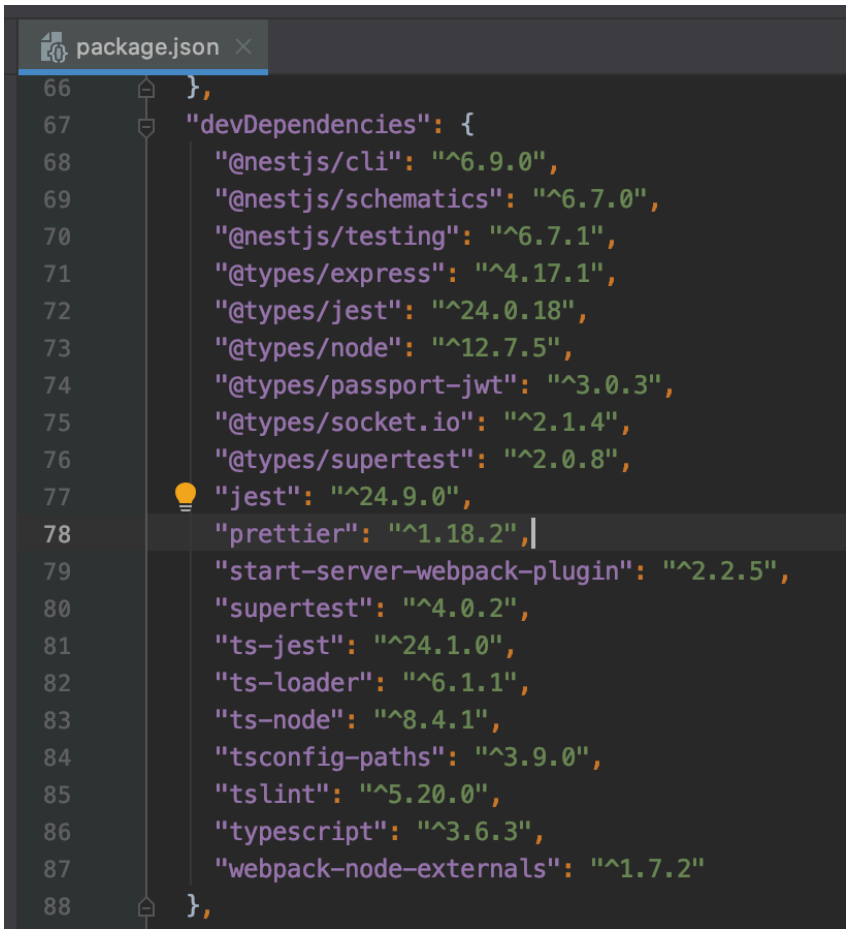


Рисунок 3.1 – Файл package.json

Відкриємо згенерований проект та додамо ще декілька залежностей через npm. Package.json необхідний для збірки проекту в npm.

Запустимо збірку проекту командою npm run build.

Запуск проекту на сервері відбувається через Docker. Діаграму розгортання наведено на рисунку 3.2.

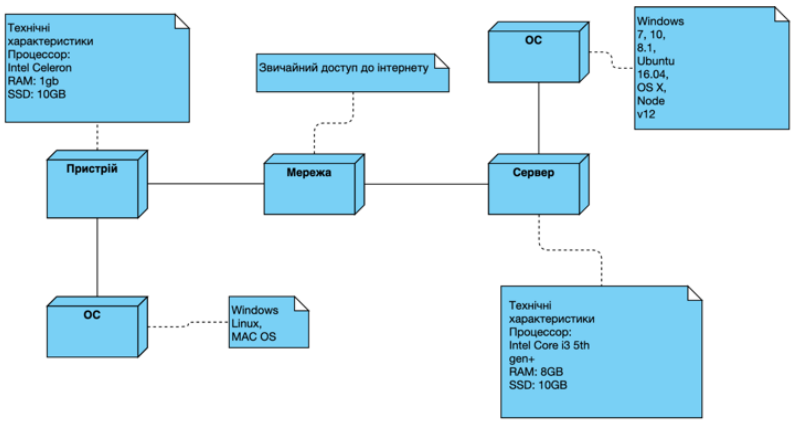


Рисунок 3.2 – Діаграма розгортання

Підключення до бази даних

Створимо пусту базу даних livefreelance в СУБД PostgreSQL:

```
CREATE DATABASE livefreelance;
```

Налаштуємо підключення програми до створеної бази даних. Для цього створимо файл connection.module.ts в src/shared/database/connect. Цей файл отримує конфігураційні дані з development.env файлу і встановляє зв'язок з базою даних через TypeORM. Для роботи з базою даних необхідно налаштувати джерело даних (datasource), порт, користувача і пароль, тип бази, в нашому випадку PostgreSQL.

Додамо такі рядки в development.env:

```
DB_TYPE=postgres
#DB_HOST=liveservices-postgres
DB_HOST=localhost
DB_PORT=5432
DB_USERNAME=postgres
DB_PASSWORD=postgres
DB_SYNC=true
```

В головному каталозі. програми ствоимо main.ts, який буде вхідною точкою входу та основним файлом для NodeJs:

```
async function bootstrap() {
  const appOptions = { cors: true, bodyParser: false };
  const app = await NestFactory.create(AppModule, appOptions);

  app.useGlobalPipes(new ValidationPipe());

  app.use(bodyParser.json({ limit: '10mb' }));
  app.use(bodyParser.urlencoded({ limit: '20mb', extended: true }));

  const options = new DocumentBuilder()
    .addBearerAuth()
    .setTitle('LiveServices')
    .setDescription('The LiveServices API description')
    .setVersion('1.0')
    .build();
  const document = SwaggerModule.createDocument(app, options);
  SwaggerModule.setup('documentation', app, document);

  await app.listen(3000);

  if (module.hot) {
    module.hot.accept();
```

```
    module.hot.dispose(() => app.close());
    module.hot.accept();
    module.hot.dispose(() => app.close());
          const document = SwaggerModule.createDocument(app, options);
  SwaggerModule.setup('documentation', app, document);


  }

  const seeder = app.get(SeederService);
  await seeder.seed();
}

bootstrap();
```

Запустимо серверну частину через WebStorm і переконаємось, що немає помилок підключення до бази даних. На рисунку 3.3 зображено логи першого запуску програми, де можна побачити успішну ініціалізацію Nest.js.

```
/usr/local/bin/node /Users/also/Desktop/work/api/node_modules/ts-node/dist/bin.js /Users/also/Desktop/work/api/src/main.ts
[Nest] 1589   - 06/08/2020, 2:59:50 PM   [NestFactory] Starting Nest application...
[Nest] 1589   - 06/08/2020, 2:59:50 PM   [InstanceLoader] ConfigModule dependencies initialized +67ms
[Nest] 1589   - 06/08/2020, 2:59:50 PM   [InstanceLoader] PassportModule dependencies initialized +1ms
[Nest] 1589   - 06/08/2020, 2:59:50 PM   [InstanceLoader] HelpersModule dependencies initialized +0ms
[Nest] 1589   - 06/08/2020, 2:59:50 PM   [InstanceLoader] ConnectModule dependencies initialized +0ms
[Nest] 1589   - 06/08/2020, 2:59:50 PM   [InstanceLoader] TypeOrmModule dependencies initialized +1ms
[Nest] 1589   - 06/08/2020, 2:59:50 PM   [InstanceLoader] AppModule dependencies initialized +58ms
```

Рисунок 3.3 – Логи запущеного вперше проекту

TypeORM автоматично створить таблиці та зв’язки між ними на основі розроблених моделей.

3.2 Моделі

Створимо моделі, які будуть відображати таблиці бази даних та їх зв’язки. Модель – це простий TypeScript клас, який містить поля, що відповідають стовпцям таблиці в базі даних. NestJs пропонує набір декораторів для зв’язування об’єктної моделі з схемою бази даних.

Декоратори є інструментом декларативного програмування, вони дозволяють додати до класів і їх членам метадані і тим самим змінити їх поведінку без зміни їх коду.

Декоратори представляють функції, які можуть застосовуватися до класів, методам, методом доступу (геттерів і сеттерів), властивостям, параметрам.

Створимо клас User з відповідними полями. Над оголошенням класу вкажемо, що цей клас відноситься до моделі БД декоратором @Entity. Над кожним полем поставимо декоратор @Column та вкажемо назву відповідного стовпця в БД. Над ключовим полем ID поставимо декортаор @PrimaryGeneratedColumn() для автоматичної генерації ідентифікаторів.

```
@PrimaryGeneratedColumn()
@Entity('user')
export class User extends BaseEntity {
  @PrimaryGeneratedColumn()
  id: number;
  @Column({ unique: true })
  email: string;
  @Column()
  roleId: number;
  @ManyToOne(type => UserRole, role => role.users,
    { cascade: ['insert', 'update'], onUpdate: 'CASCADE', onDelete: 'RESTRICT' })
  @JoinColumn({ name: 'roleId' })
  role?: UserRole;
  @Column()
  password: string;
  @Column({ nullable: true })
  photoUrl: string;
  @CreateDateColumn({ type: 'timestamp', default: () => 'LOCALTIMESTAMP' })
  createDate?: string;
  @UpdateDateColumn({ type: 'timestamp', default: () => 'LOCALTIMESTAMP' })
  updateDate?: string;
  @Column({ default: true })
  isActive?: boolean;
  @Column({ default: false })
  isDeleted?: boolean = false;
  @OneToOne(type => CustomerInfo, info => info.user)
  customerInfo?: CustomerInfo;
  @OneToOne(type => AdminInfo, info => info.user)
  adminInfo?: AdminInfo;
  @OneToMany(type => RefreshToken, refreshToken => refreshToken.user)
  refreshTokens: Promise<RefreshToken[]>;
  @OneToMany(type => Document, document => document.user)
  public documents: Promise<Document[]>;
  @OneToMany(type => ServiceToUser, serviceToUser => serviceToUser.user)
  public serviceToUsers: ServiceToUser[];
  @OneToMany(type => Request, request => request.client)
  public requestClients: Request[];
  @OneToMany(type => Request, request => request.performer)
  public requestPerformers!: Request[];
  @OneToOne(type => UserRating, rating => rating.user)
```

```
  rating?: UserRating;
  @OneToMany(type => Report, report => report.user)
  public reports: Report[];
  @OneToMany(type => Review, review => review.user)
  public reviews: Review[];
  @OneToMany(type => Review, review => review.author)
  public writtenReviews: Review[];


  @OneToMany(type => Favourite, favourite => favourite.user)
  public favourites: Favourite[];
  @OneToMany(type => Favourite, favourite => favourite.favouriteUser)
  public favouritesUsers: Favourite[];
  @OneToMany(type => ExpoToken, expoToken => expoToken.user)
```

Хорошим тоном програмування є додавання в кожну Entity полів createDateTime та updateDateTime, які будуть містити точний час створення та оновлення моделі. Генерація значень забезпечується декораторами

```
@CreateDateColumn({ type: 'timestamp', default: () => 'LOCALTIMESTAMP' })
@CreateUpdatedDateColumn({ type: 'timestamp', default: () => 'LOCALTIMESTAMP' })
@CreateCreatedDateColumn({ type: 'timestamp', default: () => 'LOCALTIMESTAMP' })
```

Окрім звичайних полів, що відповідають стовпцям БД, маємо також пов’язуючі поля, наприклад documents, reviews, reports. Насправді таких стовпців в таблиці немає. Це можливість TypeORM автоматично зв’язувати об’єкти один з одним у відповідності з відношеннями БД. Розглянемо типовий при проектуванні зв’язок - один до багатьох:

```
@OneToMany(type => Request, request => request.performer)
@Reference(otherSide =>Request)
public requestPerformers!: Request[];
```

За допомогою декораторів вказуємо тип зв’язку. Параметр type вказує на назву змінної в класі Request, яка пов’язує User та Request. В класі Request має бути визначена зміна user з оберненою анотацією @ManyToOne.

```
@ManyToOne(type => User, user => user.requestClients,
{ cascade: true, onDelete: 'CASCADE',
onUpdate: 'CASCADE' })
@JoinColumn({ name: 'clientId' })
@Reference(otherSide => User)
client: User;
```

@JoinColumn вказує на зовнішній ключ в таблиці Request. Таким чином відбувається зв’язування об’єктів TypeScript за допомогою TypeOrm у відповідності до відношення один до багатьох.

Зв'язки один до одного та багато до одного створюються аналогічним до попереднього прикладу образом.

Розглянемо відношення багато до багатьох на прикладі сфер діяльності і користувачів. Один користувач може мати багато сфер діяльності, а одна сфера діяльності може бути у багатьох користувачів. Для цього створимо проміжну табличку Service-to-user. Зі сторони студента визначаємо таке поле:

```
@OneToMany(type => ServiceToUser,
serviceToUser => serviceToUser.user)
@Reference(otherSide => ServiceToUser)
public serviceToUsers: ServiceToUser[];
```

В класі Service-to-user створюємо поля service, serviceId, user, userId і isActive і вказуємо відповідні декоратори. Цей зв’язок реалізується складніше, оскільки необхідно створити проміжну таблицю.

```
@Entity('service-to-user')
@Unique(['serviceId', 'userId'])
export class ServiceToUser extends BaseEntity {
  @PrimaryGeneratedColumn()
  public id!: number;

  @Column()
  public serviceId: number;

  @Column()
  public userId: number;

  @Column({ default: false })
  public isActive: boolean;

  @Column()
  public label: number;

  @Column()
  public block: number;

  @Column()
  public age: number;

  @Column()
  public shortName: string;

  @ManyToOne(type => Service, service => service.serviceToUsers,
```

```
    { cascade: ['insert', 'update'], onUpdate: 'CASCADE', onDelete: 'CASCADE' })
  @JoinColumn({ name: 'serviceId' })
  public service: Service;

  @ManyToOne(type => User, user => user.serviceToUsers,
    { cascade: ['insert', 'update'], onUpdate: 'CASCADE', onDelete: 'CASCADE' })
  @JoinColumn({ name: 'userId' })
  public user: User;
}
```

Cascade визначає типи операцій над батьківською таблицею, які будуть мати ефект на відповідні записи в дочірній таблиці. В даному випадку це CASCADE, при видалені та оновленні.

І відповідно у таблиці Service створюємо поле serviceToUsers і вказуємо відповідні анотації TypeOrm.

```
@Entity('service')
export class Service extends BaseEntity {
  @PrimaryGeneratedColumn()
  id: number;

  @Column({ nullable: false })
  name: string;

  @Column({ nullable: false, unique: true })
  key: string;

  // @ManyToMany(type => User, user => user.services)
  // @JoinTable()
  // users: User[];

  @OneToMany(type => Document, document => document.service)
  documents: Promise<Document[]>;

  @OneToMany(type => ServiceToUser, serviceToUser => serviceToUser.service)
  public serviceToUsers: ServiceToUser[];

  @OneToMany(type => ServiceToUser, serviceToUser => serviceToUser.service)
  public serviceToUsers: ServiceToUser[];
  @OneToMany(type => Request, request => request.service)
  public requests: Request[];
}
```

Таким чином реалізується зв'язок ManyToMany.

Використання TypeORM значно спрощує роботу з базою даних, оскільки маючи один об’єкт можна дістати всі пов’язані з ним об’єкти без написання складних SQL запитів.

3.3 Репозиторії

Розглянемо створення шару репозиторіїв для взаємодії з базою даних. Для створення репозиторію необхідно додати в проект клас, який наслідуватиме один із стандантних класів TypeORM, та буде помічений анотацією @EntityRepository. Створимо репозиторій для моделі користувача:

```
@EntityRepository(User)
export class UsersRepository extends Repository<User> {

  public async store(manager: EntityManager, createUserDto: UserCreateDto):
Promise<User> {

    const userToSave: DeepPartial<User> = {
      email: createUserDto.email,
      password: createUserDto.hashedPassword,
      roleId: createUserDto.roleId,
      photoUrl: createUserDto.photoUrl ? createUserDto.photoUrl : null,
      roleId: createUserDto.roleId,
      photoUrl: createUserDto.photoUrl ? createUserDto.photoUrl : null,

    };

    const insertResult = await manager.insert(User, userToSave);
    return await manager.findOne(User, insertResult.identifiers[0].id);
  }
}
```

Інтерфейс UserRepository наслідує Repository<>, який вже містить набір стандартних методів (запитів) для роботи з базою даних, таких як findById, save, deleteById, deleteAll, findAll. TypeORM автоматично згенерує реалізацію для таких методів, що значно спрощує взаємодію з базою даних та дозволяє не писати вручну SQL запити.

Щоб визначити свої запити можна просто описати методи в цьому ж класі.

Якщо запит досить складний чи потребує власної реалізації, можна скористатися методом entitytMeneger.query().

Запити корректно відпрацюють на будь-якій реляційній базі даних незалежно від SQL синтаксису. Таким чином реалізовано рівень репозиторіїв автоматизованої системи.

3.4 Сервіси

Наступний шар програми це рівень сервісів. Сервіси відповідають за основну логіку програми з обробки даних. Сервіси можуть дублювати деякі методи репозиторіїв, або містити додатковий функціонал.

Розглянемо ReportService:

```
@Injectable()
export class ReportsService {
  constructor(
    @InjectRepository(ReportsRepository)
    private readonly reportsRepository: ReportsRepository,
    private readonly usersService: UsersService,
  ) {
  }

  async saveReport(entityManager: EntityManager, userId: number, createReportDto:
ReportCreateDto) {
    try {
      createReportDto = plainToClass(ReportCreateDto, createReportDto);
      await validateOrReject(createReportDto);
    } catch (e) {
      throw new BadRequestException(`Post report values incorrect:
${JSON.stringify(e)}`);
    }
    return await this.reportsRepository.store(entityManager, createReportDto,
userId);
  }

  async getReportById(id: number) {
    const report = await this.reportsRepository.findOne({ id });
    if (!report) {
      throw new NotFoundException('Report not found');
    }
    return report;
  }

  async getReportsByUserId(userId: number, options: PaginationInterface):
Promise<ReportsListResponseDto> {
    const queryBuilder = this.reportsRepository.createQueryBuilder()
      .where({ userId });
    return await paginate(queryBuilder, options);
  }

  async deleteReport(id: number) {
    const report = await this.getReportById(id);
    return report.remove();
  }

  async getAllReports(options: PaginationInterface):
Promise<ReportsListResponseDto> {
    const queryBuilder = await
this.reportsRepository.createQueryBuilder('report');
    return await paginate(queryBuilder, options);
  }
```

Сервіси мають бути позначені декоратором @Injectable. Декоратори @Injectable над класами вказують фреймворку, що при запуску програми необхідно створити екземпляр цих класів та помістити в IoC контейнер. В програмі буде завжди один екземпляр такого класу, тобто реалізується шаблон Singleton. Щоб використати будь-який об’єкт з контейнера в іншому, необхідного оголосити змінну відповідного типу в конструкторі. Фреймворк знайде реалізацію вказаного типу в контейнері та підставить об’єкт в змінну:

```
constructor(
  @InjectRepository(ReportsRepository)
  private readonly reportsRepository: ReportsRepository,
  private readonly usersService: UsersService,
  private readonly usersService: PostsService,

) {
}
```

Таким чином реалізується шаблон Dependency Injection.

3.5 Безпека

Правильно впроваджена авторизація і автентифікація в API є надзвичайно важливою з точки зору безпеки. Вони дозволяють забезпечити захист конфіденційності, цілісності та доступності даних користувачів, а також контролювати доступ до ресурсів системи. Це допомагає запобігти несанкціонованому доступу, злому та зловживанню.

Автентифікація – це процес перевірки ідентичності користувача, тобто підтвердження, що користувач є тим, за кого себе видає. Це може включати перевірку логіна і пароля, використання токенів або сертифікатів. Завдяки автентифікації система переконується, що лише дозволені користувачі мають доступ до ресурсів.

Авторизація – це процес встановлення прав доступу користувача після успішної автентифікації. Після того, як користувач ідентифікований, система

перевіряє його права доступу і дозволяє або забороняє доступ до певних функцій, ресурсів або даних. Це дозволяє обмежити доступ до конфіденційної і критичної інформації тільки для вповноважених користувачів.

Для захисту додатку та реалізації авторизації використаємо NestJs Passport та Passport JWT. NestJs Passport фреймворк, що надає механізми побудови систем аутентифікації та авторизації, а також інші можливості забезпечення безпеки для промислових додатків, створених за допомогою NestJs Framework.

Запобігання запитам з невідомих джерел також є важливим аспектом безпеки. Для цього можуть використовуватись додаткові механізми, такі як перевірка IP-адреси, використання HTTPS-з'єднань з SSL-сертифікатами, обмеження доступу до API за допомогою налаштувань файрволу або використанням мережевих проксі-серверів.

Враховуючи важливість безпеки, розробники повинні ретельно розробити імплементацію авторизації та автентифікації в своєму API, використовуючи надійні протоколи та стандарти безпеки. Такі заходи допоможуть підвищити рівень захисту даних користувачів і запобігти потенційним загрозам безпеки.

Створимо локальку стратегію, яка при логіні користувача буде з контексту діставати його данні, перевіряти їх корректність та в успішному випадку повертати refresh та access token`s.

```
@Injectable()
export class LocalStrategy extends PassportStrategy(Strategy) {
  constructor(private readonly authService: AuthService) {
    super({ usernameField: 'email' });
  }

  async validate(email: string, password: string): Promise<any> {
    const user = await this.authService.validateUser(email, password);
    if (!user) {
      throw new BadRequestException('Password or email invalid');
    }
    return user;
  }
  async check(email: string, password: string): Promise<boolean> {
    const user = await this.authService.validateUser(email, password);
    if (!user) {
      return fasle
    }
    return Boolean(user);
  }
```

Створимо jwt. стратегію, яка буде використовуватись для автентифікації юзера при запитах до API. Для доступу по певному шляху API, клієнт повинен прикріпити Accsess токен який має отримати по окремому шляху, використовуючи Refresh токен. В контексті стратегія розшифрує токен і отримає з нього email і id користувача і API відповідно до цього email і id далі продовжить свою роботу вже ідентифікувавши користувача.

```
@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy) {
  constructor(
    private readonly config: ConfigService,
  ) {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      ignoreExpiration: false,
      secretOrKey: config.get('JWT_KEY'),
    });

    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      ignoreExpiration: false,
      secretOrKey: config.get('JWT_KEY'),
    });
  }

  async validate(payload: any) {
    return {
      id: Number(aes256.decrypt(this.config.get('AES256_KEY'), payload.sub)),
      email: payload.email,

    };
  }
}
```

Отже авторизація і автентифікація є дуже важливим модулем api. Так як може захистити. дані користувачів і заборонити запити з невідомих джерел.

3.6 Керівництво користувача

Для початку вам потрібно створити обліковий запис на нашій платформі, зареєструвавшись як фрілансер або роботодавець. Після реєстрації ви можете увійти і почати користуватися платформою. Для реєстрації використовується /signup ендпойт, який приймає тіло з атрибутами користувача.

Щоб змінити свій профіль, увійдіть на платформу та отримайте аксес і рефреш токен, зробити це можна за допомогою ендпойнту /login, який імплементує локальну стратегію авторизації і приймає в тіло запиту логін і пароль. Рефреш токен використовується коли термін дії аксес токену вичерпано і потрібно згенерувати новий, для цього можна використати ендпойт /auth передавши рефреш токен і отримати новий згенерований аксес токен. Якщо час сесії вичерпано і рефреш токен видалено, юзер повинен знову пройти через процес локальної стратегії авторизації.

З аксес токеном, можна використовувати всі ендпойти /user/profile з необхідними параметрами запиту, та отримувати інформацію про профіль юзера або оновлювати її. Ці ендпойти можуть редагувати особисту інформацію користувача, додавати його навички та оновлювати історію роботи. Користувач має підтримувати свій профіль в актуальному стані, оскільки це допоможе потенційним роботодавцям знайти його.

Щоб шукати вакансії або співробітників, можна скористатись з ендпойнтами пошуку, які приймають різноманітні критерії вакансії та співробітників і повернуть відповідний результат. Юзер можете встановити критерії пошуку на основі місцезнаходження, навичок та інших вимог. Після того, як ви встановите критерії пошуку і відправите запит, сервер поверне список релевантних результатів, які відповідають вашим критеріям.

Щоб переглянути профіль іншого користувача, можна скористатись ендпойтами профілю юзера. За допомогою них ви можете переглянути особисту інформацію, навички та історію роботи іншого юзера, просто вказавши його ідентифікатор в параметрах запиту. В результуючих даних ви також можете перевірити його рейтинг і прочитати відгуки від інших користувачів.

Щоб написати відгук, перейдіть потрібно відправити запит по ендпойту з відгуками, передавши в тіло ідентифікатор користувача, якому ви хочете його залишити. Також в тілі запиту, потрібно вказати інформацію, яка буде відображати оцінку користувача за шкалою від 1 до 5 зірок і надати письмовий відгук про ваш досвід спілкування з ним в поле опису. Цей відгук буде видно

іншим користувачам і буде повертатись в результуючих даних разом з їхнім профілем і допоможе їм приймати більш обґрунтовані рішення при роботі з цим користувачем та мати певну довіру.

Панель адміністратора доступна лише адміністраторам платформи тобто юзерам, які в базі даних мають відповідну роллю. Всі едпойти, промарковані префіксом – адмін і до них має доступ користувач з відповідною роллю. За допомогою цих ендпойнтів ви можете керувати всіма користувачами на платформі, в тому числі змінювати профілі користувачів, видаляти користувачів і переглядати їхню активність.

3.7 Технічні характеристики серверної частини

Як було вказано раніше всі компоненти серверної чатстини розгортаються за допомогою Docker. Тому в цілому можна абстрагуватись від операційної системи та середовища та сконцентруватись лиши на характеристиках серверу. Так, як в нас є досить велика база даних і сам сервер потребує певних ресурсів, вимогу будуть такими:

- будь яка ОС;
- встановлене та налаштоване ПЗ Docker
- 2 GB ОЗП;
- 15 GB постійної пам’яті.

3.8 Тестування серверної частини

Тестування REST API передбачає перевірку того, що API поводиться як очікується, шляхом надсилання запитів до кінцевих точок API та вивчення релевантних відповідей.

Модульне тестування: Цей метод тестує окремі функції, методи або класи ізольовано. У контексті REST API він передбачає тестування бізнес-логіки

кожної кінцевої точки API. Юніт-тести можуть бути написані за допомогою фреймворків для тестування, таких як Mocha, Jest або Jasmine.

Інтеграційне тестування: Інтеграційне тестування перевіряє, чи працюють окремі компоненти системи разом, як очікувалося. У контексті REST API воно включає в себе перевірку того, як кінцеві точки API працюють разом з іншими компонентами системи, такими як база даних або система автентифікації. Інтеграційні тести можуть бути написані за допомогою фреймворків тестування, таких як Supertest або Chai.

Функціональне тестування: функціональне тестування перевіряє, чи відповідає система функціональним вимогам, зазначеним в документації до API. У контексті REST API воно включає в себе тестування кінцевих точок API на відповідність вимогам специфікації API. Функціональні тести можуть бути написані за допомогою фреймворків тестування, таких як Cucumber або Behave.

Навантажувальне тестування: навантажувальне тестування перевіряє поведінку системи під великим навантаженням. У контексті REST API воно включає в себе тестування кінцевих точок API в умовах інтенсивного трафіку або декількох одночасних запитів. Навантажувальні тести можуть бути написані за допомогою фреймворків тестування, таких як Apache JMeter або Gatling.

Тестування безпеки: тестування безпеки перевіряє безпеку API, намагаючись використати вразливості в системі. У контексті REST API воно включає в себе тестування кінцевих точок API на наявність загальних уразливостей безпеки, таких як SQL-ін'єкція, міжсайтовий скриптинг (XSS) або підробка міжсайтових запитів (CSRF). Тести безпеки можуть бути написані за допомогою фреймворків тестування, таких як OWASP ZAP або Burp Suite.

Ручне тестування: ручне тестування передбачає тестування кінцевих точок API вручну за допомогою веб-браузера або інтерфейсу командного рядка, наприклад, cURL. Ручне тестування може бути корисним для дослідницького тестування, спеціального тестування або для тестування крайніх випадків.

Підсумовуючи, тестування REST API включає в себе поєднання модульного тестування, інтеграційного тестування, функціонального тестування,

навантажувального тестування, тестування безпеки та ручного тестування. Кожен метод служить різним цілям і повинен використовуватися в залежності від конкретних потреб API в тестуванні.

Виходячи з бізнес задач та можливостей в ході розробки програмного забезпечення, було проведено тільки мануальне тестування, тобто тестування вручну. Цей спосіб тестування є найбільш точним та дозволяє виявити найбільш проблемні та вразливі місця системи і також провалідувати програму на відповідність всім вимогам.

Зміни у графічних інтерфейсах користувача дуже важко перевірити автоматично. Опишемо декілька тестових сценарії в таблиці 4.1.

Описані в таблиці 4.1 сценарії покривають більшість ключових функції та процесів програми, протестувавши які, можна впевнитись в його готовності та відповідності до всіх вимог.

Мануальне тестування є важливою частиною процесу розробки програмного забезпечення і дозволяє виявити проблеми та вразливості в системі. Цей метод тестування передбачає виконання тестових сценаріїв та перевірку функціональності програми вручну.

Переваги мануального тестування включають:

– тестувальний спеціаліст може уважно перевірити кожну функцію та аспект програми, що дозволяє виявити навіть незначні проблеми;

– мануальне тестування дозволяє адаптувати тестові сценарії під час виконання, змінюючи послідовність та параметри тестів залежно від потреб;

– відсутність залежності від автоматизації: мануальне тестування може бути проведене навіть без повноцінних автоматизованих тестових сценаріїв, що дозволяє швидко виконувати перевірку нових функцій або змін у програмі;

– тестувальний спеціаліст може використовувати свої знання та інтуїцію для виявлення потенційних проблем або несподіваних ситуацій у програмі.

Однак, важливо розуміти, що мануальне тестування має свої обмеження:

– часові затрати (мануальне тестування може бути часо- та ресурсозатратним, особливо при великому обсязі функціоналу або повторних тестах);

– людський фактор (тестування залежить від тестувальних спеціалістів, їхнього досвіду та уваги. Іноді можуть виникати помилки або пропуски через людський фактор);

– складність повторного виконання: (мануальні тестові сценарії можуть бути складні для повторного виконання, особливо при необхідності регулярних перевірок або автоматизованих випусках).

У процесі розробки програмного забезпечення рекомендується комбінувати мануальне тестування з автоматизованим тестуванням, щоб забезпечити якість та ефективність тестування. Автоматизовані тести можуть бути використані для повторного виконання рутинних сценаріїв, тестування інтеграцій, продуктивності та інших аспектів системи, тоді як мануальне тестування може бути спрямоване на більш творчу перевірку та виявлення проблем, які можуть уникнути автоматизації.

В коді нижче можна побачити приклад описанню юніт тестів для сервісу юзерів за допомогою бібліотеки Jest.

```
const UserService = require('./user');
test('додає користувача', () => {
  const userService = new UserService();
  const user = { id: 1, name: 'John' };

  userService.addUser(user);

  expect(userService.getUsers()).toContain(user);
});

test('отримує користувача за ідентифікатором', () => {
  const userService = new UserService();
  const user1 = { id: 1, name: 'John' };
  const user2 = { id: 2, name: 'Jane' };

  userService.addUser(user1);
  userService.addUser(user2);

  expect(userService.getUserById(1)).toBe(user1);
});

test('видаляє користувача за ідентифікатором', () => {
  const userService = new UserService();
```

```
  const user1 = { id: 1, name: 'John' };
  const user2 = { id: 2, name: 'Jane' };

  userService.addUser(user1);
  userService.addUser(user2);

  userService.deleteUserById(1);

  expect(userService.getUsers()).not.toContain(user1);
  expect(userService.getUsers()).toContain(user2);
});
```

В наступному прикладі коду описано тестування модулю авторизацію за допомогою бібліотеки Jest.

```
// auth.service.ts
import { UserRepository } from './user.repository';
import { HashService } from './hash.service';

export class AuthService {
  constructor(
    private readonly userRepository: UserRepository,
    private readonly hashService: HashService,
  ) {}

  async login(username: string, password: string): Promise<boolean> {
    const user = await this.userRepository.findByUsername(username);
    if (!user) {
      return false;
    }

    const passwordMatch = await this.hashService.comparePasswords(
      password,
      user.password,
    );
    if (!passwordMatch) {
      return false;
    }

    return true;
 it('should return false if password is incorrect', async () => {
      const username = 'testuser';
      const password = 'testpassword';
      const hashedPassword = await hashService.hashPassword('wrongpassword');

      jest
        .spyOn(userRepository, 'findByUsername')
        .mockResolvedValueOnce({ username, password: hashedPassword });

      jest
        .spyOn(hashService, 'comparePasswords')
        .mockResolvedValueOnce(false);

      const result = await authService.login(username, password);

      expect(result).toBe(false);
    });
  });
```

Таблиця 3.1 – Тестові сценарії

Назва сценарію	Модуль	Метод	Вхідні дані	Очікуваний результат
Реєстрація з коректними даними	User	Signup	Тіло запиту з атрибутами юзера	Створення юзера в базі та успішний результат від серверу
Реєстрація з некоректними даними	User	Signup	Тіло запиту не з усіма атрибутами юзера	Помилка від серверу через неправильне тіло запиту
Логін з коректними даними	Auth	Login	Тіло запиту з коректним паролем і логіном	Генерація аксес токену, рефреш токену і успішний результат від серверу
Логін з некоректними даними	Auth	Logic	Тіло запиту з неправильним паролем або логіном	Повернення помилки серверу через неправильний пароль або логін
Отримання профілю користувача	User	GET user	Прикріплення аксес токену до запиту	Дані профілю юзера в результаті запиту
Зміна паролю	User	resetPassword	Тіло запиту з старим паролем, новим паролем та емейлом	Оновлений пароль юзера та можливість залогінитись з новим паролем
Перегляд профілю іншого користувача	User	GET user	Ідентифікатор необхідного юзеру в параметрах запиту	Повернення успішного результату від сервера з профілем необхідного користувача
Пошук користувача з існуючими параметрами	User	user/search	Тіло запиту з необхідними квері параметрами, при умові, що юзер з такими параметрами існує	Список знайдених юзерів з збігом параметрів
Пошук користувача з неіснуючими параметрами	User	User/search	Тіло запиту з необхідними квері параметрами, при умові, що юзер з такими параметрами не існує	Успішна відповідь від серверу з пустими списком юзерів
Написання відгуку іншому користувачу	Reviews	POST review	Тіло запиту з ідентифікатором користувача, якому пишеться відук, оцінкою/рейтингом та коротким описом, який включає стислу оцінку роботи	Повернення успішної відповіді з серверу з моделлю Review
Завантаження документу	User	POST user/documents	Посилання на файл	Прикріплений файл до користувача
Написання повідомлення іншому користувачу	Message	POST message	Отримувач та текст повідомлення	Надіслане повідомлення

Модульне тестування – це процес тестування окремих "модулів" програмного забезпечення, щоб переконатися, що кожен модуль працює правильно. Таким чином, результативне модульне тестування може допомогти виявити окремі помилки та проблеми, які можуть виникнути в процесі роботи програми. Це важлива складова процесу розробки програмного забезпечення, яке дозволяє перевірити правильність роботи окремих модулів. Модуль в даному контексті визначається як найменша функціональна одиниця програми, яка може бути тестована індивідуально.

Результати модульного тестування наведено у таблиці 4.2.

Таблиця 3.2 – Результати модульного тестування

Назва сценарію	Модуль	Метод	Вхідні дані	Очікуваний результат	Отриманий результат
Реєстрація з коректними даними	User	Signup	Тіло запиту з атрибутами юзера	Створення юзера в базі та успішний результат від серверу	Позитивний
Реєстрація з некоректними даними	User	Signup	Тіло запиту не з усіма атрибутами юзера	Помилка від серверу через неправильне тіло запиту	Позитивний
Логін з коректними даними	Auth	Login	Тіло запиту з коректним паролем і логіном	Генерація аксес токену, рефреш токену і успішний результат від серверу	Позитивний
Логін з некоректними даними	Auth	Logic	Тіло запиту з неправильним паролем або логіном	Повернення помилки серверу через неправильний пароль або логін	Помилка. Помилку виправлено.
Отримання профілю користувача	User	GET user	Прикріплення аксес токену до запиту	Дані профілю юзера в результаті запиту	Позитивний
Зміна паролю	User	resetPassword	Тіло запиту з старим паролем, новим паролем та емейлом	Оновлений пароль юзера та можливість залогінитись з новим паролем	Позитивний

Кінець таблиці 3.2

Назва сценарію	Модуль	Метод	Вхідні дані	Очікуваний результат	Отриманий результат
Перегляд профілю іншого користувача	User	GET user	Ідентифікатор необхідного юзеру в параметрах запиту	Повернення успішного результату від сервера з профілем необхідного користувача	Позитивний
Пошук користувача з існуючими параметрами	User	user/search	Тіло запиту з необхідними квері параметрами, при умові, що юзер з такими параметрами існує	Список знайдених юзерів з збігом параметрів	Позитивний
Пошук користувача з неіснуючими параметрами	User	User/search	Тіло запиту з необхідними квері параметрами, при умові, що юзер з такими параметрами не існує	Успішна відповідь від серверу з пустими списком юзерів	Помидка. Помилку виправлено
Написання відгуку іншому користувачу	Reviews	POST review	Тіло запиту з ідентифікатором користувача, якому пишеться відук, оцінкою/рейтингом та коротким описом, який включає стислу оцінку роботи	Повернення успішної відповіді з серверу з моделлю Review	Позитивний
Завантаження документу	User	POST user/documents	Посилання на файл	Прикріплений файл до користувача	Помилка, помилку виправлено

3.9 Висновки до розділу 3

Висновок. В даному розділі було виконана програмна реалізація розробленого проекту на платформі NodeJs, NestJs Framework, визначено основні структурні компоненти та взаємодію між ними.

Структура програмної системи базується на концепції модульності і розділеності функціональності. Різні компоненти системи, такі як адміністраторський модуль, модуль звичайного користувача та інші, взаємодіють між собою через визначені інтерфейси та API.

Завдяки використанню Docker, серверну частину можна розгортати на будь-якій операційній системі, що підтримує Docker, незалежно від конкретної

конфігурації. Це забезпечує гнучкість і зручність в установці та розгортанні сервера, оскільки весь необхідний софт вже міститься в контейнерах Docker.

Отже в результаті ручного тестування було виявлено та виправлено декілька недоліків програмного продукту.

Результати тестування демонструють, що програмне забезпечення задовольняє всім встановленим вимогам. Очікувані результати збігаються з фактичними результатами тестування, що дозволяє зробити висновок про те, що програма була розроблена відповідно до специфікацій та вимог викладених у технічному завданні.

Протягом процесу тестування було виявлено та виправлено деякі помилки і проблеми, що виникали, а також проведено оптимізацію швидкодії та надійності системи. Всі виявлені проблеми були вирішені, що дозволяє зробити висновок про те, що програма була розроблена відповідно до вимог, викладених у технічному завданні, і демонструє задовільні результати.

Цей успішний результат тестування свідчить про якісну розробку програмного забезпечення і готовність його до впровадження та використання. Завдяки проведеному тестуванню можна мати впевненість, що програмне забезпечення працездатне та надійне у реальних умовах експлуатації.

Повний код програмної системи наведено у додатку Б.

ВИСНОВКИ

В даній кваліфікаційній роботі було розроблено REST API для фріланс-сервісу. Реалізовано підсистеми для адміністратору та звичайного користувача, передбачено облік основних об’єктів предметної області, роботу з документами, заявками, рейтингами, відгуками, сферами діяльності та перегляд статистики і менеджменту користувачів адміністратором.

В ході роботи було проаналізовано предметну область, проведено оцінку актуальності та доцільності розробки, розглянуто аналоги. Розроблене програмне забезпечення відрізняється від аналогів через багатий вибір сфер діяльності, які не налаштовані під онлайн послуги. Воно також є кросс-платформним, не має обмежень і є досить доступним для користувачів. Ця система має великий потенціал для використання у фріланс-сервісах. Розроблене програмне забезпечення відрізняється від аналогів своєю широкою функціональністю та гнучкістю. Користувачі мають можливість обирати з багатого спектру сфер діяльності та категорій робіт, що дозволяє задовольнити різноманітні потреби. У процесі аналізу предметної області було виявлено потребу у визначенні ролей користувачів, що дозволяє надавати різні рівні доступу до функціоналу системи. Адміністратор системи може керувати користувачами, модерувати заявки та відгуки, встановлювати правила і обмеження. Звичайний користувач має можливість створювати проекти, подавати заявки на виконання робіт, спілкуватись з іншими користувачами та залишати відгуки.

Визначено постановку завдання та вимоги, розроблено специфікацію програмного продукту та UML діаграми. Спроектовано архітектуру програмної системи, структуру бази.

Для розробки проекту було обрано платформу NodeJs, та мову програмування TypeScript і фреймоврк NestJs з TypeOrm. Важливим аспектом розробленої системи є також можливість зберігання та обробки великого обсягу даних. Застосована реляційна база даних PostgreSQL дозволяє ефективно зберігати, управляти та отримувати доступ до інформації про користувачів,

проекти, заявки та інші об'єкти предметної області. Здійснено ручне тестування серверної частини розробленого програмного забезпечення.

Розроблене програмне забезпечення може використовуватись для фріланс-сервісу. Система виділяється серед аналогів: багатим вибором сфер діяльності, які не налаштоване під онлайн послуги, кросс-платформеністю, відсутністю обмежень та доступністю. Загалом, розроблена система фріланс-сервісу є потужним інструментом для замовників та фрілансерів, забезпечуючи зручний спосіб знаходження та виконання проектів, ефективне спілкування та розрахунок за надані послуги.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Фріланс біржа № 1 в Україні. URL: https://freelance.ua (дата звернення: 17.04.2023).

2. Онлайн-сервіс замовлення послуг. URL: https://kabanchik.ua (дата звернення: 17.04.2023).

3. Сервіс найкращих фрилансерів для виконання ваших завдань. URL: https://freelancehunt.com/ua (дата звернення: 21.04.2023).

4. Біржа фріланса. URL: https://www.weblancer.net (дата звернення: 19.04.2023).

5. Рефакторинг.Гуру, сервіс, який присвячений темним матеріям програмування. URL: https://refactoring.guru/ (дата звернення: 25.04.2023).

6. Stack Overflow допомагає людям знаходити потрібні відповіді, коли вони їм потрібні. URL: https://stackoverflow.com/ (дата звернення: 27.04.2023).

7. PostgreSQL – найдосконаліша у світі реляційна база даних з відкритим кодом. URL: https://www.postgresql.org/ (дата звернення: 27.04.2023).

8. Node.js – платформа з відкритим кодом для виконання високопродуктивних мережевих застосунків, написаних мовою JavaScript. URL: https://nodejs.org/uk (дата звернення: 27.04.2023).

9. Принципи SOLID. URL: https://uk.wikipedia.org/wiki/SOLID (дата звернення: 17.04.2023).

10. Берт Брейтс, Ерік Фрімен. Патерни проектування : практ. посіб. Київ : Фабула, 2020. 672 с.

11. Документація NestJs. URL: https://nestjs.com/ (дата звернення: 21.04.2023).

12. Документація TypeScript. URL: https://www.typescriptlang.org/ (дата звернення: 19.04.2023).

13. Патрік Дебуа, Джон Вілліс. DevOps : практ. посіб. Київ : Фабула, 2020. 384 с.

14. MailGun служба доставки електронної пошти. URL: https://www.mailgun.com/ (дата звернення: 29.04.2023).

15. GeoNames – географічна база даних. URL: https://www.geonames.org/ (дата звернення: 29.04.2023).

16. Docker – інструментарій для управління ізольованимми контейнерами. URL: https://www.docker.com/ (дата звернення: 29.04.2023).

17. Minio – ресурс для збереження артифактів. URL: https://min.io/ (дата звернення: 29.04.2023).

18. Redis – швидкодоступне сховище данних. URL: https://redis.io/ (дата звернення: 29.04.2023).

19. NPM – пакетний менеджер. URL: https://www.npmjs.com/ (дата звернення: 21.04.2023).

20. PNPM – пакетний менеджер. URL: https://www.pnpmjs.com/ (дата звернення: 21.04.2023).

21. Yarn – пакетний менеджер. URL: https://yarnpkg.com/ (дата звернення: 21.04.2023).

22. Stack Exchange допомагає людям знаходити потрібні відповіді, коли вони їм потрібні. URL: https://stackexchange.com/ (дата звернення: 27.04.2023).

23. Vim – текстовий редактор. URL: https://www.vim.org/ (дата звернення: 25.04.2023).

24. Документація NextJs. URL: https://nextjs.org (дата звернення: 21.04.2023).

25. Документація ExpressJS. URL: https://expressjs.com/ (дата звернення: 27.04.2023).

26. Документація TypeOrm. URL: https://typeorm.io/ (дата звернення: 29.04.2023).

26. Документація SequelizeOrm. URL: https://sequelize.org/ (дата звернення: 30.04.2023).

27. Документація PrismaOrm. URL: https://www.prisma.io/ (дата звернення: 30.04.2023).

28. Документація та довідник по JavaScript. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript (дата звернення: 1.05.2023).

29. MySQL – вільна система керування реляційними базами даних. URL: https://www.mysql.com/ (дата звернення: 1.05.2023).

30. Сервіс для збереження кодової бази GitHub. URL: https://github.com/ (дата звернення: 27.04.2023).

31. Принципи YAGNI. URL: https://uk.wikipedia.org/wiki/Принцип_YAGNI (дата звернення: 21.04.2023).

32. Принципи KISS. URL: https://uk.wikipedia.org/wiki/Принцип_KISS (дата звернення: 21.04.2023).

33. Платформа для тестування та створення API. URL: https://www.postman.com/ (дата звернення: 19.04.2023).

34. Сервіс для збереження кодової бази GitLab. URL: https://about.gitlab.com/ (дата звернення: 27.04.2023).

35. Документація Git. URL: https://git-scm.com/ (дата звернення: 2.05.2023).

36. AWS – платформа хмарних обчислень. URL: https://aws.amazon.com/ (дата звернення: 25.04.2023).

37. GoogleCloud – платформа хмарних обчислень. URL: https://cloud.google.com/ (дата звернення: 25.04.2023).

38. Microsoft Azure – платформа хмарних обчислень. URL: https://azure.microsoft.com/ru-ru (дата звернення: 02.05.2023).

39. DigitalOcean – провайдер хмарних ресурсів. URL: https://www.digitalocean.com/ (дата звернення: 27.04.2023).

40. CockroachLabs – документація реляційної бази даних CockroachDB. URL: https://www.cockroachlabs.com/ (дата звернення: 5.05.2023).

ДОДАТОК А
(обов’язковий)

ТЕХНІЧНЕ ЗАВДАННЯ

Введення

Робота виконується в рамках кваліфікаційної роботи розробки серверної частини REST API для кросплатформного фріланс-сервісу LiveFreelance. Технічне завдання розроблено у відповідності до стандарту ГОСТ 19.201–78.

1 Підстава для розробки

Підставою для розробки є «Завдання на кваліфікаційну роботу», затверджене завідувачем кафедри інженерії програмного забезпечення. Найменування розробки: «Серверна частина REST API для кросплатформного фріланс-сервісу LiveFreelance».

2 Призначення розробки

2.1 Функціональне призначення

Функціональним призначенням додатку є створення серверної частини для фріланс-сервісу.

2.2 Експлуатаційне призначення

Програма може експлуатуватись і використовуватись, як інтеграція з сервером. Ітеграція може відбуватись з будь якого клієнта, який можу надсилати HTTP запити.

3 Вимоги до програми

3.1 Вимоги до функціональних характеристик

Програма повинна забезпечувати можливості виконання таких функцій:

– функція реєстрації та логіну;
– функція скидання паролю;
– функція перегляду та оновлення свого профілю;
– функція завантаження документів в профіль;
– можливість пошуку інших користувачів за заданими параметрами;
– можливість переглядати профіль інших користувачів;
– функція написання відгуків іншим користувачам;
– функція перегляду відгуків інших користувачів;
– функція для комунікації між користувачами;
– функція видалення документів з профілю;
– функція виходу з системи;

– функція адміністративної панелі;

– забезпечення трьох ролей користувачів;

– доступ користувачам з роллю адміністратору до адмін панелі;

– менеджмент юзерів з адміністративної панелі;

– генерація та оновлення аналітики.

3.2 Вимоги до надійності

Програма повинна забезпечувати наступні вимоги до надійності:

– усі сенсативні дані повинні надійно зберігатись та шифруватись в системі;

– передбачення та відображення інтуїтивно зрозумілих помилок для користувача;

– можливість видалення всіх особистих даних користувача;

– можливість для масштабування інфраструктури в разі збільшення трафіку.

3.3 Умови експлуатації та вимоги до технічних засобів

Програмне забезпечення та всі його компоненти розгортаються за допомогою Docker, який містить в собі повне налаштування оточення, яке є необхідним для експлуатації додатку.

Для успішного запуску додатку пристрій повинен відповідати наступним рекомендованим вимогам:

– встановлене та налаштоване ПЗ Docker;

– 2 ГБ ОЗП;

– 15 GB постійної пам’яті.

3.4 Вимоги до інформаційної та програмної сумісності

При розробці планується використати платформу Node.js, яка є досить зручною та легко масштабується в разі потреби. Мовою програмування має бути TypeScript який може використовуватись не тільки на стороні серверної частини але й на стороні клієнта, що може значно спростити розробку. В якості СУБД використовуватиметься Postgresql, а TypeOrm, як ORM, яка дозволяє побудувати ентіті відповідно до бізнес-логіки та відобразити їх в базі, створивши таблички, зв’язки та автоматично згенерувавши міграції.

3.5 Спеціальні вимоги

Програма має бути інтуїтивно зрозумілою, швидкою, та мати доступний для кінцевого користувача інтерфейс, який дозволить йому вдало та зручно з нею взаємодіяти та працювати.

4 Вимоги до програмної документації

У момент здачі кваліфікаційнї роботи замовнику надається наступний набір документів:

– текст програми;

– опис програми;

– технічне завдання;

– керівництво користувача.

5 Стадії та етапи розробки

Стадії та етапи розробки серверної частини REST API для кросплатформного фріланс-сервісу LiveFreelance подані у таблиці А.1.

Таблиця А.1 – Стадії та етапи розробки

Стадія розробки	Етапи робіт	Зміст робіт
1	2	3
Технічне завдання 02.01.23 – 31.01.23	Обґрунтування необхідності розробки програми	Коротка характеристика програмного забезпечення; підстава і призначення розробки; вимоги до програмної системи і документація; стадії і етапи розробки програми; порядок контролю і приймання
Ескізний проект 01.02.23 – 28.02.23	Розробка ескізного проекту	Попередня розробка структури вхідних і вихідних даних; уточнення середовища програмування; розробка і опис загальної алгоритмічної структури
Технічний проект 1.03.23 – 15.03.23	Розробка технічного проекту	Уточнення структури вхідних і вихідних даних; розробка докладного алгоритму; розробка структури програми
Робочий проект 15.03.23 – 7.04.23	Розробка програмного забезпечення	Реалізація програмного забезпечення; відлагодження; проведення попереднього тестування
Розробка програмної документації 7.04.23 – 10.04.23	Розробка документації до програмного забезпечення	Розробка необхідної документації, передбаченої технічним завданням
Тестування системи 11.04.23 – 30.04.23	Проведення тестування програмного забезпечення	Розробка методики тестування; проведення основних тестів; коректування програмного забезпечення
Впровадження	Підготовка і передача програми	Підготовка і розгортання програмного забезпечення

6 Порядок контролю та приймання

Контроль здійснюється клієнтами серверу, підключеними під час процесу тестування та автоматизованим програмним забезпеченням під час розгортання.

ДОДАТОК Б
(обов’язковий)

КОД (ЛІСТИНГ) СЕРВЕРНОЇ ЧАСТИНИ LIVE FRELANCE

AdminInfo.ts

```
@Entity('admin-info')
export class AdminInfo extends BaseEntity {

  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  userId: number;

  @OneToOne(type => User, user => user.adminInfo,
    { cascade: true, onDelete: 'CASCADE', onUpdate: 'CASCADE' },
  )
  @JoinColumn({ name: 'userId' })
  user;

  @Column()
  firstName: string;

  @Column({ nullable: true })
  lastName: string;

  @Column({
    nullable: true,
  })
  phoneNumber: string;
}
```

CustomerInfo.ts

```
@Entity('customer-info')
export class CustomerInfo extends BaseEntity {

  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  userId: number;

  @OneToOne(type => User, user => user.customerInfo,
    { cascade: true, onDelete: 'CASCADE', onUpdate: 'CASCADE' },
  )
  @JoinColumn({ name: 'userId' })
  user;

  @Column()
  firstName: string;

  @Column({ nullable: true })
  lastName: string;

  @Column({ nullable: true })
  cityId: number;

  @Column({ nullable: true })
  longitude: number;

  @Column({ nullable: true })
```

```
  latitude: number;

  @Column({ nullable: true })
  description: string;

  @Column({ nullable: true })
  addressLine: string;

  @OneToMany(type => CustomerPhone, customerPhone => customerPhone.customer)
  phoneNumbers: CustomerPhone[];

  city?: CityDto;
}
```

Document.ts

```
@Entity('document')
export class Document extends BaseEntity {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  name: string;

  @Column()
  description: string;

  @Column({ nullable: true })
  isActive: boolean;

  @Column({ default: () => 'FALSE' })
  isDeleted: boolean;

  @CreateDateColumn({ type: 'timestamp', default: () => 'LOCALTIMESTAMP' })
  createDate?: string;

  @UpdateDateColumn({ type: 'timestamp', default: () => 'LOCALTIMESTAMP' })
  updateDate?: string;

  @Column()
  userId: number;

  @ManyToOne(type => User, user => user.documents,
    { cascade: true, onDelete: 'CASCADE', onUpdate: 'CASCADE' },
  )
  @JoinColumn({ name: 'userId' })
  user: User;

  @Column()
  serviceId: number;

  @ManyToOne(type => Service, service => service.documents,
    { cascade: true, onDelete: 'CASCADE', onUpdate: 'CASCADE' },
  )
  @JoinColumn({ name: 'serviceId' })
  service: Service;

  @OneToMany(type => DocumentPhoto, documentPhoto => documentPhoto.document)
  documentPhotos?: Promise<DocumentPhoto[]>;
}
```

Request.ts

```
@Entity('request')
@Check(`"clientId" <> "performerId"`)
export class Request extends BaseEntity {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
```

```
  clientId: number;

  @ManyToOne(type => User, user => user.requestClients,
    { cascade: true, onDelete: 'CASCADE', onUpdate: 'CASCADE' })
  @JoinColumn({ name: 'clientId' })
  client: User;

  @Column()
  performerId: number;

  @ManyToOne(type => User, user => user.requestPerformers,
    { cascade: true, onDelete: 'CASCADE', onUpdate: 'CASCADE' })
  @JoinColumn({ name: 'performerId' })
  performer: User;

  @Column()
  serviceId: number;

  @ManyToOne(type => Service, service => service.requests,
    { cascade: true, onDelete: 'CASCADE', onUpdate: 'CASCADE' })
  @JoinColumn({ name: 'serviceId' })
  service: Service;

  @Column()
  message: string;

  @Column({ nullable: true })
  clientPhone: string;

  @Column()
  statusId: number;

  @ManyToOne(type => RequestStatus, status => status.requests,
    { cascade: ['insert', 'update'], onDelete: 'RESTRICT', onUpdate: 'CASCADE' })
  @JoinColumn({ name: 'statusId' })
  status: RequestStatus;

  @OneToMany(type => RequestFile, file => file.request)
  requestFiles?: RequestFile[];

  @CreateDateColumn({ type: 'timestamp', default: () => 'LOCALTIMESTAMP' })
  createDate?: string;

  @UpdateDateColumn({ type: 'timestamp', default: () => 'LOCALTIMESTAMP' })
  updateDate?: string;
}
```

Review.ts

```
@Entity('review')
@Check(`"userId" <> "authorId"`)
export class Review extends BaseEntity {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  userId: number;

  @ManyToOne(type => User, user => user.reviews,
    { cascade: true, onDelete: 'CASCADE', onUpdate: 'CASCADE' })
  @JoinColumn({ name: 'userId' })
  user: User;

  @Column({ type: 'integer' })
  rating: number;

  @Column()
```

```
  authorId: number;

  @ManyToOne(type => User, user => user.writtenReviews,
    { cascade: true, onDelete: 'CASCADE', onUpdate: 'CASCADE' })
  @JoinColumn({ name: 'authorId' })
  author: User;

  @Column({ nullable: true })
  text?: string;

  @Column({ nullable: true })
  answer?: string;

  @CreateDateColumn({ type: 'timestamp', default: () => 'LOCALTIMESTAMP' })
  createDate?: string;

  @UpdateDateColumn({ type: 'timestamp', default: () => 'LOCALTIMESTAMP' })
  updateDate?: string;
}
```

Service.ts

```
@Entity('service')
export class Service extends BaseEntity {
  @PrimaryGeneratedColumn()
  id: number;

  @Column({ nullable: false })
  name: string;

  @Column({ nullable: false, unique: true })
  key: string;

  @OneToMany(type => Document, document => document.service)
  documents: Promise<Document[]>;

  @OneToMany(type => ServiceToUser, serviceToUser => serviceToUser.service)
  public serviceToUsers: ServiceToUser[];

  @OneToMany(type => Request, request => request.service)
  public requests: Request[];
}
```

Сервіси

AdminsServices.ts

```
@Injectable()
export class AdminsInfoService {
  selections = ['admin-info.id', 'user.id', 'admin-info.firstName'
    , 'admin-info.lastName', 'admin-info.phoneNumber'
    , 'user.email', 'user.photoUrl'];

  constructor(
    private connection: Connection,
    private readonly usersService: UsersService,
    private readonly authService: AuthService,
    private readonly usersRatingsService: UsersRatingsService,
    @InjectRepository(AdminsInfoRepository)
    private readonly adminInfoRepository: AdminsInfoRepository,
  ) {
  }

  async save(createUserDto: UserCreateDto) {
    createUserDto.roleId = 1;
    if (createUserDto.password !== createUserDto.passwordConfirm) {
      throw new BadRequestException('Passwords are not match');
    }
    if (await this.usersService.findUserByEmail(createUserDto.email)) {
      throw new ConflictException('User with such email has already exist');
    }
```

```
    const savedUser = await this.connection.transaction(async manager => {
      const user = await this.usersService.store(manager, createUserDto);
      await this.adminInfoRepository.store(manager, createUserDto, user.id);
      return plainToClass(UserDto, user);
    });
    return await this.authService.login(savedUser);
  }

  async getAdminInfoByUserId(userId: number): Promise<AdminInfo> {
    return await this.adminInfoRepository.createQueryBuilder('admin-info')
      .innerJoin('admin-info.user', 'user')
      .where('user.id = :id', { id: userId })
      .andWhere('user.isDeleted = false')
      .andWhere('user.isActive = true')
      .getOne();
  }

  async getAdminAllInfoByUserId(userId: number): Promise<AdminProfileResponseDto>
{
    const adminProfile = plainToClass(AdminProfileResponseDto, await
this.adminInfoRepository.createQueryBuilder('admin-info')
      .select(this.selections)// And services, documents and reviews....
      .innerJoin('admin-info.user', 'user')
      .where({ userId })
      .andWhere('user.isDeleted = false')
      .andWhere('user.isActive = true')
      .getOne());
    if (!adminProfile) {
      throw new NotFoundException('Profile not found');
    }
    return adminProfile;
  }

  async getAllAdmins(options: PaginationInterface):
Promise<AdminsListResponseDto> {
    const queryBuilder = await
this.adminInfoRepository.createQueryBuilder('admin-info')
      .select(this.selections)// And services, documents and reviews....
      .innerJoin('admin-info.user', 'user')
      .innerJoin('user.role', 'role')
      .where('user.isDeleted = false')
      .andWhere('user.isActive = true');
    return await paginate<AdminInfo>(queryBuilder, options);
  }

  async updateAdminInfo(userId: number, adminInfoFullUpdateDto:
AdminInfoFullUpdateDto): Promise<AdminInfoFullUpdateResponseDto> {
    const adminInfoUpdateResponse: AdminInfoFullUpdateResponseDto = new
AdminInfoFullUpdateResponseDto();
    let adminInfo = await this.getAdminInfoByUserId(userId);
    if (!adminInfo) {
      throw new NotFoundException('User not found');
    }
    adminInfoFullUpdateDto = plainToClass(AdminInfoFullUpdateDto,
adminInfoFullUpdateDto, { excludeExtraneousValues: true });
    if (adminInfoFullUpdateDto.email) {
      const user = await this.usersService.findUserById(userId);
      user.email = adminInfoFullUpdateDto.email;
      await user.save();
      adminInfoUpdateResponse.email = user.email;
    }
    adminInfo = await this.adminInfoRepository.merge(adminInfo,
adminInfoFullUpdateDto);
    adminInfoUpdateResponse.adminInfo = await adminInfo.save();
    return adminInfoUpdateResponse;
  }
```

```
}
@Injectable()
export class AdminsInfoAccountService {
  validator = new Validator();

  constructor(
    private readonly adminInfoService: AdminsInfoService,
    private readonly userService: UsersService,
  ) {
  }

  async getAdminProfile(userId: number) {
    if (!userId || !Number(userId)) {
      throw new BadRequestException('Invalid token');
    }
    return plainToClass(AdminProfileResponseDto, await
this.adminInfoService.getAdminAllInfoByUserId(userId));
  }

  async updateAdminField(userId: number, changeAdminInfoFieldDto:
AdminInfoFieldUpdateDto) {
    const adminInfo = await this.adminInfoService.getAdminInfoByUserId(userId);

    if (!adminInfo) {
      throw new NotFoundException('User not found');
    }
    if (changeAdminInfoFieldDto.key === 'email') {
      if (!this.validator.isEmail(changeAdminInfoFieldDto.value)) {
        throw new BadRequestException('Value must be email');
      }
      const user = await this.userService.findUserById(userId);
      user.email = changeAdminInfoFieldDto.value;
      await user.save();
      return { userId, email: user.email };
    }
    if (typeof adminInfo[changeAdminInfoFieldDto.key] !== typeof
changeAdminInfoFieldDto.value
      && adminInfo[changeAdminInfoFieldDto.key] != null) {
      throw new BadRequestException('Field and value must have the same type');
    }

    adminInfo[changeAdminInfoFieldDto.key] = changeAdminInfoFieldDto.value;

    return await adminInfo.save();
  }

  async fullUpdateAdmin(userId: number, fullUpdateAdminInfoDto:
AdminInfoFullUpdateDto) {
    return this.adminInfoService.updateAdminInfo(userId, fullUpdateAdminInfoDto);
  }

  async updateAdminPhoto(userId: number, file: File):
Promise<UserPhotoUpdateResponseDto> {
    return await this.userService.updateUserPhoto(userId, file);
  }
}
Auth.service.ts
@Injectable()
export class AuthService {

  constructor(
    private readonly tokensService: TokensService,
    @InjectRepository(UsersRepository)
    private readonly usersRepository: UsersRepository,
    @InjectRepository(RefreshToken)
    private readonly refreshTokensRepository: Repository<RefreshToken>,
```

```
    private readonly passwordService: PasswordsService,
    private readonly keyValueStorageService: KeyValueStorageService,
    private readonly accountService: AccountsService,
    private readonly mailService: MailService,
  ) {
  }

  public async validateUser(email: string, pass: string): Promise<any> {
    const user = await this.usersRepository.findOne({
      where: [{ email, isDeleted: false, isActive: true }],
      select: ['password', 'id', 'email', 'roleId'],
    });
    if (!user || !await this.passwordService.comparePasswords(pass,
user.password)) {
      return false;
    }
    const { password, ...result } = user;
    return result;
  }

  public async login(user: any): Promise<TokenResponseDto> {
    const accessToken = this.tokensService.createJwtToken(user.id, user.email);
    const refreshToken = this.tokensService.createJwtToken(user.id, user.email);

    await this.tokensService.storeRefreshToken(refreshToken, user.id);

    return { accessToken, refreshToken } as TokenResponseDto;
  }

  async getAccessToken(token: string): Promise<any> {
    const refreshToken = await
this.refreshTokensRepository.createQueryBuilder('refreshToken')
      .innerJoinAndSelect('refreshToken.user', 'user')
      .where('refreshToken.token = :token', { token })
      .andWhere('user.isDeleted = false')
      .andWhere('user.isActive = true')
      .getOne();
    if (!refreshToken || !refreshToken.user) {
      throw new BadRequestException('Refresh token is invalid');
    }
    const accessToken = this.tokensService.createJwtToken(refreshToken.user.id,
refreshToken.user.email);
    return { accessToken } as TokenResponseDto;
  }

  public async resetPasswordStart(dto: StartResetPasswordDto): Promise<void> {
    const user = await this.usersRepository.createQueryBuilder('user')
      .where('user.email = :email', { email: dto.email })
      .andWhere('user.isDeleted = false')
      .andWhere('user.isActive = true')
      .getOne();
    if (!user) {
      throw new NotFoundException('User not found');
    }

    const code = randomString.generate({
      length: 8,
      capitalization: 'uppercase',
    });
    await Promise.all([
      this.keyValueStorageService.set(code, dto.email),
      this.mailService.send({
        subject: 'Recovery password',
        to: user.email,
        text: 'You code for recovery password: ' + code,
      }),
```

```
    ]);
    return;
  }

  public async checkResetPasswordCode(dto: CheckResetPasswordCodeDto):
Promise<boolean> {
    const code = await this.keyValueStorageService.get(dto.code);
    return Boolean(code);
  }

  public async resetPassword(dto: ResetPasswordDto): Promise<TokenResponseDto> {
    if (dto.newPassword !== dto.confirmPassword) {
      throw new BadRequestException('Passwords are not match');
    }
    const email = await this.keyValueStorageService.get(dto.code);
    if (!email) {
      throw new BadRequestException('Secret code is invalid');
    }

    const user = await this.usersRepository.findOne({ email });
    if (!user) {
      throw new NotFoundException('User not found');
    }

    await this.accountService.updatePassword(user.id, dto.newPassword);

    return this.login(user);
  }
}
Customer-info-document.service.ts
@Injectable()
export class CustomersInfoDocumentService {
  constructor(
    @InjectRepository(CustomersInfoRepository)
    private readonly customerInfoRepository: CustomersInfoRepository,
  ) {
  }

  async findCustomersByUserIds(userIds: number[], selections?: string[]) {
    if (userIds.length === 0) {
      return [];
    }
    return await this.customerInfoRepository.createQueryBuilder('customer-info')
      .select(selections)
      .where({ userId: In(userIds) })
      .getMany();
  }
}
Documents.service.ts
@Injectable()
export class DocumentsService {
  constructor(
    private readonly servicesService: ServicesService,
    private readonly documentsPhotosService: DocumentsPhotosService,
    private readonly connection: Connection,
    @InjectRepository(DocumentsRepository)
    private readonly documentsRepository: DocumentsRepository,
    private readonly serviceToUserService: ServicesToUsersService,
    private readonly usersService: UsersService,
    private readonly customerInfoDocumentService: CustomersInfoDocumentService,
    private readonly citiesService: CitiesService,
    private readonly countriesService: CountriesService,
    private readonly fileStorageService: FileStorageService,
    private readonly pushNotificationsService: PushNotificationsService,
  ) {
  }
```

```
  async saveDocument(entityManager: EntityManager, userId: number,
createDocumentDto: DocumentSaveDto, files: File[]) {
    await this.usersService.findUserById(userId);
    createDocumentDto.photosUrl = await this.savePhotos(files);
    const document = await this.documentsRepository.store(entityManager,
createDocumentDto, userId);
    const documentPhotos = await
this.documentsPhotosService.saveDocumentPhotos(entityManager, createDocumentDto,
document.id);
    return { document, documentPhotos } as DocumentSaveResponseDto;
  }

  async saveDocumentWithoutService(userId: number, createDocumentDto:
DocumentSaveDto, files: File[]): Promise<DocumentSaveResponseDto> {
    await this.servicesService.checkServiceExist(createDocumentDto.serviceId);
    return await this.connection.transaction(async entityManager => {
      return await this.saveDocument(entityManager, userId, createDocumentDto,
files);
    });
  }

  async saveDocumentWithService(entityManager: EntityManager, userId: number,
createDocumentDto: DocumentSaveDto, files: File[])
    : Promise<DocumentSaveResponseDto> {
    await this.servicesService.checkServiceExist(createDocumentDto.serviceId);
    return this.saveDocument(entityManager, userId, createDocumentDto, files);
  }

  async getDocumentsFromUser(userId: number, options: PaginationInterface):
Promise<DocumentsListResponseDto> {
    await this.usersService.findUserById(userId);
    const queryBuilder = await this.connection
      .getRepository(Document)
      .createQueryBuilder('document')
      .leftJoinAndSelect('document.documentPhotos', 'docPhoto')
      .where('document.userId = :userId', { userId })
      .andWhere('document.isDeleted = false');
    // console.log(await queryBuilder.getMany());
    return await paginate(queryBuilder, options);
  }

  async getAllDocuments(options: PaginationInterface, selections: string[],
                        filterDto?: DocumentsFilterDto):
Promise<DocumentsListResponseDto> {
    let queryBuilder = await
this.documentsRepository.createQueryBuilder('document').select(selections);
    if (filterDto) {
      if (filterDto.filterField === 'isActive') {
        if (filterDto.param == null) {
          queryBuilder = queryBuilder.andWhere('document.isActive is null');
        } else {
          queryBuilder = queryBuilder.andWhere('document.isActive = :status', {
status: filterDto.param });
        }
      } else if (filterDto.filterField === 'userFirstName') {
        queryBuilder = queryBuilder.innerJoin('document.user', 'user')
          .leftJoin('user.customerInfo', 'customer-info')
          .andWhere('customer-info.firstName = :name', { name: filterDto.param
});
      } else if (filterDto.filterField === 'userLastName') {
        queryBuilder = queryBuilder.innerJoin('document.user', 'user')
          .leftJoin('user.customerInfo', 'customer-info')
          .andWhere('customer-info.lastName = :name', { name: filterDto.param });
      } else if (filterDto.filterField === 'documentName') {
        queryBuilder = queryBuilder
```

```
          .andWhere('document.name = :name', { name: filterDto.param });
      } else if (filterDto.filterField === 'country') {
        queryBuilder = queryBuilder = queryBuilder.innerJoin('document.user',
'user')
          .leftJoin('user.customerInfo', 'customer-info');
        const citiesIds = await
this.citiesService.findCitiesIdsFromCountry(filterDto.param);
        queryBuilder = queryBuilder
          .andWhere('customer-info.cityId IN (:...ids)', { ids: citiesIds} );
      } else if (filterDto.filterField === 'service') {
        queryBuilder = queryBuilder = queryBuilder.innerJoin('document.service',
'service')
          .andWhere('service.key = :key', { key: filterDto.param });
      }
    }
    const result = await paginate(queryBuilder, options);
    const documents = result.items;
    if (documents.length === 0) {
      return result;
    }
    const users = await this.usersService.findUsersByIds(documents.map(document
=> document.userId), ['user.id']);
    const customers = await
this.customerInfoDocumentService.findCustomersByUserIds(users.map(user =>
user.id),
      ['customer-info.id', 'customer-info.firstName',
        'customer-info.lastName', 'customer-info.cityId', 'customer-
info.userId']);
    const cities = await
this.citiesService.findCitiesByIds(customers.map(customer => customer.cityId));
    // include countries into cities
    const countries = await
this.countriesService.findCountriesByIds(cities.map(city => city.countryId));
    const countriesObj = {};
    countries.map(it => {
      countriesObj[it.id] = it;
    });
    cities.map(city => {
      city.country = countriesObj[city.countryId];
    });
    // finish include countries into cities
    const citiesObj = {};
    cities.map(it => {
      citiesObj[it.id] = it;
    });
    customers.map(customer => {
      customer.city = citiesObj[customer.cityId];
    });
    // include customer-info into user
    const customersObj = {};
    customers.map(it => {
      customersObj[it.userId] = it;
    });
    users.map(user => {
      user.customerInfo = customersObj[user.id];
    });
    // include users into documents
    const usersObj = {};
    users.map(it => {
      usersObj[it.id] = it;
    });
    documents.map(document => {
      document.user = usersObj[document.userId];
    });
    result.items = documents;
    return result;
```

```
  }

  async approveDocument(documentUpdateStatusDto: DocumentUpdateStatusDto):
Promise<DocumentDto> {
    let document = await this.documentsRepository
      .findOne(documentUpdateStatusDto.documentId, { where: { isDelete: false }
});
    if (!document) {
      throw new NotFoundException('Document not found');
    }
    document.isActive = documentUpdateStatusDto.status;
    document = await document.save();
    await this.serviceToUserService.approveUserService(
      { serviceId: document.serviceId, userId: document.userId, status:
document.isActive },
    );
    const status = document.isActive === true ? 'accepted' : 'declined';
    await this.pushNotificationsService.sendNotification(document.userId,
      {
        sound: 'default',
        body: 'Your document was ' + status,
        data: { documentId: document.id, userId: document.userId },
      },
    );
    return plainToClass(DocumentDto, document);
  }

  async deleteDocument(documentId: number): Promise<DocumentDeleteDto> {
    const document = await this.documentsRepository.findOne(documentId);
    if (!document) {
      throw new NotFoundException('Document not found');
    }
    document.isDeleted = true;
    await this.serviceToUserService.approveUserService(
      { serviceId: document.serviceId, userId: document.userId, status: false },
    );
    await document.save();
    return { documentId: document.id, isDeleted: document.isDeleted } as
DocumentDeleteDto;
  }

  async savePhotos(files: File[]) {
    if (!files || files.length === 0) {
      throw new BadRequestException('Files is empty');
    }
    const photosUrls: string[] = [];
    for (const file of files) {
      photosUrls.push(await this.fileStorageService.saveImage(file));
      /*photosUrls.push(randomString.generate({
        length: 12,
        charset: 'alphabetic',
      }));*/
    }
    if (photosUrls.length === 0) {
      throw new BadRequestException('Files is empty');
    }
    return photosUrls;
  }

  async checkDocumentApproved(userId: number) {
    return await this.documentsRepository.createQueryBuilder('document')
      .where('document.userId =:userId', { userId })
      .andWhere('document.isActive = true')
      .getMany();
  }
}
```

Request.services.ts

```
@Injectable()
export class RequestsService {
  private badWords;
  private filter = new Filter();

  constructor(
    @InjectRepository(RequestsRepository)
    private readonly requestsRepository: RequestsRepository,
    @InjectRepository(RequestsFilesRepository)
    private readonly requestFileRepository: RequestsFilesRepository,
    private readonly usersService: UsersService,
    private readonly helpersService: HelpersService,
    private readonly connection: Connection,
    private readonly reportsService: ReportsService,
    private readonly serviceToUserService: ServicesToUsersService,
    private readonly requestStatusService: RequestsStatusService,
    private readonly fileStorageService: FileStorageService,
    private readonly requestGateway: RequestGateway,
    private readonly notificationsService: PushNotificationsService,
    private readonly citiesService: CitiesService,
    private readonly countriesService: CountriesService
  ) {
  }

  async createRequest(clientId: number, createRequestDto: RequestCreateDto,
files: File[]): Promise<RequestSaveResponseDto> {
    const status = await
this.requestStatusService.findRequestStatusByKey('hold');
    if (!status) {
      throw new NotFoundException('Request status was not found');
    }
    if (createRequestDto.performerId === clientId) {
      throw new BadRequestException(`Client can't be a performer`);
    }
    const client = await this.usersService.findUserById(clientId);
    await this.serviceToUserService.findUserService(createRequestDto.performerId,
createRequestDto.serviceId);
    this.badWords = await this.helpersService.getBadWords();
    this.filter.addWords(...this.badWords);
    createRequestDto.message = this.filter.clean(createRequestDto.message);
    return await this.connection.transaction(async entityManager => {
      const savedRequest = await this.requestsRepository.store(entityManager,
createRequestDto, clientId, status.id);
      let savedFiles = [];
      if (files && files.length !== 0) {
        createRequestDto.photosUrl = await this.savePhotos(files);
        savedFiles = await this.requestFileRepository.store(entityManager,
createRequestDto, savedRequest.id);
      }
      if (createRequestDto.message.includes('*')) {
        await this.reportsService.saveReport(entityManager, client.id,
          {
            title: 'Obscene word',
            parentTable: 'request',
            parentId: savedRequest.id,
            data: createRequestDto.message,
          } as ReportCreateDto);
      }

      await this.sendSocketOrNotification(savedRequest.performerId, 'newRequest',
        { message: 'send request to', performerId: savedRequest.performerId },
        {
          sound: 'default', body: 'Received request',
          data: { requestId: savedRequest.id, from: savedRequest.clientId },
        });
```

```
      return { request: savedRequest, requestPhotos: savedFiles } as
RequestSaveResponseDto;
    });
  }

  async findRequestById(id: number) {
    const request = await this.requestsRepository.findOne({ id });
    if (!request) {
      throw new NotFoundException('Request not found');
    }
    return request;
  }

  async getRequestsFromClient(userId: number, options: PaginationInterface,
selections: string[],
                              filterDto: RequestFilterDto):
Promise<RequestListResponseDto> {
    let queryBuilder = await
this.requestsRepository.createQueryBuilder('request').select(selections);
    if (filterDto.type === 'income') {
      queryBuilder = queryBuilder.where('request.performerId = :userId', { userId
})
        .innerJoin('request.client', 'user');
    } else if (filterDto.type === 'outgoing') {
      queryBuilder.where('request.clientId = :userId', { userId })
        .innerJoin('request.performer', 'user');
    }
    queryBuilder = queryBuilder
      .leftJoin('user.customerInfo', 'customer-info')
      //.leftJoin('customer-info.city', 'city')
      .leftJoin('request.service', 'service')
      .leftJoin('request.status', 'status')
      .leftJoin('user.serviceToUsers', 'serviceToUsers',
'serviceToUsers.serviceId = service.id');
    if (filterDto.filterField === 'status') {
      queryBuilder = queryBuilder
        .andWhere('status.key = :status', { status: filterDto.param });
    } else if (filterDto.filterField === 'service') {
      queryBuilder = queryBuilder
        .andWhere('service.key = :key', { key: filterDto.param });
    }
    const response = await paginate(queryBuilder, options);
    for (const item of response.items) {
      if (item.client) {
        const city: CityDto = await
this.citiesService.findCityById(item.client.customerInfo.cityId, 'ru');
        city.country = await
this.countriesService.findCountryById(city.countryId, 'ru');
        item.client.customerInfo.city = city;
      } else if (item.performer) {
        const city: CityDto = await
this.citiesService.findCityById(item.performer.customerInfo.cityId, 'ru');
        city.country = await
this.countriesService.findCountryById(city.countryId, 'ru');
        item.performer.customerInfo.city = city;
      }
    }
    return response;
  }

  async updateRequestStatus(userId: number, updateRequestStatusDto:
RequestUpdateStatusDto) {
    const requestStatus = await
this.requestStatusService.findRequestStatusByKey(updateRequestStatusDto.statusKey
);
    if (!requestStatus) {
```

```
      throw new NotFoundException('Request status not found');
    }
    let receiverUserId: number = 0;
    const eventKey: string = 'changeRequestStatus';
    let request = await this.findRequestById(updateRequestStatusDto.requestId);

    const socketInfo = { requestId: request.id, receiverUserId: 0, message:
'Change request status', statusKey: requestStatus.key };
    const notificationInfo: NotificationMessageDto = {
      sound: 'default', body: 'Change request status',
      data: { requestId: request.id, receiverUserId: 0, statusKey:
requestStatus.key },
    };

    if (requestStatus.key === 'progress' || requestStatus.key === 'decline') {
      if (request.performerId != userId) {
        throw new ForbiddenException('Forbid change status for this user');
      }
      receiverUserId = request.clientId;
      socketInfo.receiverUserId = receiverUserId;
      // @ts-ignore
      notificationInfo.data.receiverUserId = receiverUserId;
    }

    if (requestStatus.key === 'done') {
      if (request.clientId != userId) {
        throw new ForbiddenException('Forbid change status for this user');
      }
      receiverUserId = request.performerId;
      socketInfo.receiverUserId = receiverUserId;
      // @ts-ignore
      notificationInfo.data.receiverUserId = receiverUserId;
    }
    request.status = requestStatus;
    request = await request.save();
    await this.sendSocketOrNotification(receiverUserId, eventKey, socketInfo,
notificationInfo);
    return request;
  }

  async sendSocketOrNotification(userId: number, eventKey: string,
                                 socketInfo: object, notificationInfo:
NotificationMessageDto) {
    const room = 'request/' + userId;
    if (await this.requestGateway.isUserOnline(userId)) {
      await this.requestGateway.emitToRoom(room,
        eventKey, socketInfo);
    } else {
      try {
        await this.notificationsService.sendNotification(userId,
notificationInfo);
      } catch (e) {
        console.error(e.message);
      }
    }
  }

  private async savePhotos(files: File[]) {
    const photosUrls: string[] = [];
    for (const file of files) {
      photosUrls.push(await this.fileStorageService.saveImage(file));
      /*photosUrls.push(randomString.generate({
        length: 12,
        charset: 'alphabetic',
      }));*/
    }
```

```
    if (photosUrls.length === 0) {
      throw new BadRequestException('Files is empty');
    }
    return photosUrls;
  }
}
Review.service.ts
@Injectable()
export class ReviewsService {
  constructor(
    @InjectRepository(ReviewsRepository)
    private readonly reviewsRepository: ReviewsRepository,
    private readonly usersService: UsersService,
    private readonly usersRatingsService: UsersRatingsService,
    private readonly adminInfoService: AdminsInfoService,
    private readonly reviewsGateway: ReviewsGateway,
    private readonly notificationsService: PushNotificationsService,
  ) {
  }

  async saveReview(authorId: number, reviewSaveDto: ReviewSaveDto):
Promise<ReviewDto> {
    if (authorId == reviewSaveDto.userId) {
      throw new BadRequestException('Author can\'t be the same user');
    }
    await this.usersService.findUserById(authorId);
    const user = await this.usersService.findUserById(reviewSaveDto.userId);
    const reviewToSave: DeepPartial<Review> = {
      userId: user.id,
      text: reviewSaveDto.text,
      rating: reviewSaveDto.rating,
      authorId,
    };
    const rating = await this.usersRatingsService.findRatingByUserId(user.id);
    const review = await this.reviewsRepository.save(reviewToSave);
    await rating.updateRatingAndCount(review.rating);
    await rating.save();
    // await this.reviewsGateway.emitToRoom('review/' + review.userId,
'newComment',
    //   { message: 'send comment to', userId: review.userId });
    await this.sendSocketOrNotification(review.userId, 'newComment',
      { message: 'send comment to', userId: review.userId },
      {
        sound: 'default', body: 'New comment',
        data: {
          reviewId: review.id, from: review.authorId, to: review.userId,
        },
      });
    return plainToClass(ReviewDto, review);
  }

  async getReviewsByUserId(userId: number, options: PaginationInterface,
selections?: string[]): Promise<ReviewsListResponseDto> {
    const user = await this.usersService.findUserById(userId);
    const queryBuilder = await
this.reviewsRepository.createQueryBuilder('review')
      .where({ userId: user.id })
      .innerJoin('review.author', 'user')
      .innerJoin('user.customerInfo', 'customer')
      .select(selections)
      .orderBy('review.createDate', 'DESC');
    return await paginate(queryBuilder, options);
  }

  async getReviewsForUserProfile(userId: number, options: PaginationInterface,
                                 selections?: string[]): Promise<ReviewDto[]> {
```

```
    const queryBuilder = this.reviewsRepository.createQueryBuilder('review')
      .select(selections)
      .where({ userId });
    const reviews = await paginate(queryBuilder, options);
    return reviews.items;
  }

  async updateReview(userId: number, reviewUpdateDto: ReviewUpdateDto):
Promise<ReviewDto> {
    const user = await this.usersService.findUserById(userId);
    if (!user) {
      throw new NotFoundException('User not found');
    }
    const review = await
this.reviewsRepository.findOne(reviewUpdateDto.reviewId);
    if (!review) {
      throw new NotFoundException('Review not found');
    }
    if (reviewUpdateDto.field === 'text') {
      if (review.authorId !== user.id) {
        throw new ForbiddenException('Forbid for current user');
      }
      review.text = reviewUpdateDto.value.toString();
      return plainToClass(ReviewDto, await review.save());
    } else if (reviewUpdateDto.field === 'answer') {
      if (review.userId !== user.id) {
        throw new ForbiddenException('Forbid for current user');
      }
      review.answer = reviewUpdateDto.value.toString();
      await this.sendSocketOrNotification(review.authorId, 'newAnswer',
        { message: 'send comment to', authorId: review.authorId },
        {
          sound: 'default', body: 'New answer',
          data: {
            reviewId: review.id, from: review.userId, to: review.authorId,
          },
        });
      return plainToClass(ReviewDto, await review.save());
    } else if (reviewUpdateDto.field === 'rating') {
      if (review.authorId !== user.id) {
        throw new ForbiddenException('Forbid for current user');
      }
      if (!Number.isInteger(Number(reviewUpdateDto.value)) ||
        Number(reviewUpdateDto.value) > 5 || Number(reviewUpdateDto.value) < 1) {
        throw new BadRequestException('Rating value incorrect');
      }
      await this.usersRatingsService.updateRating(review.userId,
Number(reviewUpdateDto.value), review.rating);
      review.rating = Number(reviewUpdateDto.value);
      return plainToClass(ReviewDto, await review.save());
    }
  }

  async deleteReview(userId: number, reviewId: number) {
    const review = await this.reviewsRepository.findOne(reviewId);
    if (!review) {
      throw new NotFoundException('Review not found');
    }
    const admin = await this.adminInfoService.getAdminInfoByUserId(userId);
    if (userId !== review.authorId && !admin) {
      throw new ForbiddenException('Forbid for current user');
    }
    await this.usersRatingsService.downgradeRating(review.userId, review.rating);
    await review.remove();
  }
```

```
  async sendSocketOrNotification(userId: number, eventKey: string,
                                 socketInfo: object, notificationInfo:
NotificationMessageDto) {
    const room = 'review/' + userId;
    if (await this.reviewsGateway.isUserOnline(userId)) {
      await this.reviewsGateway.emitToRoom(room,
        eventKey, socketInfo);
    } else {
      try {
        await this.notificationsService.sendNotification(userId,
notificationInfo);
      } catch (e) {
        console.error(e.message);
      }
    }
  }
}
Statsitcs.services.ts
export class DocumentsStatisticsService {
  constructor(
    @InjectRepository(DocumentsRepository)
    private readonly documentsRepository: DocumentsRepository,
    private readonly statisticsService: StatisticsService,
  ) {
  }

  async amountOfDocumentsInReview(): Promise<AmountDto> {
    const key = keys.amountDocumentsInReview;
    const response = await this.statisticsService.getCash(key);
    if (response) {
      return response as AmountDto;
    } else {
      const amount = await
this.documentsRepository.createQueryBuilder('document')
        .where('document.isActive IS NULL')
        .getCount();
      await this.statisticsService.setCash(key, { amount });
      return { amount } as AmountDto;
    }
  }
}
export class RequestsStatisticsService {
  constructor(
    @InjectRepository(RequestsRepository)
    private readonly requestsRepository: RequestsRepository,
    private readonly statisticsService: StatisticsService,
  ) {
  }

  async getCountRequestEveryDay(amountOfRequestsByDate:
AmountOfRequestsByDateDto): Promise<AmountOfRequestsDto[]> {
    const key = keys.amountOfRequestsByTime + '-' +
amountOfRequestsByDate.interval
      + '-' + amountOfRequestsByDate.filterBy + '-' +
amountOfRequestsByDate.param;
    const response = await this.statisticsService.getCash(key);
    if (response) {
      console.log(response);
      return response;
    } else {
      const result = await this.requestsRepository.createQueryBuilder('request')
        .select(`EXTRACT(${amountOfRequestsByDate.interval}\n` +
          `               FROM request."createDate") AS
${amountOfRequestsByDate.interval},\n` +
          '       count(request) AS amountOfRequests,\n' +
          '       "request"."createDate"')
```

```
        .groupBy(`EXTRACT(${amountOfRequestsByDate.interval}\n` +
          '                 FROM request."createDate"),  "request"."createDate"')
        .orderBy('request.createDate', 'ASC')
        .where(`EXTRACT(${amountOfRequestsByDate.filterBy} FROM
request."createDate") = :param`,
          { param: amountOfRequestsByDate.param })
        .getRawMany();
      await this.statisticsService.setCash(key, result);
      return result;
    }
  }

  async getRequestsByTime(statisticDateBetweenDto: StatisticDateBetweenDto,
                          options: PaginationInterface):
Promise<RequestsByTimeResponseDto> {
    const key = keys.requestsByTime;
    const response = await this.statisticsService.getCash(key,
statisticDateBetweenDto, options);
    if (response) {
      console.log(response);
      return response as RequestsByTimeResponseDto;
    } else {
      const qb = await this.requestsRepository.createQueryBuilder('request')
        .where('request.createDate BETWEEN (:afterDate) AND (:beforeDate)',
          { afterDate: statisticDateBetweenDto.afterDate, beforeDate:
statisticDateBetweenDto.beforeDate });
      const result = await paginate(qb, options);
      await this.statisticsService.setCash(key
        , result.items.length === 0 ? null : result, statisticDateBetweenDto,
options);
      return result;
    }
  }
}
export class ServicesStatisticService {
  constructor(
    @InjectRepository(ServicesRepository)
    private readonly servicesRepository: ServicesRepository,
    private readonly statisticService: StatisticsService,
  ) {
  }

  async getAmountUsersInServices(): Promise<AmountServicesDto[]> {
    const key = keys.amountOfUsersInServices;
    const response = await this.statisticService.getCash(key);
    if (response) {
      return response;
    } else {
      const amount = await this.servicesRepository.createQueryBuilder('service')
        .select('service.name, count(user)')
        .innerJoin('service.serviceToUsers', 'serviceToUser')
        .innerJoin('serviceToUser.user', 'user')
        .where('user.isDeleted = false')
        .andWhere('user.isActive = true')
        .groupBy('service.name')
        .getRawMany();
      await this.statisticService.setCash(key, { amount });
      return amount;
    }
  }

  async getAmountRequestsInServices(): Promise<AmountServicesDto[]> {
    const key = keys.amountOfRequestsInServices;
    const response = await this.statisticService.getCash(key);
    if (response) {
      return response;
```

```
    } else {
      const amount = await this.servicesRepository.createQueryBuilder('service')
        .select('service.name, count(request)')
        .innerJoin('service.requests', 'request')
        .groupBy('service.name')
        .getRawMany();
      await this.statisticService.setCash(key, { amount });
      return amount;
    }
  }
}
@Injectable()
export class StatisticsService {
  constructor(
    private readonly keyValueStorageService: KeyValueStorageService,
  ) {
  }

  public async setCash(prefix: string, dtoData: object, dtoFilter?:
StatisticDateBetweenDto,
                       options?: PaginationInterface)
    : Promise<void> {
    await this.keyValueStorageService.set(
      this.makeRedisKey(prefix, dtoFilter, options),
      dtoData,
    );
  }

  public async getCash(prefix: string, dto?: StatisticDateBetweenDto,
                       options?: PaginationInterface) {
    const cashedData = await
this.keyValueStorageService.get(this.makeRedisKey(prefix, dto, options));
    return cashedData ? cashedData : null;
  }

  private makeRedisKey(prefix: string, dto?: StatisticDateBetweenDto,
                       options?: PaginationInterface): string {
    if (options) {
      prefix = options.page ? prefix + 'Page' + options.page : prefix;
      prefix = options.limit ? prefix + 'Limit' + options.limit : prefix;
      prefix = options.sort ? prefix + 'Sort' + options.sort : prefix;
      prefix = options.sortField ? prefix + 'SortField' + options.sortField :
prefix;
    }
    return dto
      ? `${prefix}/${dto.afterDate}-${dto.beforeDate}`
      : prefix;
  }
}
```

Репозиторії

Admin-info.repository.ts

```
@EntityRepository(AdminInfo)
export class AdminsInfoRepository extends Repository<AdminInfo> {

  public async store(manager: EntityManager, dto: UserCreateDto, userId: number):
Promise<AdminInfo> {

    const toSave: DeepPartial<AdminInfo> = {
      firstName: dto.firstName,
      userId,
    };

    const insertResult = await manager.insert(AdminInfo, toSave);
    return await manager.findOne(AdminInfo, insertResult.identifiers[0].id);
  }
}
```

```
Customer-info.repository.ts
@EntityRepository(CustomerInfo)
export class CustomersInfoRepository extends Repository<CustomerInfo> {

  public async store(manager: EntityManager, createUserDto: UserCreateDto,
userId: number): Promise<CustomerInfo> {

    const toSave: DeepPartial<CustomerInfo> = {
      firstName: createUserDto.firstName,
      lastName: createUserDto.lastName,
      cityId: createUserDto.cityId,
      longitude: createUserDto.longitude,
      latitude: createUserDto.latitude,
      userId,
      addressLine: createUserDto.addressLine ? createUserDto.addressLine : null,
    };

    const insertResult = await manager.insert(CustomerInfo, toSave);
    return await manager.findOne(CustomerInfo, insertResult.identifiers[0].id);
  }
}

Docuemnts.repository.ts
@EntityRepository((Document))
export class DocumentsRepository extends Repository<Document> {
  public async store(manager: EntityManager, createDocumentDto: DocumentSaveDto,
userId: number): Promise<Document> {

    const documentToSave: DeepPartial<Document> = {
      userId,
      name: createDocumentDto.name,
      description: createDocumentDto.description,
      serviceId: createDocumentDto.serviceId,
    };
    const insertResult = await manager.insert(Document, documentToSave);
    return await manager.findOne(Document, insertResult.identifiers[0].id);
  }
}

Reports.repository.ts
@EntityRepository((Report))
export class ReportsRepository extends Repository<Report> {
  public async store(manager: EntityManager, createReportDto: ReportCreateDto,
userId: number): Promise<Report> {

    const reportToSave: DeepPartial<Report> = {
      title: createReportDto.title,
      parentTable: createReportDto.parentTable,
      parentId: createReportDto.parentId,
      data: createReportDto.data,
      userId,
    };
    const insertResult = await manager.insert(Report, reportToSave);
    return await manager.findOne(Report, insertResult.identifiers[0].id);
  }
}

Requests.repository.ts
@EntityRepository(Request)
export class RequestsRepository extends Repository<Request> {
  public async store(manager: EntityManager, createRequestDto: RequestCreateDto,
                     clientId: number, statusId: number): Promise<Request> {
    const requestToSave: DeepPartial<Request> = {
      clientId,
      performerId: createRequestDto.performerId,
      serviceId: createRequestDto.serviceId,
```

```
      message: createRequestDto.message,
      clientPhone: createRequestDto.clientPhone,
      statusId,
    };
    const insertResult = await manager.insert(Request, requestToSave);
    return await manager.findOne(Request, insertResult.identifiers[0].id);
  }
}
```

Контроллери

Admin-info.controller.ts

```
@ApiTags('Admins general')
@Controller('admins')
export class AdminsInfoController {
  constructor(
    private readonly adminService: AdminsInfoService,
  ) {
  }

  @ApiBody({ type: UserCreateDto })
  @ApiResponse({ status: 201, description: 'The admin-user has been successfully
created.', type: TokenResponseDto })
  @ApiResponse({ status: 400, description: 'Bad request' })
  @ApiResponse({ status: 409, description: 'User already exists' })
  @Post('registration')
  async registration(@Body() createUserDto: UserCreateDto) {
    return await this.adminService.save(createUserDto);
  }

  @ApiBearerAuth()
  @ApiPagination()
  @ApiForbiddenResponse({ description: 'Forbidden for this user' })
  @ApiResponse({ status: 200, description: 'Returned admin-info list', type:
AdminsListResponseDto })
  @UseGuards(AuthGuard('jwt'), RoleGuard)
  @Roles('admin')
  @Get()
  async getAllAdmins(@Req() req, @Pagination() pagination) {
    return await this.adminService.getAllAdmins(pagination);
  }
}
```

Admin-info.account.controller.ts

```
@ApiBearerAuth()
@ApiForbiddenResponse({ description: 'Forbidden for this user' })
@ApiTags('Admins account')
@UseGuards(AuthGuard('jwt'), RoleGuard)
@Controller('admins/account')
export class AdminsInfoAccountController {
  constructor(private readonly adminInfoAccountService: AdminsInfoAccountService)
{
  }

  @ApiResponse({ status: 404, description: 'Admin-info profile not found' })
  @ApiResponse({ status: 400, description: 'Invalid token' })
  @ApiResponse({ status: 200, description: 'Return admin-info profile', type:
AdminProfileResponseDto })
  @Roles('admin')
  @Get()
  async getAdminProfile(@Req() req) {
    return await this.adminInfoAccountService.getAdminProfile(req.user.id);
  }

  @ApiResponse({ status: 404, description: 'Admin-info profile not found' })
  @ApiResponse({ status: 200, description: 'Return admin-info profile', type:
AdminProfileResponseDto })
  @Roles('admin')
```

```
  @Get(':id')
  async getAdminProfileById(@Param('id') userId: number) {
    return await this.adminInfoAccountService.getAdminProfile(userId);
  }

  @ApiBody({ type: AdminInfoFieldUpdateDto })
  @ApiResponse({ status: 404, description: 'User not found' })
  @ApiResponse({ status: 400, description: 'Value must be email' })
  @ApiResponse({ status: 200, description: 'Admin-info field has been updated',
type: AdminInfoDto })
  @Roles('admin')
  @Patch()
  changeAdminField(@Req() req, @Body() changeAdminInfoFieldDto:
AdminInfoFieldUpdateDto) {
    return this.adminInfoAccountService.updateAdminField(req.user.id,
changeAdminInfoFieldDto);
  }

  @ApiBody({ type: AdminInfoFullUpdateDto })
  @ApiResponse({ status: 404, description: 'User not found' })
  @ApiResponse({ status: 400, description: 'Bad request' })
  @ApiResponse({ status: 200, description: 'Admin-info has been updated', type:
AdminInfoFullUpdateResponseDto })
  @Roles('admin')
  @Patch('full')
  fullUpdateAdmin(@Req() req, @Body() fullUpdateAdminInfoDto:
AdminInfoFullUpdateDto) {
    return this.adminInfoAccountService.fullUpdateAdmin(req.user.id,
fullUpdateAdminInfoDto);
  }

  @ApiConsumes('multipart/form-data')
  @ApiBody({ type: FileUploadDto })
  @ApiResponse({ status: 404, description: 'User not found/File not found' })
  @ApiResponse({ status: 400, description: 'Bad request' })
  @ApiResponse({ status: 200, description: 'Admin-info photo has been updated',
type: UserPhotoUpdateResponseDto })
  @UseInterceptors(FileInterceptor('image'))
  @Roles('admin')
  @Patch('photo')
  updateAdminPhoto(@Req() req, @UploadedFile() file):
Promise<UserPhotoUpdateResponseDto> {
    return this.adminInfoAccountService.updateAdminPhoto(req.user.id, file);
  }
}
Auth.controller.java
@ApiTags('Authorization')
@Controller('auth')
export class AuthController {
  constructor(private readonly authService: AuthService) {
  }

  @ApiBody({ type: LoginDto })
  @ApiResponse({ status: 201, description: 'The new access token has been
successfully created.', type: TokenResponseDto })
  @ApiResponse({ status: 401, description: 'Email or password is incorrect' })
  @UseGuards(AuthGuard('local'))
  @Post('login')
  login(@Request() req): Promise<TokenResponseDto> {
    return this.authService.login(req.user);
  }

  @ApiBody({ type: RefreshTokenDto })
  @ApiResponse({ status: 201, description: 'The new access token has been
successfully created.', type: TokenResponseDto })
  @ApiResponse({ status: 400, description: 'Refresh token is invalid' })
```

```
  @Post('getAccessToken')
  async getAccessToken(@Body() req) {
    return await this.authService.getAccessToken(req.refreshToken);
  }

  @ApiResponse({ status: 200, description: 'Secret code has been sent to email'
})
  @ApiResponse({ status: 404, description: 'User not found' })
  @Post('/reset-password-start')
  resetPasswordStart(@Body() dto: StartResetPasswordDto): Promise<void> {
    return this.authService.resetPasswordStart(dto);
  }

  @ApiResponse({ status: 200, description: 'Secret code is true/false', type:
Boolean })
  @Post('/reset-password-check')
  resetPasswordCheckCode(@Body() dto: CheckResetPasswordCodeDto):
Promise<boolean> {
    return this.authService.checkResetPasswordCode(dto);
  }

  @ApiResponse({ status: 201, description: 'User password has been changed',
type: TokenResponseDto })
  @ApiResponse({ status: 400, description: 'Passwords are not match/Secret code
is invalid' })
  @ApiResponse({ status: 404, description: 'User not found' })
  @Post('/reset-password')
  resetPassword(@Body() dto: ResetPasswordDto): Promise<TokenResponseDto> {
    return this.authService.resetPassword(dto);
  }
}
Customer-info-admin.controller.ts
const usersListSelections = ['customer-info.firstName', 'customer-info.lastName',
  'customer-info.id', 'customer-info.userId',
  'customer-info.cityId', 'customer-info.longitude',
  'customer-info.latitude'];

@ApiBearerAuth()
@ApiForbiddenResponse({ description: 'Forbidden for this user' })
@ApiTags('Customer admin operations')
@UseGuards(AuthGuard('jwt'), RoleGuard)
@Controller('admin/customers')
export class CustomersInfoAdminController {
  constructor(
    private readonly customerInfoService: CustomersInfoService,
  ) {
  }

  @ApiBody({ type: CustomerStatusUpdateDto })
  @ApiResponse({ status: 404, description: 'User not found' })
  @ApiResponse({ status: 200, description: 'Customer status has been updated',
type: CustomerStatusUpdateDto })
  @Roles('admin')
  @Patch('')
  updateCustomerStatus(@Req() req, @Body() updateCustomerStatusDto:
CustomerStatusUpdateDto): Promise<CustomerStatusUpdateDto> {
    return
this.customerInfoService.changeCustomerStatus(updateCustomerStatusDto);
  }

  @ApiResponse({ status: 404, description: 'User not found' })
  @ApiResponse({ status: 200, description: 'Customer has been deleted', type:
CustomerDeleteDto })
  @Roles('admin')
  @Delete(':id')
  deleteCustomer(@Param('id') userId: number): Promise<CustomerDeleteDto> {
```

```
    return this.customerInfoService.deleteCustomer(userId);
  }

  @ApiBody({ type: CustomerFilterArrayDto })
  @ApiPagination()
  @ApiResponse({ status: 200, description: 'Returned customers list for admin
panel', type: CustomerCardsDto })
  @Roles('admin')
  @Get('')
  async getAllCustomers(@Req() req, @Pagination() paginateOptions,
                        @Body() filtersDto?: CustomerFilterArrayDto):
Promise<CustomersProfilesListResponseDto> {
    return await
this.customerInfoService.getAllCustomersForAdminPanel(paginateOptions,
usersListSelections, filtersDto);
  }
}
Customer-info-application.controller.ts
const usersListSelections = ['customer-info.firstName', 'customer-info.lastName',
  'customer-info.id', 'customer-info.userId', 'customer-info.description',
  'customer-info.latitude', 'customer-info.longitude',
  'user.isDeleted', 'user.isActive', 'user.photoUrl',
  'user.id', 'rating.id', 'rating.rating',
  'rating.ratesCount', 'rating.userId'];

@ApiTags('Customers application')
@Controller('app/customers')
export class CustomersInfoApplicationController {
  constructor(
    private readonly customerService: CustomersInfoService,
  ) {
  }

  @ApiConsumes('multipart/form-data')
  @ApiBody({ type: UserCreateDto })
  @ApiResponse({ status: 201, description: 'The customer-user has been
successfully created.', type: TokenResponseDto })
  @ApiResponse({ status: 400, description: 'Passwords are not match/Incorrect
phone values' })
  @ApiResponse({ status: 404, description: 'Service not found' })
  @ApiResponse({ status: 409, description: 'User with such email has already
exist' })
  @UseInterceptors(FileInterceptor('image'))
  @Post('registration')
  async registration(@Body() createUserDto: UserCreateDto, @UploadedFile() file):
Promise<TokenResponseDto> {
    /*let createUserDto;
    try {
      createUserDto = plainToClass(UserCreateDto, JSON.parse(str.str));
      await validateOrReject(createUserDto);
    } catch (e) {
      throw new BadRequestException(`Post values incorrect:
${JSON.stringify(e)}`);
    }*/
    return await this.customerService.save(createUserDto, file);
  }

  @ApiBearerAuth()
  @ApiForbiddenResponse({ description: 'Forbidden for this user' })
  @ApiResponse({ status: 404, description: 'Customer profile not found' })
  @ApiResponse({ status: 200, description: 'Return customer profile', type:
CustomerProfileResponseDto })
  @UseGuards(AuthGuard('jwt'), RoleGuard)
  @Roles('customer')
  @Get(':id')
  async getCustomerProfile(@Req() req, @Param('id') userId: number)
```

```
    : Promise<CustomerProfileResponseDto> {
    return await this.customerService.getCustomerProfile(userId, req.user.id);
  }

  @ApiBody({ type: CustomerFilterArrayDto })
  @ApiPagination()
  @ApiResponse({
    status: 200, description: 'Returned customers list for app panel',
    type: CustomersProfilesListResponseDto,
  })
  @UseGuards(AuthGuard('jwt'))
  @Get('')
  async getAllCustomers(@Req() req, @Pagination() paginateOptions,
                        @Body() filtersDto?: CustomerFilterArrayDto):
Promise<CustomersProfilesListResponseDto> {
    return await this.customerService.getAllCustomersForAppMainPage(req.user.id,
paginateOptions, usersListSelections, filtersDto);
  }

  // @ApiResponse({ status: 404, description: 'User not found' })
  // @ApiResponse({ status: 200, description: 'Return customer profile', type:
CustomerProfileResponseDto })
  // @Get(':id')
  // async getCustomerProfileForRequest(@Param('id') userId: number) {
  //   return await this.customerService.getCustomerInfoForCreateRequest(userId);
  // }
}
Document-admin.controller.ts
export class DocumentsAdminController {
  constructor(private readonly documentsService: DocumentsService) {
  }

  @ApiBody({ type: DocumentUpdateStatusDto })
  @ApiResponse({ status: 404, description: 'Document not found' })
  @ApiResponse({ status: 200, description: 'Return approved document', type:
DocumentDto })
  @Roles('admin')
  @Patch()
  async approveDocument(@Body() documentUpdateStatusDto: DocumentUpdateStatusDto)
{
    return await this.documentsService.approveDocument(documentUpdateStatusDto);
  }

  @ApiPagination()
  @ApiResponse({ status: 400, description: 'User not found' })
  @ApiResponse({ status: 200, description: 'Returned documents list by user id',
type: DocumentsListResponseDto })
  @Roles('admin')
  @Get('user/:id')
  async getAllDocumentsByUserId(@Param('id') userId: number, @Pagination()
pagination) {
    return await this.documentsService.getDocumentsFromUser(userId, pagination);
  }

  @ApiPagination()
  @ApiResponse({ status: 200, description: 'Returned documents list', type:
DocumentsListResponseDto })
  @Roles('admin')
  @Get('')
  async getAllDocuments(@Pagination() pagination, @Body() filterDto?:
DocumentsFilterDto) {
    return await this.documentsService.getAllDocuments(pagination,
selectDocuments, filterDto);
  }

  @ApiResponse({ status: 404, description: 'Document not found' })
```

```
  @ApiResponse({ status: 200, description: 'Document has been deleted', type:
DocumentDeleteDto })
  @Roles('admin')
  @Delete(':id')
  async deleteDocument(@Param('id') documentId: number) {
    return await this.documentsService.deleteDocument(documentId);
  }
}
Document-customer.controller.ts
@UseGuards(AuthGuard('jwt'))
@Controller('customer/documents')
export class DocumentsCustomerController {
  constructor(private readonly documentsService: DocumentsService) {
  }

  @ApiConsumes('multipart/form-data')
  @ApiBody({ type: DocumentSaveDto })
  @ApiResponse({ status: 404, description: 'User not found/Service not found' })
  @ApiResponse({ status: 400, description: 'Files is empty' })
  @ApiResponse({ status: 200, description: 'Document has been saved
successfully', type: DocumentSaveResponseDto })
  @UseInterceptors(FilesInterceptor('files'))
  @Roles('customer')
  @Post()
  async createDocument(@UploadedFiles() files, @Req() req, @Body()
createDocumentDto: DocumentSaveDto) {
    // let createDocumentDto: DocumentSaveDto;
    // try {
    //   createDocumentDto = plainToClass(DocumentSaveDto, JSON.parse(str.str));
    //   await validateOrReject(createDocumentDto);
    // } catch (e) {
    //   throw new BadRequestException(`Post values incorrect:
${JSON.stringify(e)}`);
    // }
    return this.documentsService.saveDocumentWithoutService(req.user.id,
createDocumentDto, files);
  }

  @ApiPagination()
  @ApiResponse({ status: 200, description: 'Returned documents list by user id',
type: DocumentsListResponseDto })
  @ApiResponse({ status: 404, description: 'User not found' })
  @Roles('customer')
  @Get(':id')
  async getAllDocumentsByUserId(@Param('id') userId: number, @Pagination()
pagination) {
    return await this.documentsService.getDocumentsFromUser(userId, pagination);
  }

  @ApiResponse({ status: 404, description: 'Document not found' })
  @ApiResponse({ status: 200, description: 'Document has been deleted', type:
DocumentDeleteDto })
  @Roles('customer')
  @Delete(':id')
  async deleteDocument(@Param('id') documentId: number) {
    return await this.documentsService.deleteDocument(documentId);
  }
}
Services.controller.ts
  addNewService(@Body() saveServiceDto: ServiceSaveDto): Promise<ServiceDto> {
    return this.servicesService.saveService(saveServiceDto);
  }

  @ApiResponse({ status: 400, description: 'Service not found' })
  @ApiResponse({ status: 200, description: 'Return service', type: ServiceDto })
  @Get(':id')
```

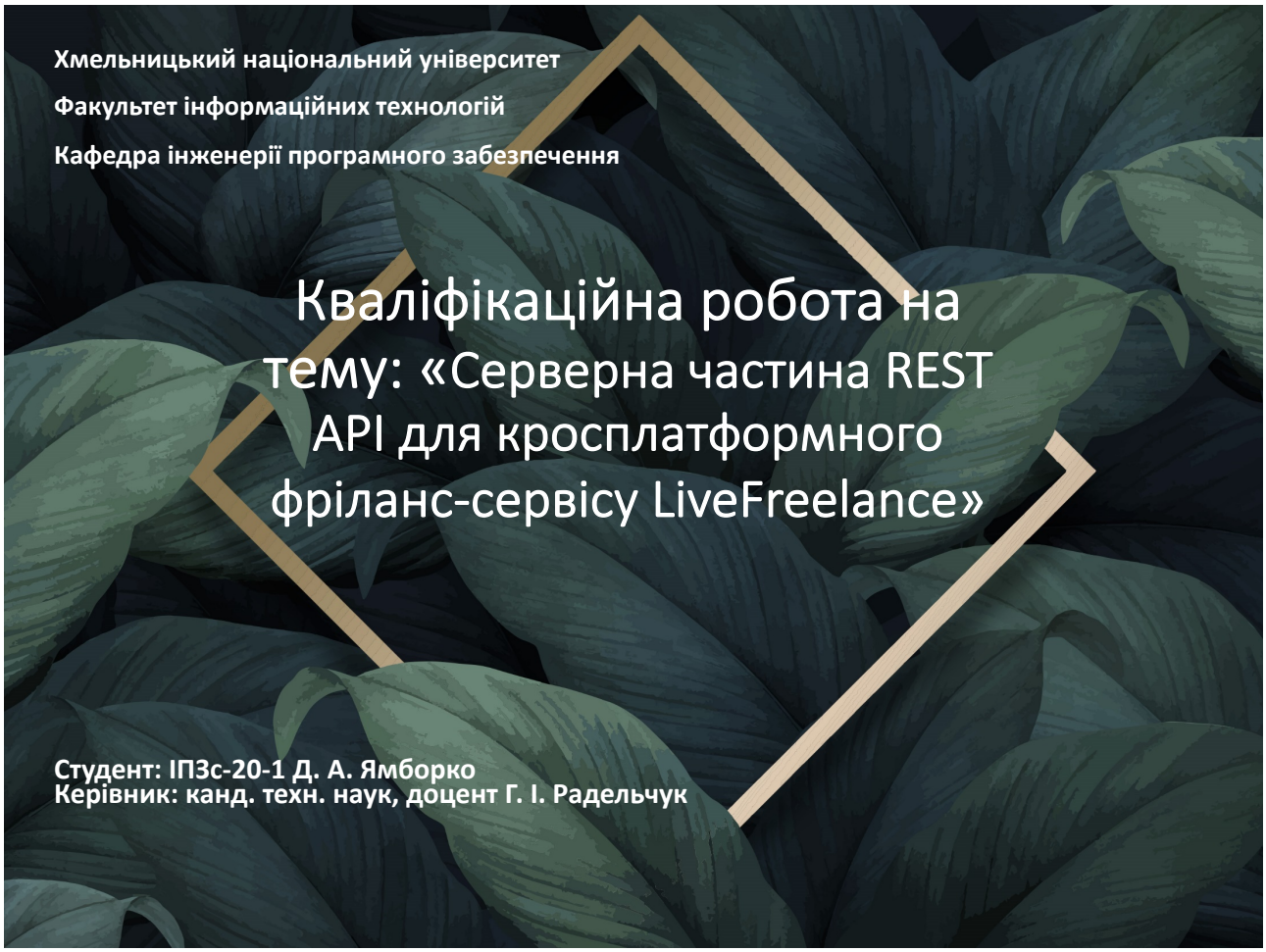
```
  getService(@Param('id') serviceId: number): Promise<ServiceDto> {
    return this.servicesService.getServicesById(serviceId);
  }

  @ApiBearerAuth()
  @ApiForbiddenResponse({ description: 'Forbidden for this user' })
  @ApiBody({ type: ServiceUpdateDto })
  @ApiResponse({ status: 400, description: 'Service not found' })
  @ApiResponse({ status: 200, description: 'Service has been updated
successfully', type: ServiceDto })
  @UseGuards(AuthGuard('jwt'), RoleGuard)
  @Roles('admin')
  @Put(':id')
  updateService(@Param('id') serviceId: number,
                @Body() updateServiceDto: ServiceUpdateDto): Promise<ServiceDto>
{
    return this.servicesService.updateService(serviceId, updateServiceDto);
  }

  @ApiBearerAuth()
  @ApiForbiddenResponse({ description: 'Forbidden for this user' })
  @ApiResponse({ status: 404, description: 'Service not found' })
  @ApiResponse({ status: 200, description: 'Service has been deleted' })
  @UseGuards(AuthGuard('jwt'), RoleGuard)
  @Roles('admin')
  @Delete(':id')
  deleteService(@Param('id') serviceId: number) {
    return this.servicesService.deleteService(serviceId);
  }

  @ApiPagination()
  @ApiResponse({ status: 200, description: 'Returned services list', type:
ServicesListResponseDto })
  @Get('')
  async getAllServices(@Pagination() pagination):
Promise<ServicesListResponseDto> {
    return await this.servicesService.getAll(pagination);
  }
}
Statistics.controller.ts
@ApiBearerAuth()
@ApiForbiddenResponse({ description: 'Forbidden for this user' })
@UseGuards(AuthGuard('jwt'), RoleGuard)
@ApiTags('Statistics')
@Controller('statistics')
export class StatisticsController {
  constructor(
    private readonly usersStatisticsService: UsersStatisticsService,
    private readonly documentsStatisticsService: DocumentsStatisticsService,
    private readonly requestStatisticsService: RequestsStatisticsService,
    private readonly servicesStatisticService: ServicesStatisticService,
  ) {
  }

  @ApiResponse({ status: 200, description: 'Return amount of active user', type:
AmountDto })
  @Roles('admin')
  @Get('users/activeAmount')
  getAmountOfActiveUsers(): Promise<AmountDto> {
    return this.usersStatisticsService.amountOfActiveCustomers();
  }

  @ApiBody({ type: StatisticDateBetweenDto })
  @ApiPagination()
  @ApiResponse({
    status: 200, description: 'Return new users for time period',
```

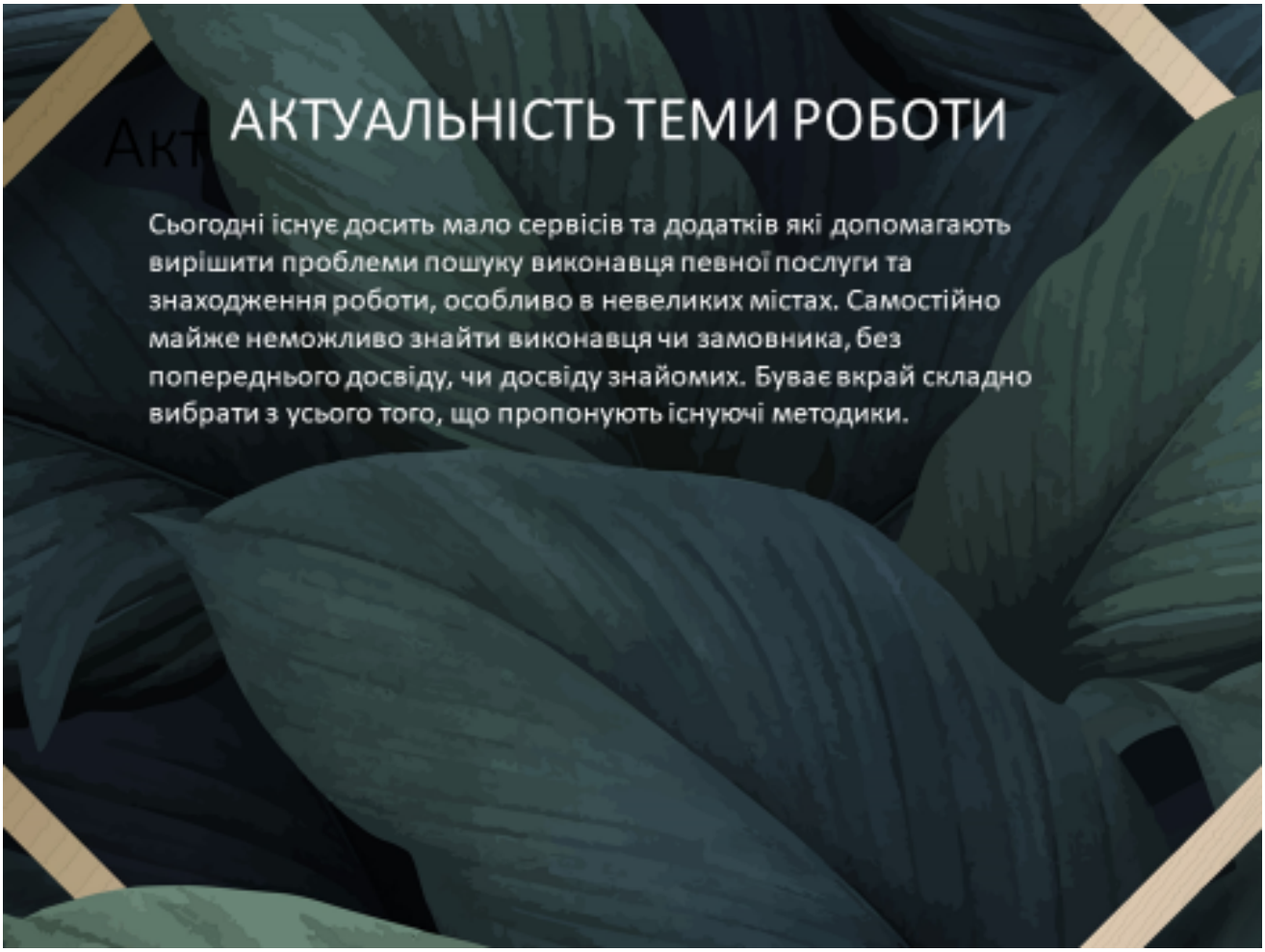
```
    type: CustomerInfoDto, isArray: true,
  })
  @Roles('admin')
  @Get('users/newUsers')
  getNewUsersByTime(@Body() statisticDateBetweenDto: StatisticDateBetweenDto,
                    @Pagination() options)
    : Promise<NewUsersResponseDto> {
    return
this.usersStatisticsService.newCustomersByTime(statisticDateBetweenDto, options);
  }

  @ApiResponse({ status: 200, description: 'Return amount of documents in
review', type: AmountDto })
  @Roles('admin')
  @Get('documents/inReview')
  getAmountOfDocumentsInReview(): Promise<AmountDto> {
    return this.documentsStatisticsService.amountOfDocumentsInReview();
  }

  @ApiBody({ type: AmountOfRequestsByDateDto })
  @ApiResponse({
    status: 200, description: 'Return amount of requests every day',
    type: AmountOfRequestsDto, isArray: true,
  })
  @Roles('admin')
  @Get('requests/byTime')
  getRequestsAmountEveryDay(@Body() amountOfRequestsByDate:
AmountOfRequestsByDateDto) {
    return
this.requestStatisticsService.getCountRequestEveryDay(amountOfRequestsByDate);
  }

  @ApiBody({ type: StatisticDateBetweenDto })
  @ApiPagination()
  @ApiResponse({
    status: 200, description: 'Return requests for time period',
    type: CustomerInfoDto, isArray: true,
  })
  @Roles('admin')
  @Get('requests/forTimePeriod')
  getRequestsForTimePeriod(@Body() statisticDateBetweenDto:
StatisticDateBetweenDto,
                           @Pagination() options)
    : Promise<RequestsByTimeResponseDto> {
    return
this.requestStatisticsService.getRequestsByTime(statisticDateBetweenDto,
options);
  }

  @ApiResponse({
    status: 200, description: 'Return amount of users in each service',
    type: AmountServicesDto, isArray: true,
  })
  @Roles('admin')
  @Get('services/users')
  getAmountOfUsersInServices(): Promise<AmountServicesDto[]> {
    return this.servicesStatisticService.getAmountUsersInServices();
  }

  @ApiResponse({
    status: 200, description: 'Return amount of requests in each service',
    type: AmountServicesDto, isArray: true,
  })
  @Roles('admin')
  @Get('services/requests')
  getAmountOfRequestsInServices(): Promise<AmountServicesDto[]> {
```

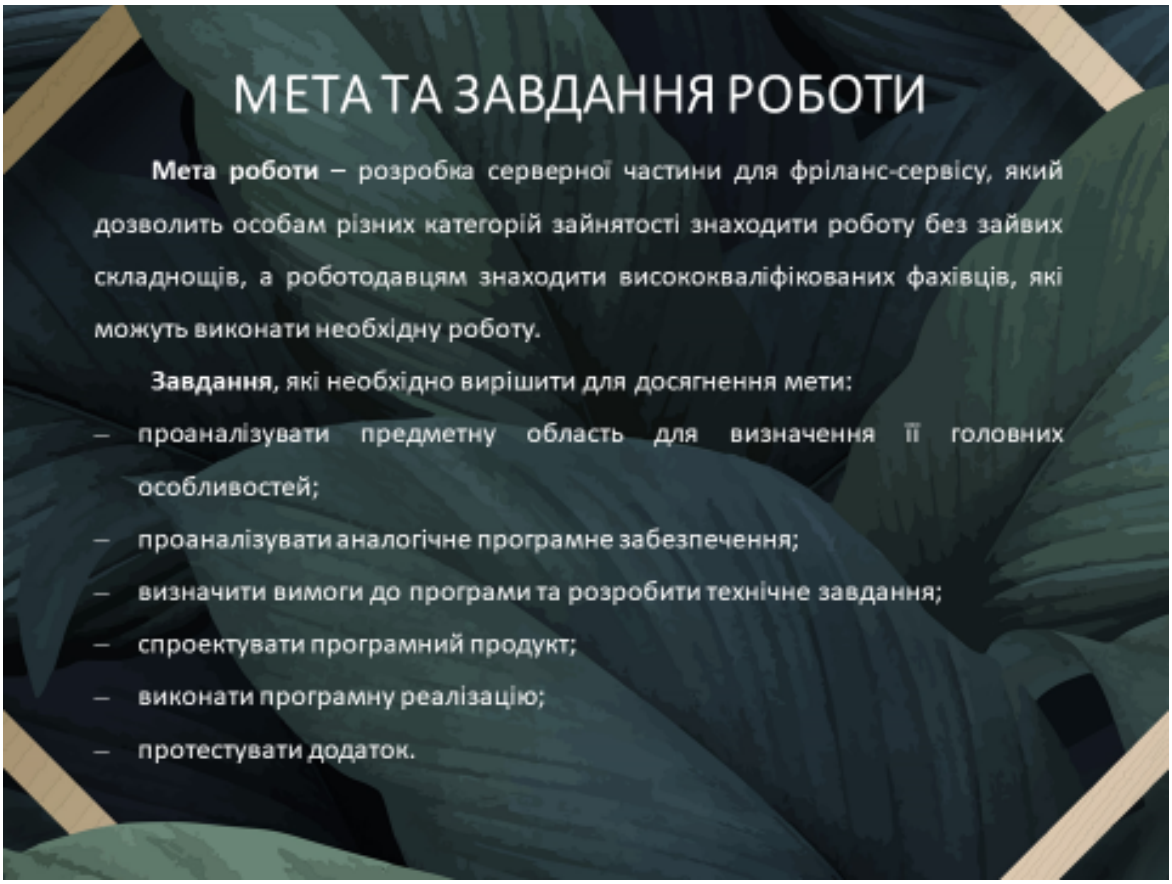
```
    return this.servicesStatisticService.getAmountRequestsInServices();
  }

  @ApiBody({ type: UsersByLocationStatisticDto })
  @ApiPagination()
  @ApiResponse({
    status: 200, description: 'Return amount of users in each city/country',
    type: AmountOfUsersFromCityResponseDto,
  })
  @Roles('admin')
  @Get('location/users')
  getAmountOfUsersInCities(@Body() usersByLocationStatisticDto:
UsersByLocationStatisticDto,
                           @Pagination() options):
Promise<AmountOfUsersFromCityResponseDto> {
    return
this.usersStatisticsService.amountOfUsersFromLocation(usersByLocationStatisticDto
, options);
  }
```

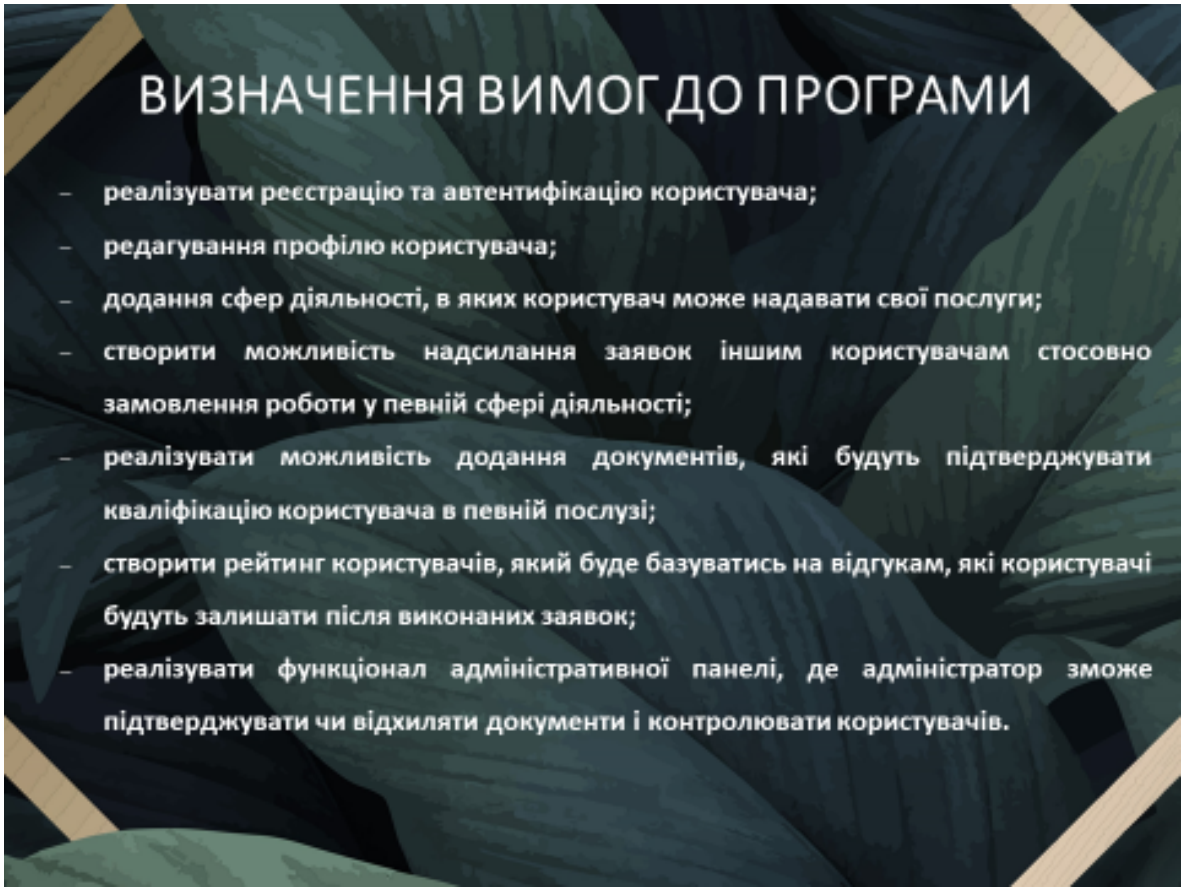
ДОДАТОК В
(обов’язковий)

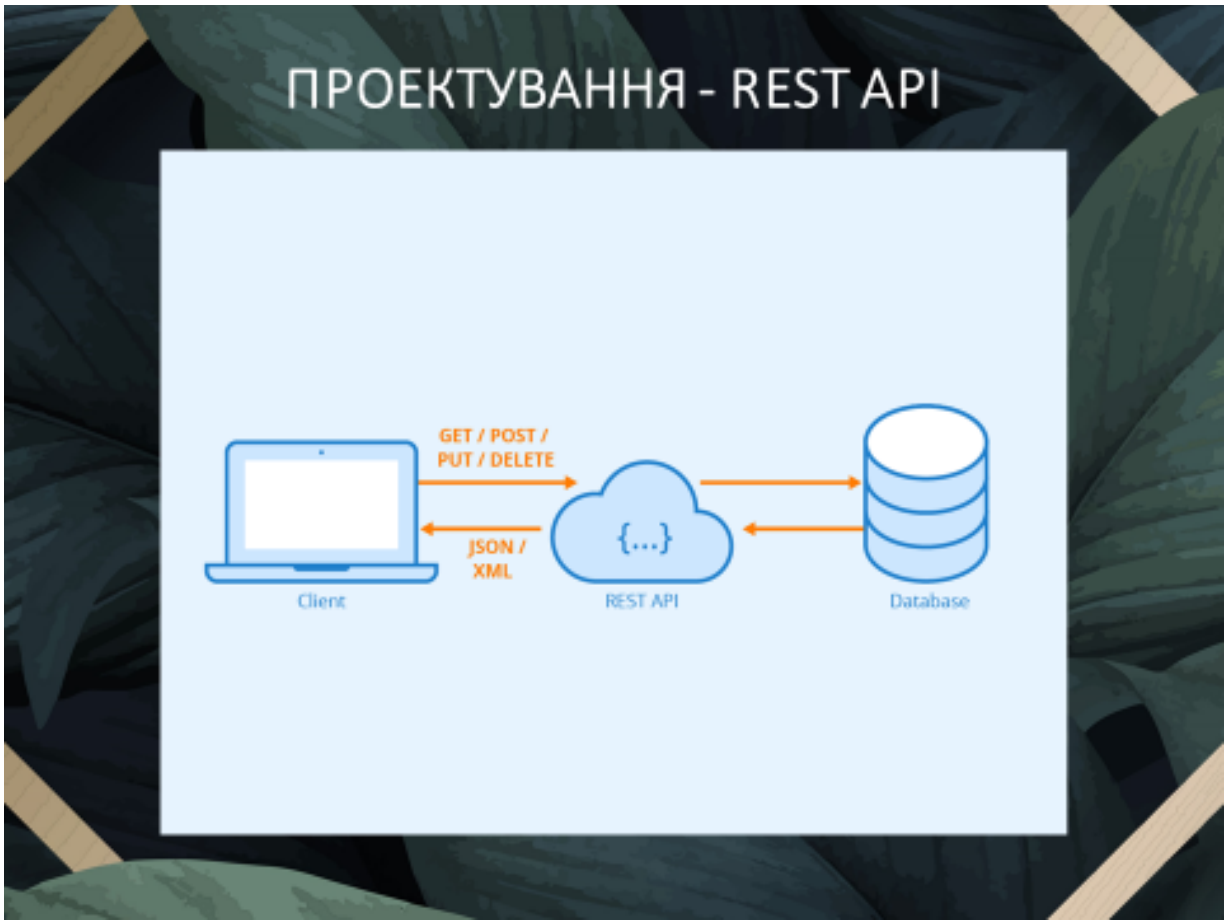
ПРЕЗЕНТАЦІЙНІ МАТЕРІАЛИ

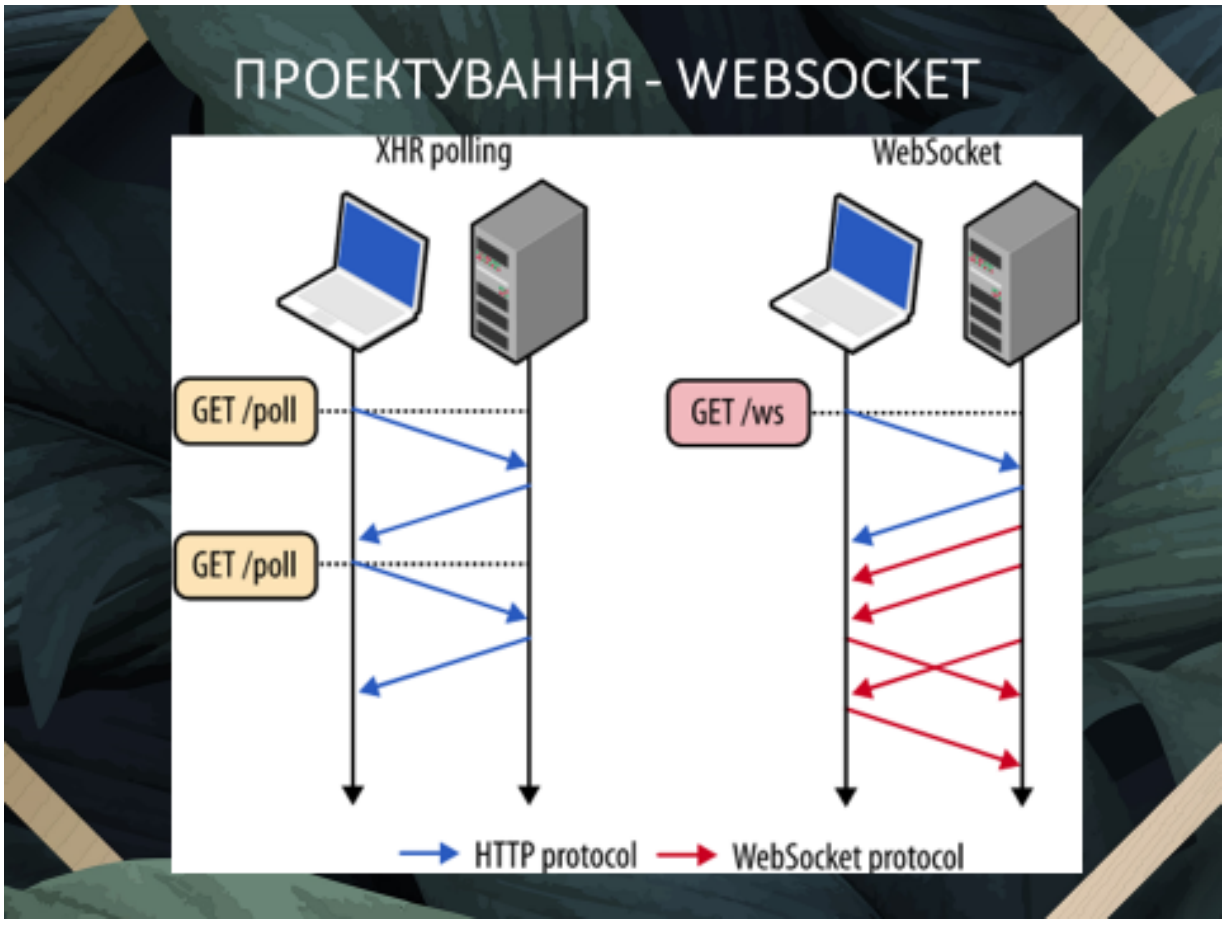

Хмельницький національний університет
Факультет інформаційних технологій
Кафедра інженерії програмного забезпечення
Кваліфікаційна робота на тему: «Серверна частина REST API для кросплатформного фріланс-сервісу LiveFreelance»
Студент: ІПЗс-20-1 Д. А. Ямборко
Керівник: канд. техн. наук, доцент Г. І. Радельчук


АКТУАЛЬНІСТЬ ТЕМИ РОБОТИ
Сьогодні існує досить мало сервісів та додатків які допомагають вирішити проблеми пошуку виконавця певної послуги та знаходження роботи, особливо в невеликих містах. Самостійно майже неможливо знайти виконавця чи замовника, без попереднього досвіду, чи досвіду знайомих. Буває вкрай складно вибрати з усього того, що пропонують існуючі методики.

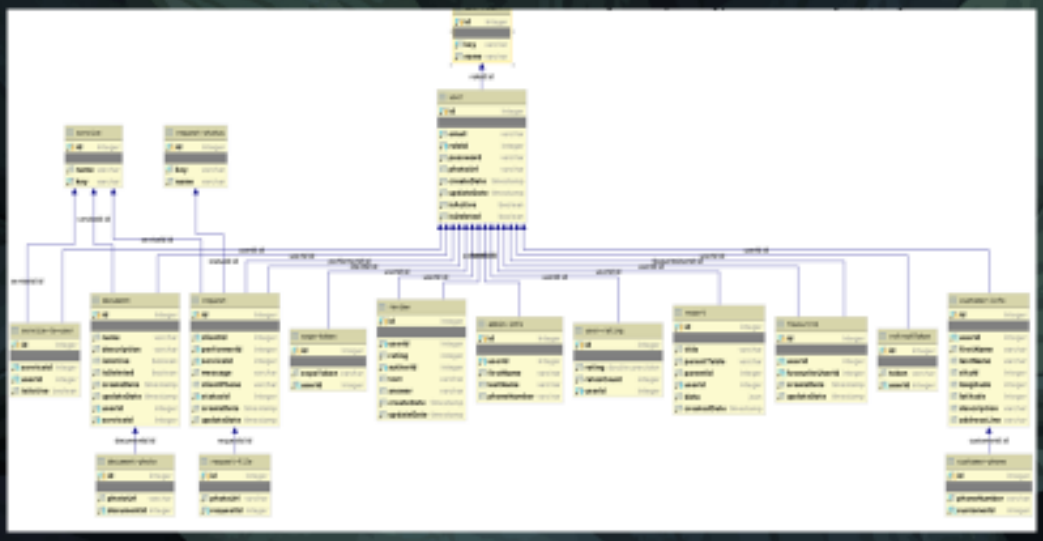

МЕТА ТА ЗАВДАННЯ РОБОТИ
Мета роботи – розробка серверної частини для фріланс-сервісу, який дозволить особам різних категорій зайнятості знаходити роботу без зайвих складнощів, а роботодавцям знаходити висококваліфікованих фахівців, які можуть виконати необхідну роботу.
Завдання, які необхідно вирішити для досягнення мети:
– проаналізувати предметну область для визначення її головних особливостей;
– проаналізувати аналогічне програмне забезпечення;
– визначити вимоги до програми та розробити технічне завдання;
– спроектувати програмний продукт;
– виконати програмну реалізацію;
– протестувати додаток.


ВИЗНАЧЕННЯ ВИМОГ ДО ПРОГРАМИ
– реалізувати реєстрацію та автентифікацію користувача;
– редагування профілю користувача;
– додання сфер діяльності, в яких користувач може надавати свої послуги;
– створити можливість надсилання заявок іншим користувачам стосовно замовлення роботи у певній сфері діяльності;
– реалізувати можливість додання документів, які будуть підтверджувати кваліфікацію користувача в певній послузі;
– створити рейтинг користувачів, який буде базуватись на відгукам, які користувачі будуть залишати після виконаних заявок;
– реалізувати функціонал адміністративної панелі, де адміністратор зможе підтверджувати чи відхиляти документи і контролювати користувачів.

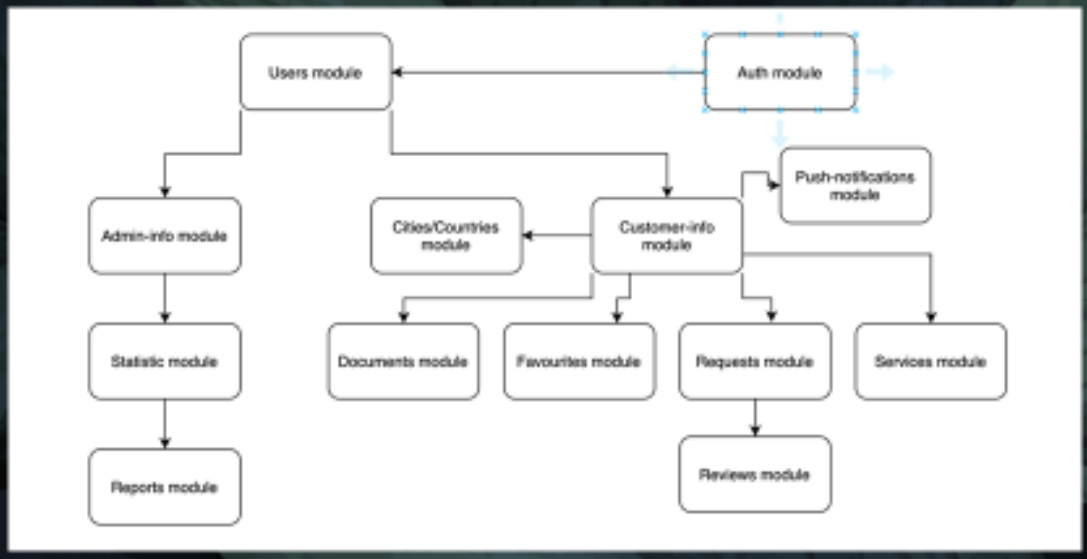

ПРОЕКТУВАННЯ - REST API
GET / POST /
PUT / DELETE
JSON /
XML
{...}
Client
REST API
Database


ПРОЕКТУВАННЯ - WEBSOCKET
XHR polling
WebSocket
GET /poll
GET /poll
GET /ws
HTTP protocol
WebSocket protocol

ПРОЕКТУВАННЯ – МОДЕЛЬ БАЗИ ДАНИХ



ПРОЕКТУВАННЯ – СТРУКТУРА ПРОГРАМИ



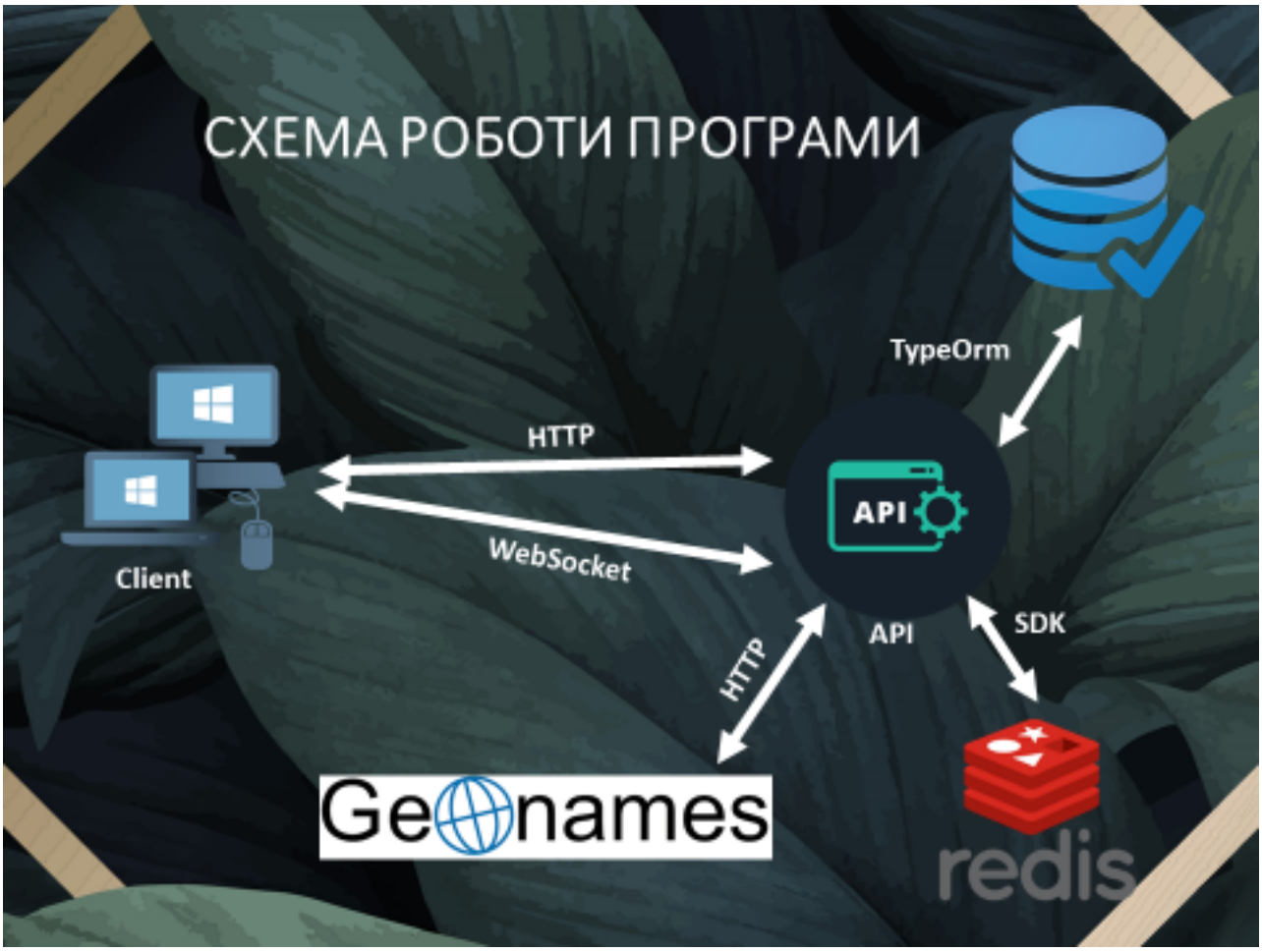


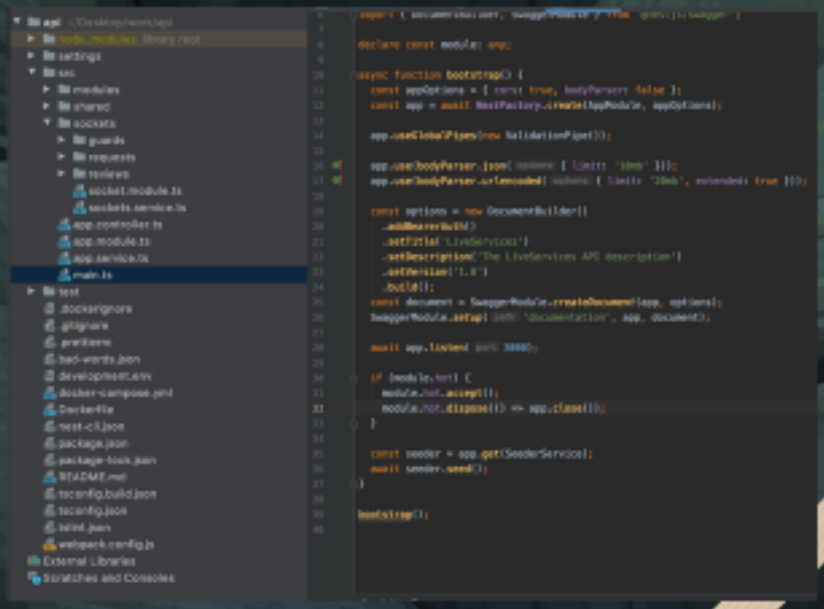
СХЕМА РОБОТИ ПРОГРАМИ
TypeOrm
HTTP
API
WebSocket
Client
API
SDK
HTTP
Geonames
redis


ВИКОРИСТАНІ ТЕХНОЛОГІЇ
JWT
JSON Web Tokens
mailgun
GeoNames
Expo
ESLint
Swagger
Supported by SMARTBEAR
nest
docker
MINIO
node
JS
redis
PostgreSQL
npm
TYPEORM
TS

ПРОГРАМНА РЕАЛІЗАЦІЯ

Приклад використання Node, TypeScript, NestJS, npm

```
  },
  "devDependencies": {
    "@nestjs/cli": "^6.9.0",
    "@nestjs/schematics": "^6.7.0",
    "@nestjs/testing": "^6.7.1",
    "@types/express": "^4.17.1",
    "@types/jest": "^24.0.18",
    "@types/node": "^12.7.5",
    "@types/passport-jwt": "^3.0.3",
    "@types/socket.io": "^2.1.4",
    "@types/supertest": "^2.0.8",
    "jest": "^24.9.0",
    "prettier": "^1.18.2",
    "start-server-webpack-plugin": "^2.2.5",
    "supertest": "^4.0.2",
    "ts-jest": "^24.1.0",
    "ts-loader": "^6.1.1",
    "ts-node": "^8.4.1",
    "tsconfig-paths": "^3.9.0",
    "tslint": "^5.20.0",
    "typescript": "^3.6.3",
    "webpack-node-externals": "^1.7.2"
  },
  "jest": {
    "moduleFileExtensions": [
      "js",
      "json",
      "ts"
    ],
    "rootDir": "src",
    "testRegex": ".spec.ts$",
    "transform": {
      "^.+\\.(t|j)s$": "ts-jest"
    },
    "coverageDirectory": "../coverage",
    "testEnvironment": "node"
  }
```



ПРОГРАМНА РЕАЛІЗАЦІЯ

Робота з TypeORM

```
@Module({
  imports: [
    ConfigModule,
    TypeOrmModule.forRootAsync({
      imports: [ConfigModule],
      inject: [ConfigService],
      useFactory: async (config: ConfigService) => config.getPostgresConfig() as PostgresConnectionOptions,
    }),
  ],
})
export class ConnectModule {
}
```

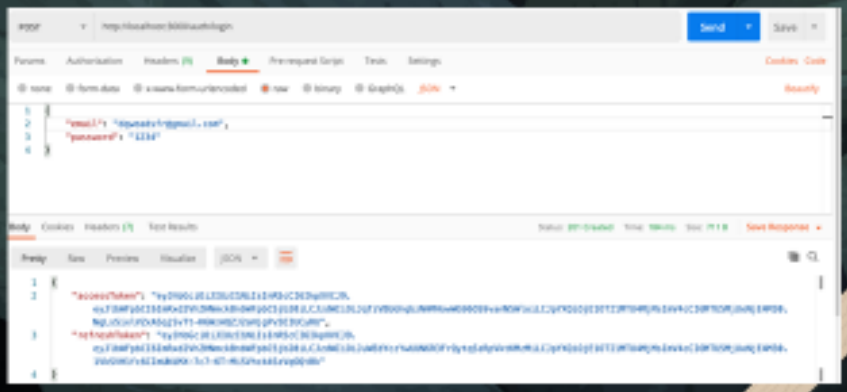
```
@Entity({ name: 'review' })
@Check(`"userId" <> "authorId"`)
export class Review extends BaseEntity {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  userId: number;

  @ManyToOne(type => User, user => user.reviews,
    { cascade: true, onDelete: 'CASCADE', onUpdate: 'CASCADE' })
  @JoinColumn({ name: 'userId' })
  user: User;

  @Column({ type: 'integer' })
  rating: number;

  @Column()
  authorId: number;

  @ManyToOne(type => User, user => user.writtenReviews,
    { cascade: true, onDelete: 'CASCADE', onUpdate: 'CASCADE' })
  @JoinColumn({ name: 'authorId' })
  author: User;
```

```
}
const profile = plainToClass(CustomerProfileResponseDto, await this.customerInfoR
  .select(this.profileSelections)
  .innerJoin('customer-info.user', 'user')
  .leftJoinAndSelect('customer-info.phoneNumbers', 'phoneNumbers')
  // .leftJoinAndSelect('customer-info.city', 'city')
  // .leftJoinAndSelect('city.country', 'country')
  .leftJoinAndSelect('user.serviceToUsers', 'serviceToUsers')
  .leftJoinAndSelect('serviceToUsers.service', 'service')
  .leftJoinAndSelect('user.documents', 'documents')
  .leftJoinAndSelect('documents.documentPhotos', 'documentPhotos')
  .leftJoinAndSelect('user.rating', 'rating')
  .where('userId = :userId', { userId })
  .andWhere('user.isDeleted = false')
  .andWhere('user.isActive = true')
  .getOne());
```

ПРОГРАМНА РЕАЛІЗАЦІЯ

Приклад авторизації і JWT

```
@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy) {
  constructor(
    private readonly config: ConfigService,
  ) {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      ignoreExpiration: false,
      secretOrKey: config.get('JWT_KEY'),
    });
  }

  async validate(payload: any) {
    return {
      id: Number(aes256.decrypt(this.config.get('AES256_KEY'), payload.sub)),
      email: payload.email,
    };
  }
}
```



РОЗГОРТАННЯ

Використання Docker, docker-compose

```
version: "3"

services:
  liveservices-api:
    build:
      context: ./
      dockerfile: Dockerfile
      args:
        - NODE_ENV=development
    environment:
      - PORT=3000
      - DB_HOST=postgres
      - DB_PORT=5432
      - DB_USER=postgres
      - DB_PW=postgres
    expose:
      - 3001
    ports:
      - 3200:3000
    depends_on:
      - liveservices-postgres
      - liveservices-minio
      - liveservices-redis
    links:
      - liveservices-redis
      - liveservices-postgres
      - liveservices-minio
    volumes:
      - ./:/home/node/app
    command: npm run start

  liveservices-redis:
    image: 'redis:4-alpine'
    command: redis-server --requirepass qwert
    ports:
      - '3203:6379'

  liveservices-postgres:
```

```
ARG DEBIAN_FRONTEND=noninteractive
FROM node:latest

RUN apt-get update
RUN apt-get install -y build-essential
RUN apt-get install -y python
RUN npm install -g @nestjs/cli

# ENV NPM_CONFIG_PREFIX=/home/node/.npm-global


RUN mkdir /home/node/app
RUN mkdir /home/node/app/node_modules

COPY ./package.json /home/node/app/package.json

RUN chown -R node /home/node/app/node_modules
USER node

RUN cd /home/node/app && ls

WORKDIR /home/node/app
COPY . /home/node/app

RUN cd /home/node/app && npm i --unsafe-perm=true
```

РЕАЛІЗАЦІЯ ДОКУМЕНТАЦІЇ API

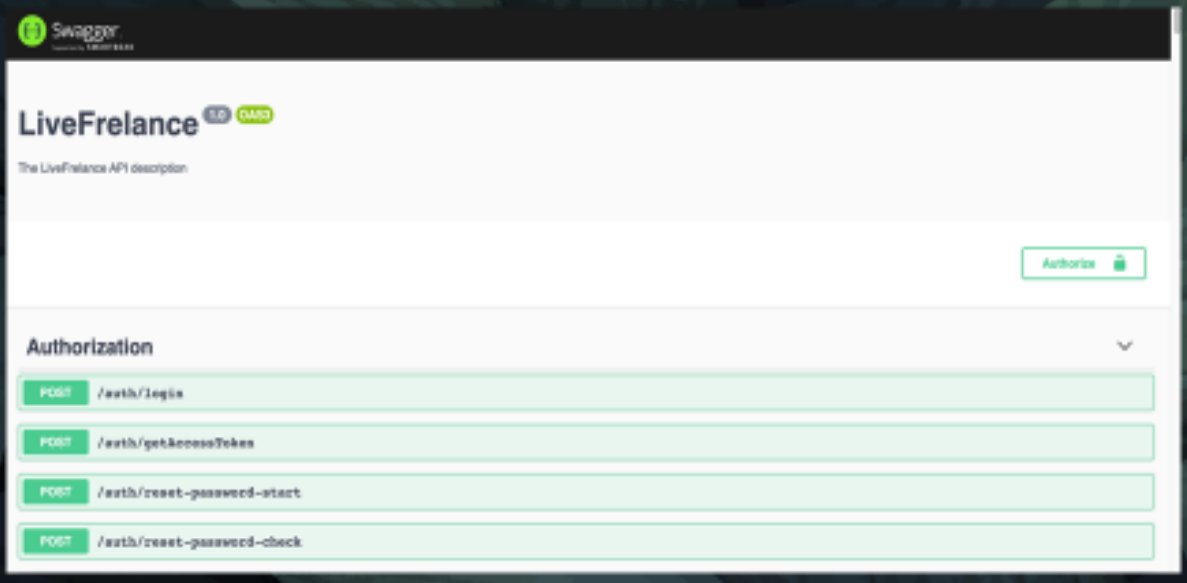
Приклад використання Swagger

```
@ApiBearerAuth()
@ApiForbiddenResponse( options: { description: 'Forbidden for this user' })
@ApiTags( tags: 'Reports')
@UseGuards(AuthGuard( type: 'jwt'), RoleGuard)
@Controller( prefix: 'reports')
export class ReportControllers {
  constructor(
    private readonly reportsService: ReportsService,
  ) {
  }

  @ApiPagination()
  @ApiResponse( options: { status: 404, description: 'User not found' })
  @ApiResponse( options: { status: 200, description: 'Returned reports list by user id', type: ReportsListResponseDto })
  @Roles( roles: 'admin')
  @Get( path: 'user/:id')
  async getAllReportsByUserId(@Param( property: 'id') userId: number, @Pagination() options): Promise<ReportsListResponseDto> {
    return await this.reportsService.getReportsByUserId(userId, options);
  }
```

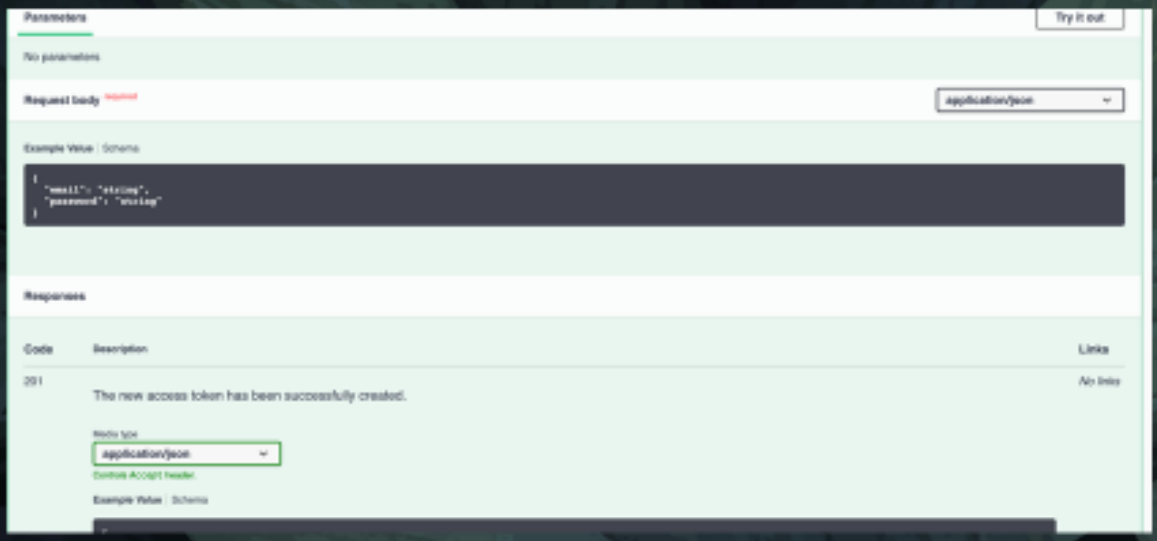
```
export class CityCreateDto {
  @ApiProperty()
  @IsNotEmpty()
  @IsString()
  name: string;
```

ГРАФІЧНИЙ ІНТЕРФЕЙС API

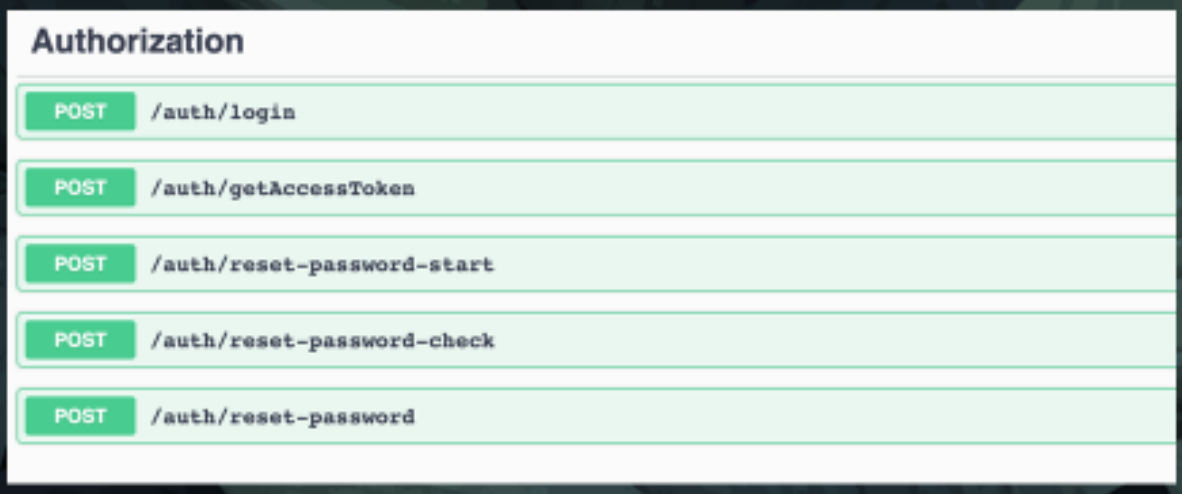
Головна сторінка документації

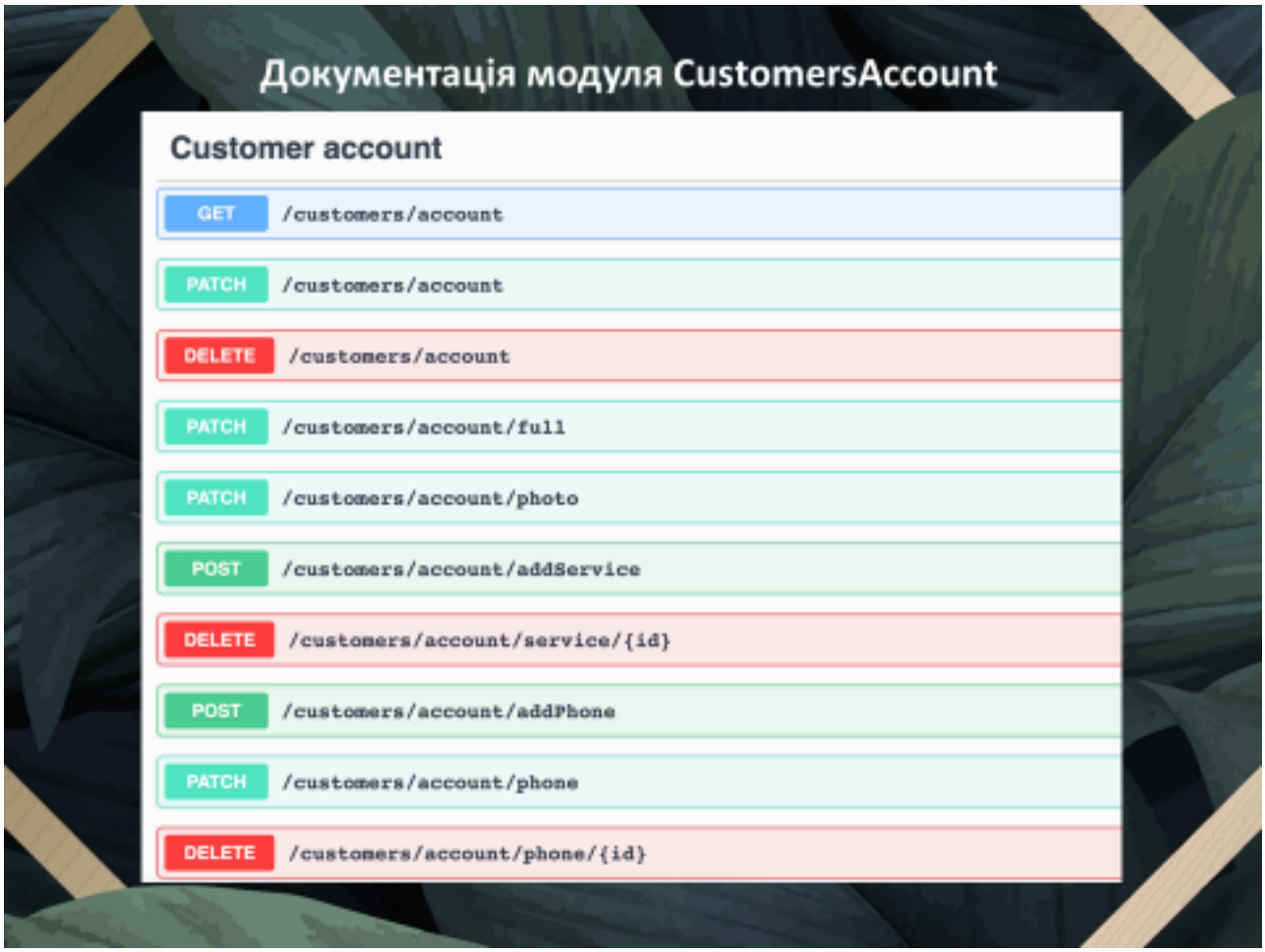


Розгорнута документація методу логін



Документація модулю Authorization

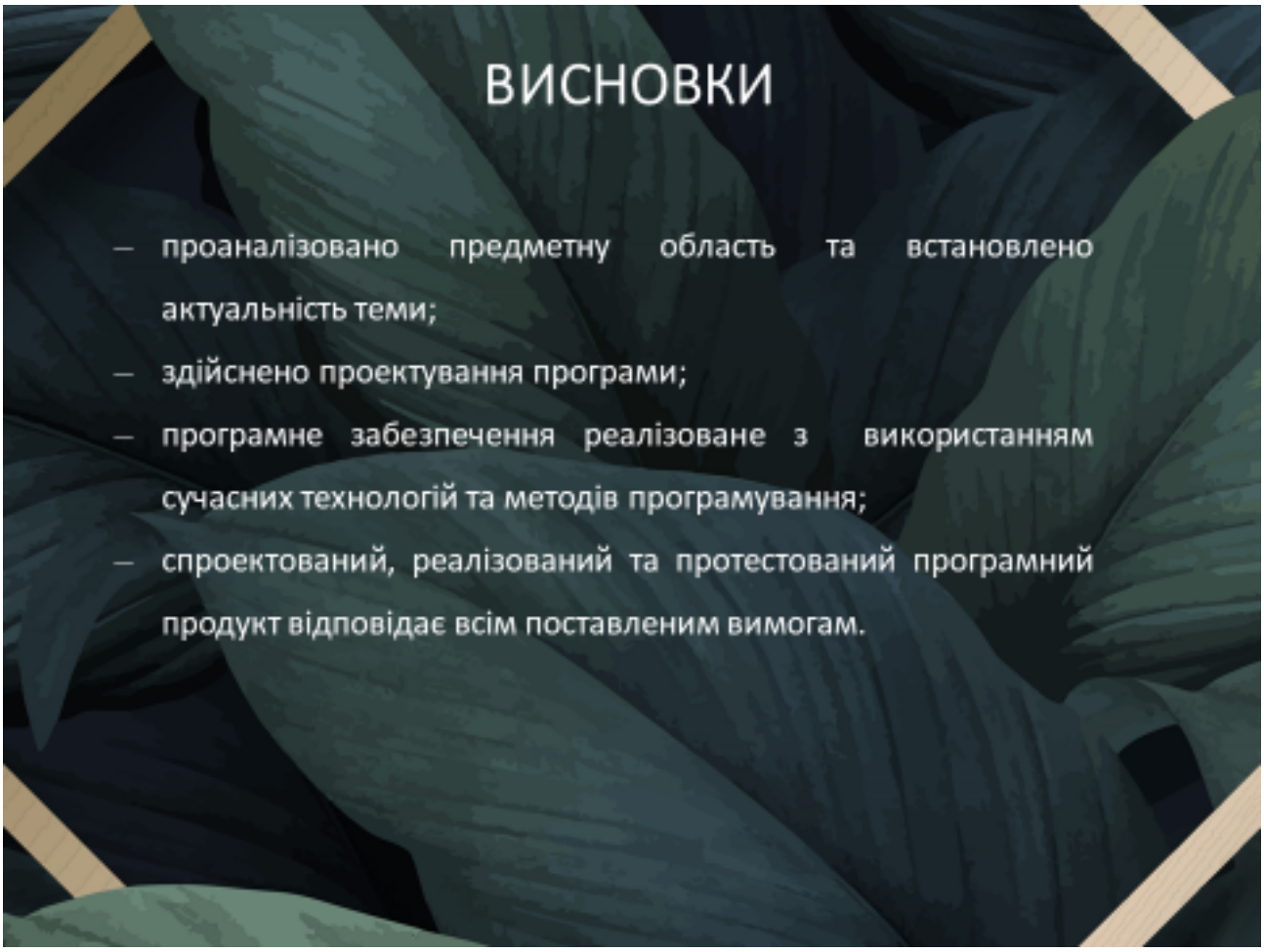




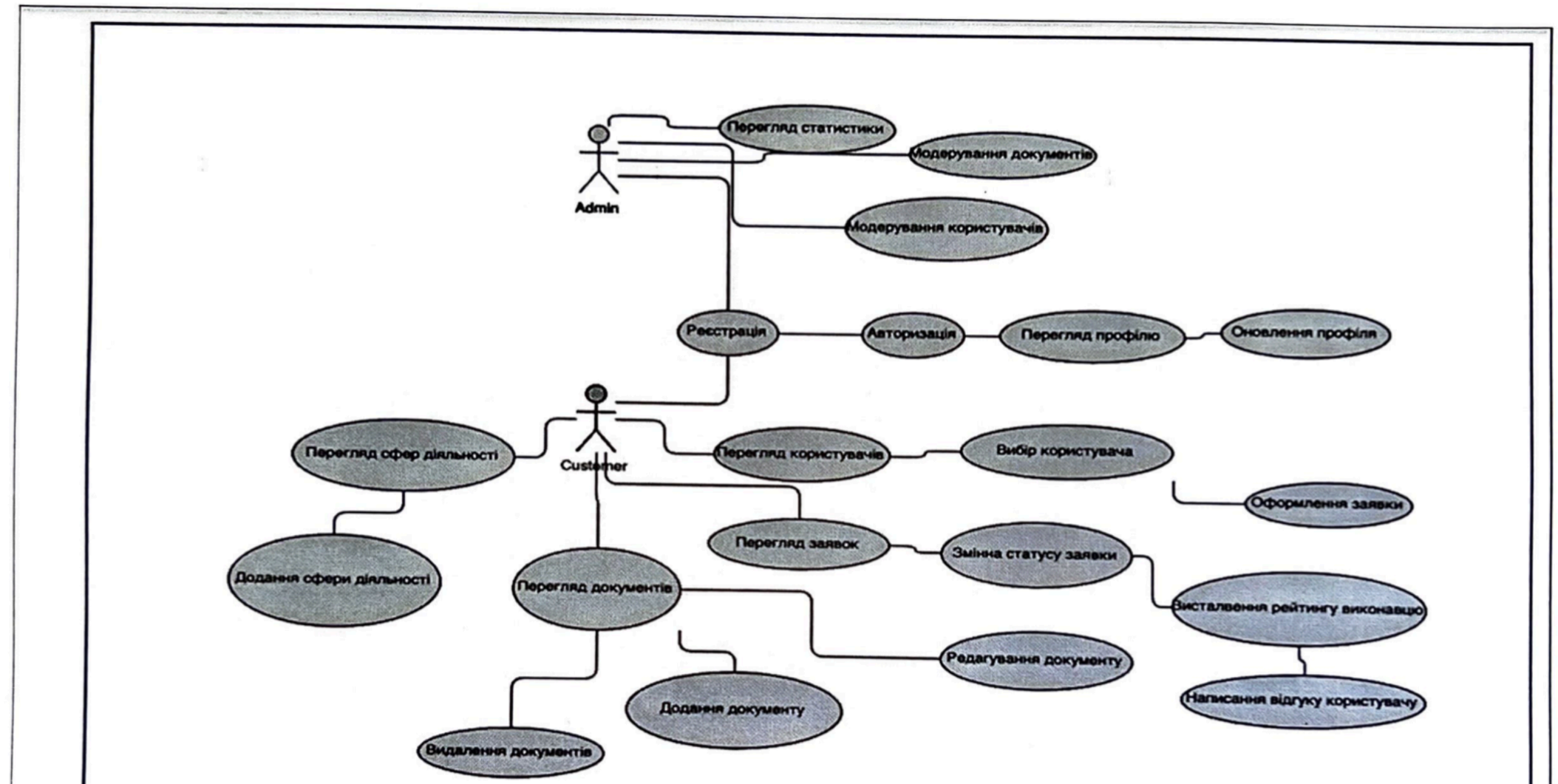
ТЕСТУВАННЯ ПРОГРАМИ

Виходячи з бізнес задач та можливостей в ході розробки програмного забезпечення, було проведено тільки мануальне тестування, тобто тестування вручну. Цей спосіб тестування є найбільш точним та дозволяє виявити найбільш проблемні та вразливі місця системи і також провалідувати програму на відповідність всім вимогам.

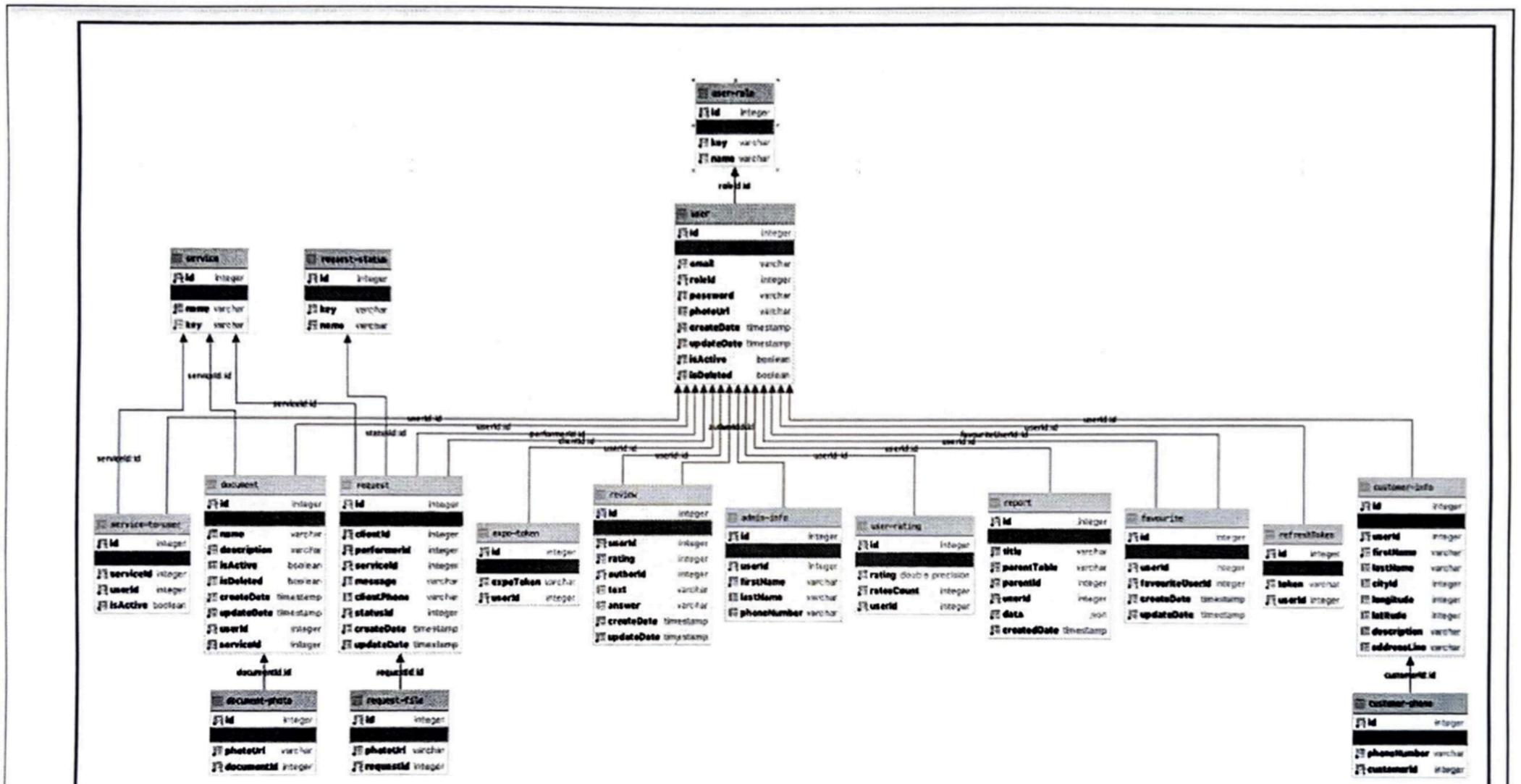
Назва сценарію	Модуль	Метод	Вхідні дані	Очікуваний результат	Отриманий результат
Реєстрація з коректними даними	User	Signup	Тіло запиту з атрибутами юзера	Створення юзера в базі та успішний результат від сервера	Пройдений
Реєстрація з некоректними даними	User	Signup	Тіло запиту без усіх атрибутами юзера	Помилка від сервера через неправильне тіло запиту	Пройдений
Логін з коректними даними	Auth	Login	Тіло запиту з коректним паролем і логіном	Генерація нових токену, рефреш токену і успішний результат від сервера	Пройдений
Логін з некоректними даними	Auth	Logic	Тіло запиту з неправильним паролем або логіном	Повернення помилки сервера через неправильний пароль або логін	Помилка. Помилку виправлено.
Отримання профілю користувача	User	GET user	Прикріплення токен до запиту	Дані профіля юзера в результаті запиту	Пройдений
Зміна паролю	User	resetPassword	Тіло запиту з старим паролем, новим паролем та емейлом	Оновлений пароль юзера та можливість залогінитися з новим паролем	Пройдений


ВИСНОВКИ
– проаналізовано предметну область та встановлено актуальність теми;
– здійснено проектування програми;
– програмне забезпечення реалізоване з використанням сучасних технологій та методів програмування;
– спроектований, реалізований та протестований програмний продукт відповідає всім поставленим вимогам.

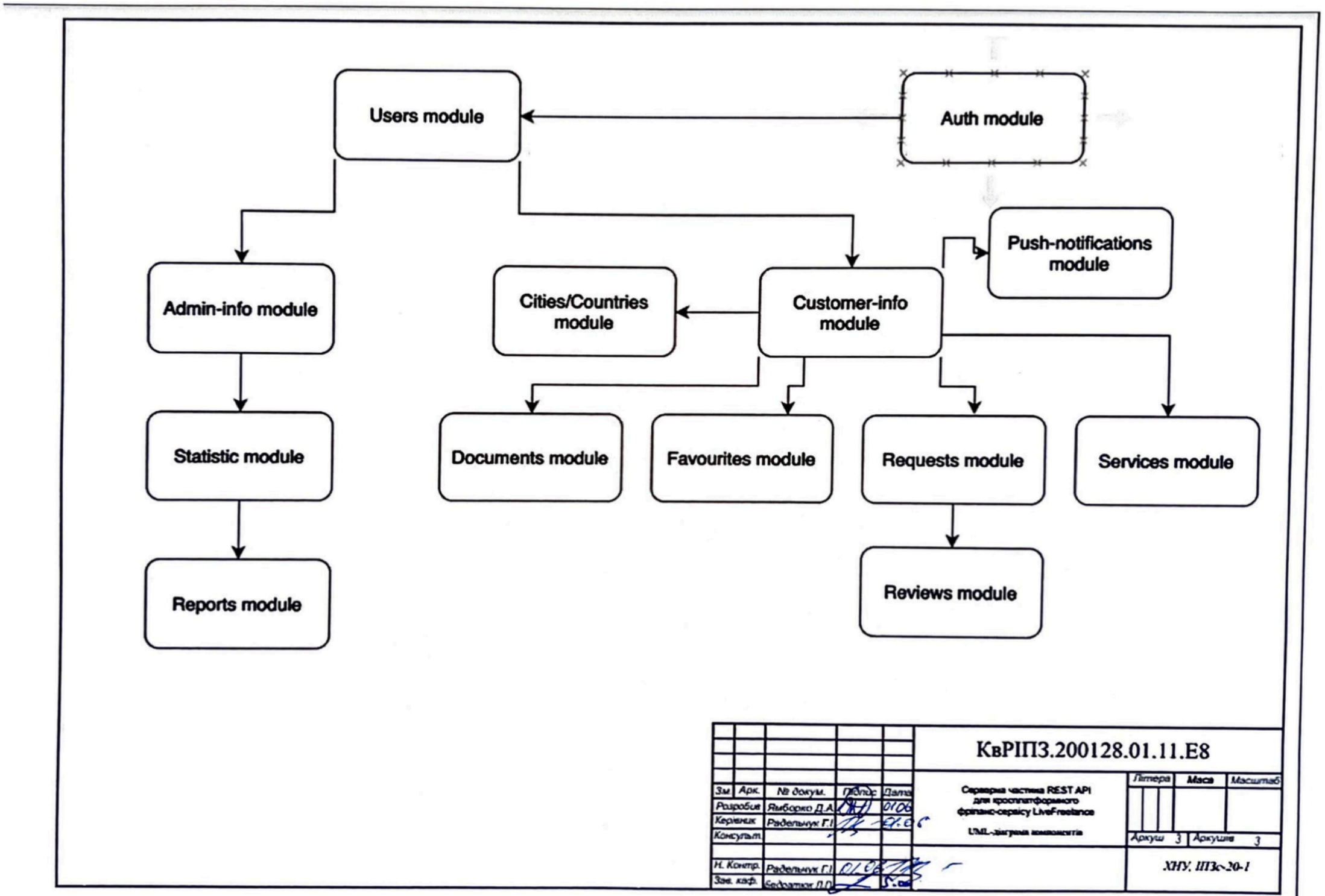
ГРАФІЧНА ЧАСТИНА



					КвРІПЗ.200128.01.11.Е8	
Зм.	Арк.	№ докум.	Підпис	Дата	Серверна частина REST API для кросплатформного фріланс-сервісу LiveFreelance UML-діаграма варіантів використання	Літера / Маса / Масштаб
Розробив		Явірко Д.А.		01.06		
Керівник		Радельчук Г.І.				Аркуш 1 / Аркушів 3
Консульт.						
Н. Контр.		Радельчук Г.І.				ХНУ, ІПЗс-20-1
Зав. каф.		Бердолаєв Д.П.				



					КвРІПЗ.200128.01.11.Е8			
Зм.	Арк.	№ докум.	Підпис	Дата	Серверна частина REST API для кросплатформного фріланс-сервісу LiveFreelance	Літера	Маса	Масштаб
Розробив		Яворівко Д.А.		01.06				
Керівник		Радельчук Г.І.		01.06		Аркуш 2	Аркушів 3	
Консульт.					Інфологічна модель бази даних			
Н. Контр.		Радельчук Г.І.		01.06		ХНУ, ІПЗс-20-1		
Зав. каф.		Бедратюк Л.П.		5.06				



					КвРІПЗ.200128.01.11.Е8		
Зм.	Арк.	№ докум.	Підпис	Дата	Серверна частина REST API для кроссплатформенного фріланс-сервісу LiveFreelance UML-діаграма компонентів	Літера	Маса / Масштаб
Розробив		Яворівко Д.А.		01.06			
Керівник		Радельчук Г.І.		01.06		Аркуш 3	Аркушів 3
Консульт.							
Н. Контр.		Радельчук Г.І.		01.06		ХНУ, ІПЗс-20-1	
Зав. каф.		Бефратюк Л.О.		01.06			

СУПРОВІДНІ ДОКУМЕНТИ

Завідувачу кафедри
інженеріїпрограмногозабезпечення проф.
Бедратюку Л. П.

здобувача вищої освіти
Ямборко Д. А.
Прізвище, ініціали

факультет ІТ, 3 курс, група ІПЗс-20-1

ЗАЯВА

З правилами чинного Положення «Про систему забезпечення академічної доброчесності в Хмельницькому національному університеті» від 01.07.2022, згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту і застосування заходів дисциплінарної та академічної відповідальності. ознайомлений. Про використання програмно-технічних засобів для перевірки кваліфікаційних робіт здобувачів вищої освіти на плагіат оповіщений та надаю свою згоду на обробку й 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/абоAnti-Plagiarism) і використання роботи для виявлення академічного 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 надається в друкованому та електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

02.05.2023
дата

підпис

Sun Jun 04 16:18:59 EEST 2023, Хіврич Володимир Русланович, Хмельницький національний університет, ХНУ

Anti-Plagiarism v-15.257

Максимальне співпадіння з одним документом 7.0%

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

	Документ		Сумарний збіг по Базі Даних	
ID: 114645 Назва: БКР Серверна частина REST API для кросплатформного фріланс-сервісу LiveFreelance Додано в БД: 2023-06-04 Автора: Ямборко Д.А. Керівники: Радельчук Г.І. к.т.н. доц. Консультанти: Опоненти:	Символи	Лексеми	Символи	Лексеми
	80475	779	10172 (13%)	116 (15%)

Джерело плагіату

ID	Опис	Наявність плагіату в документі	
		Символи	Лексеми

UNICHECK

Ім'я користувача:
Кафедра ІПЗ

Дата перевірки:
04.06.2023 17:51:05 EEST

Дата звіту:
04.06.2023 18:56:43 EEST

ID перевірки:
1015414217

Тип перевірки:
Doc vs Internet + Library

ID користувача:
100005589

Назва документа: ІПЗс-20-1_КвР_Ямборко_на_плагіат

Кількість сторінок: 72 Кількість слів: 12397 Кількість символів: 100046 Розмір файлу: 5.17 MB ID файлу: 1015076867

Виявлено модифікації тексту (можуть впливати на відсоток схожості)

8.07%
Схожість

Найбільша схожість: 2.53% з джерелом з Бібліотеки (ID файлу: 1011305742)

6.65% Джерела з Інтернету 346 ..... Сторінка 74

4.83% Джерела з Бібліотеки 88 ..... Сторінка 76

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 5

Підозріле форматування 18 сторінок

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА КВЛІФІКАЦІЙНУ РОБОТУ
освітнього ступеня «Бакалавр»

Дипломник Ямборко Дмитро Анатолійович

Тема Серверна частина REST API для кросплатформного фріланс-сервісу LiveFreelance

Спеціальність 121 – Інженерія програмного забезпечення

Обсяг кваліфікаційної роботи:

Кількість листів креслень 3 ; кількість сторінок записки 67

1.Короткий зміст пояснювальної записки та прийнятих рішень У процесі виконання кваліфікаційної роботи здійснено дослідження та аналіз предметної області, встановлені функціональні та нефункціональні вимоги. Проведений огляд наявних програм-аналогів на ринку, розглянуті їх переваги та недоліки, і підтверджена актуальність розробки нового програмного забезпечення. Були розглянуті різні інструменти та засоби для реалізації запропонованих рішень, що призвело до створення програмного забезпечення. Також було проведено тестування програми, під час якого було підтверджено, що розроблене програмне забезпечення працює правильно і готове до використання.

2. Висновок про відповідність роботи поставленому завданню Кваліфікаційна робота виконана згідно з поставленим завданням і у відповідності до всіх вимог.

3. Характеристика виконання кожного розділу роботи, ступінь використання останніх досягнень науки і техніки та передових методів роботи У вступі була обгрунтована актуальність теми, сформульовано мету та завдання дипломного проектування. У першому розділі проведений аналіз предметної області, розглянуті наявні рішення та визначені функціональні та нефункціональні вимоги до розроблюваного програмного забезпечення. У другому розділі проведено аналіз сучасних архітектур, розглянуті їх переваги та недоліки, і визначено, що система буде базуватись на монолітній архітектурі та моделі клієнт-сервер. У третьому розділі були підготовлені всі необхідні залежності для написання коду, виконана практична розробка програмних модулів і описані їх особливості, що призвело до створення програмного продукту. Також у розділі здійснено ручне тестування системи згідно з функціональними вимогами, що підтвердило правильну роботу програми.

4. Позитивні сторони роботи Тематика кваліфікаційної роботи є актуальною, оскільки на сьогодні в Україні фріланс-сервіси для пошуку роботи не є достатньо розвинутими та не мають достатньої кількості функціональних можливостей. Також було застосовано новітні технології для побудови програмного продукту та актуальні архітектурні рішення.

5. Негативні сторони роботи Час виконання певних запитів займає певний період, що може сповільнити роботу клієнта, який буде використовувати API. Також сервер мав би повертати більш зрозумілий текст помилок.

6. Оцінка графічного оформлення та пояснювальної записки Графічне оформлення виконано відповідно до тематики кваліфікаційної роботи, і представлено у вигляді діаграм та рисунків. Пояснювальна записка оформлена відповідно до вимог.

7. Відгук про кваліфікаційну роботу в цілому Кваліфікаційна робота є достойною та заслуговує позитивної оцінки. Матеріал пояснювальної записки структурований, послідовний, чіткий та легко зрозумілий, що дозволило чітко осягнути викладений матеріал в рамках теми проектування. Графічний матеріал вдалося використати для наочного відображення деталей проектування системи.

8. Інші зауваження

9. Оцінка кваліфікаційної роботи Кваліфікаційна робота виконана у повному обсязі, відповідає поставленій задачі та заслуговує на оцінку «задовільно».

РЕЦЕНЗЕНТ Мартинюк Валерій Володимирович, доктор технічних наук, професор, зав. кафедри автоматизації, комп'ютерно-інтегрованих технологій та робототехніки

"05" 06 2023 р. _____
(підпис)

РІШЕННЯ ЕКСПЕРНОЇ КОМІСІЇ
КАФЕДРИ ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ
ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Підтверджуємо ознайомлення з результатами звіту/звітів подібності щодо роботи. продукованими програмно-технічним засобом (ами) перевірки текстів на плагіат:

Назва: «Серверна частина REST API для кросплатформного фріланс-сервісу LiveFreelance»
Автор: Ямборко Дмитро Анатолійович
Спеціальність: 121 – Інженерія програмного забезпечення
Освітня програма: Освітньо-професійна програма «Інженерія програмного забезпечення»
Науковий керівник: Радельчук Галина Іванівна, кандидат технічних наук. доцент
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 й 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 кваліфікаційної роботи системами перевірки на плагіат виявлено схожість з деякими документами в частині загальновживаних обов`язкових словосполучень у стандартних бланках (титулка. відомість документів). у структурі змісту. назвах розділів/підрозділів тощо. у назвах публікацій у переліку джерел посилання;

2) в якості запозичень системою було зафіксовано деякі послідовності вихідного коду і посиланання на бібліотеки. які є стандартними мовними конструкціями програмування та не можуть розглядатися як об'єкт авторських прав і. відповідно. їх порушення;

3) усі запозичення є фрагментарними або мають належним чином оформленні посилання;

4) виявлені модифікації тексту не впливають на відсоток схожості.

Сумарний обсяг всіх запозичень. визначений системою виявлення збігів ідентичності/ схожості. складає 8.07% і адресується до 346 джерел з Інтернету та 88 джерел з бібліотеки. що. з урахуванням наведених обґрунтувань. відповідає характеру теми і свідчить на користь кваліфікаційної роботи.

Дата 05.06.2023

Завідувач кафедри  Леонід БЕДРАТЮК

Гарант освітньої програми  Леонід БЕДРАТЮК

Керівник кваліфікаційної роботи  Галина РАДЕЛЬЧУК