Вінницький національний технічний університет
Факультет інтелектуальних інформаційних технологій та автоматизації
Кафедра комп'ютерних наук

Бакалаврська дипломна робота на тему:

WEB-ЗАСТОСУНОК ДЛЯ ПРОХОДЖЕННЯ ОПИТУВАНЬ. ЧАСТИНА 1. СЕРВЕРНА ЧАСТИНА

Виконав: студент 4 курсу, групи 3КН-20б
спеціальності 122 – Комп'ютерні науки
(шифр і назва напряму підготовки, спеціальності)
Гарук В. В.
(прізвище та ініціали)

Керівник: к.т.н., доцент каф. КН
Колодний В. В.
(прізвище та ініціали)
«07» 06 2024 р.

Рецензент: к.т.н., доцент каф. АІІТ
Барабан М. В.
(прізвище та ініціали)
«07» 06 2024 р.

Допущено до захисту
Зав. кафедри Яровий А.А.
«07» 06 2024 р.

Вінниця ВНТУ - 2024 рік

Вінницький національний технічний університет
Факультет інтелектуальних інформаційних технологій та автоматизації
Кафедра комп'ютерних наук
Рівень вищої освіти перший (бакалаврський)
Галузь знань – 12 Інформаційні технології
Спеціальність – 122 Комп'ютерні науки
Освітньо-професійна програма – Комп'ютерні науки

ЗАТВЕРДЖУЮ
Завідувач кафедри КН
проф., д.т.н. Яровий А.А.
«07» 03 2024 р.

ЗАВДАННЯ
НА БАКАЛАВРСЬКУ ДИПЛОМНУ РОБОТУ СТУДЕНТУ
Гаруку Віталію Володимировичу

1. Тема роботи: WEB-застосункок для проходження опитувань. частина 1. серверна частина
керівник роботи: Колодний Володимир Володимирович, к.т.н., доц.
затверджені наказом вищого навчального закладу «11» 03 2024 року № 80

2. Термін подання студентом роботи 27 05 2024 р.

3. Вихідні дані до роботи:
Кількість варіантів аутентифікації – 2;
Кількість типів запитань – 3;
Кількість можливих питань – 7-10;
Кількість аналізованих емоцій – 1-9;
API для взаємодії з клієнтською частиною.
Документація до API

4. Зміст текстової частини (перелік питань, які потрібно розробити):
вступ; обґрунтування доцільності розробки WEB-застосунку для проходження опитувань; проектування серверної частини WEB-застосунку для проведення опитувань; програмна реалізація серверної частини WEB-застосунку для онлайн-опитувань; тестування; створення документації; висновки; список використаних джерел; додатки.

5. Перелік ілюстративного матеріалу (з точним зазначенням обов'язкових креслень):
Загальна структурна схема архітектури серверної частини; схема функціонування серверної частини; загальна UML-діаграма класів; фрагмент реляційної моделі бази даних;

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1–3	Колодний В. В., к.т.н., доц. каф. КН		

7. Дата видачі завдання 07 03 2024

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва та зміст етапу	Термін виконання		Примітка
		початок	закінчення	
1	Обґрунтування доцільності розробки WEB-застосунку для проходження опитувань.	06.05.24	09.05.24	
2	Розробка структури серверної частини WEB-застосунку для проведення опитувань	10.05.24	14.05.24	
3	Обґрунтування вибору бекенд технологій	15.05.24	19.05.24	
4	Розробка модулів та компонентів системи	20.05.24	23.05.24	
5	Тестування та створення API документації	24.05.24	27.05.24	
6	Висновки та аналіз отриманих результатів	28.05.24	01.06.24	
7	Оформлення матеріалів до захисту БДР	02.06.24	06.06.24	

Студент _____ **Гарук В. В.**
(підпис) (прізвище та ініціали)

Керівник роботи _____ **Колодний В.**
(підпис) (прізвище та ініці

АНОТАЦІЯ

Бакалаврська дипломна робота складається з 92 сторінок формату А4, містить 8 рисунків та 4 таблиці. Список використаних джерел налічує 20 найменування. Метою роботи є розробка серверної частини веб-ресурсу для проведення опитувань.

У роботі обґрунтовано необхідність створення веб-ресурсу для проведення опитувань, виходячи з аналізу існуючих технічних рішень та сучасних методів і технологій. Було вибрано найбільш підходящі бекенд-технології, зокрема Nest.js, PostgreSQL та Docker, завдяки їх перевагам та відповідності вимогам проекту.

Для реалізації веб-ресурсу було розроблено алгоритм роботи програмного модуля. Програмний продукт створено в інтегрованому середовищі Visual Studio Code з використанням мов програмування TypeScript та SQL, а також фреймворків Nest.js. Використання PostgreSQL та Docker дозволило забезпечити надійність та масштабованість бекенду.

Розроблено механізми автентифікації та авторизації користувачів за допомогою JWT токенів, а також реалізовано можливість логіну через електронну пошту та Google Account.

Тестування програмного продукту підтвердило успішну реалізацію основних функцій веб-ресурсу для проведення опитувань, що свідчить про досягнення поставлених цілей.

Ключові слова: веб-ресурс, опитування, бекенд, Nest.js, PostgreSQL, Docker.

ABSTRACT

The bachelor's thesis consists of 92 A4 pages, containing 8 figures and 4 tables. The list of references includes 20 items. The aim of the thesis is to develop the server-side part of a web resource for conducting surveys.

The necessity of creating a web resource for conducting surveys is justified in the work, based on the analysis of existing technical solutions and modern methods and technologies. The most suitable backend technologies, such as Nest.js, PostgreSQL, and Docker, were chosen due to their advantages and compliance with project requirements.

An algorithm for the operation of the software module was developed for the implementation of the web resource. The software product was created in the integrated development environment Visual Studio Code using the programming languages TypeScript and SQL, as well as the Nest.js framework. The use of PostgreSQL and Docker ensured the reliability and scalability of the backend.

Mechanisms for user authentication and authorization using JWT tokens were developed, and the ability to log in via email and Google Account was implemented.

Testing of the software product confirmed the successful implementation of the main functions of the web resource for conducting surveys, indicating that the set goals were achieved.

Keywords: web resource, surveys, backend, Nest.js, PostgreSQL, Docker.

ЗМІСТ

ВСТУП

Актуальність теми. В останні роки значно зросла популярність онлайн-інструментів для збору даних, таких як веб-ресурси для проведення опитувань. Вони стали невід'ємною частиною різних сфер діяльності, включаючи маркетинг, соціологію, освітні проекти та внутрішні дослідження компаній. Ці ресурси дозволяють ефективно збирати та аналізувати думки і зворотній зв'язок від широкої аудиторії, що є критично важливим для прийняття обґрунтованих рішень.

За даними звіту SurveyMonkey, загальна кількість користувачів платформ для проведення опитувань продовжує зростати, що вказує на важливість цих інструментів у сучасному світі. Такі ресурси, як Google Forms, SurveyMonkey та Typeform, стали необхідними інструментами для різних дослідницьких завдань, адже вони дозволяють легко створювати, поширювати та аналізувати опитування.

На сьогодні існує багато засобів для проведення опитувань, доступних як на веб-сторінках, так і у вигляді мобільних додатків. Доступність веб-ресурсу для проведення опитувань у браузері є важливою для бізнес-користувачів, які потребують швидкого та зручного способу збору даних від клієнтів та співробітників. Вони можуть легко використовувати веб-ресурс прямо зі своїх робочих місць, що дозволяє швидко отримувати інформацію та аналізувати результати незалежно від місцезнаходження.

Метою дослідження є впровадження функціональних можливостей для дослідження спектру емоцій користувачів за допомогою опитувань.

Об’єкт дослідження: процес створення та проведення онлайн опитувань.

Предмет дослідження: програмні засоби та алгоритми для розробки веб-ресурсу проведення онлайн опитувань.

Задачі дослідження:

1. Обґрунтувати доцільність розробки веб-ресурсу для проходження опитувань;

2. Обґрунтувати вибір бекенд технологій;

3. Розробити механізми автентифікації та авторизації;

4. Розробка модулів та компонентів системи;

5. Виконання тестування API та налагодження системи;

Апробація роботи: Результати досліджень були апробовані LIII науково-технічній конференції підрозділів Вінницького національного технічного університету (НТКП ВНТУ–2024) у м. Вінниці у 2024 р. [1].

Публікації: за основними результатами досліджень опубліковано тези доповіді на науково-технічній конференції [1].

1 ОБҐРУНТУВАННЯ ТА ПОСТАНОВКА ЗАДАЧІ WEB-ЗАСТОСУНКУ ДЛЯ ПРОХОДЖЕННЯ ОПИТУВАНЬ

1.1 Формулювання вимог та постановка задачі

Сучасний світ все більше орієнтується на цифрові технології, що зумовлює зростання попиту на онлайн-інструменти для збору даних, такі як веб-ресурси для проведення опитувань. Ці ресурси дозволяють організаціям швидко отримувати зворотний зв'язок від широкої аудиторії, що є ключовим елементом у прийнятті обґрунтованих рішень.

Онлайн-опитування мають ряд переваг порівняно з традиційними методами збору даних. Вони забезпечують швидкий доступ до великої кількості респондентів, знижують витрати на проведення досліджень та автоматизують процес обробки результатів. Завдяки онлайн-опитуванням можливо оперативно реагувати на зміну настроїв та потреб аудиторії, що є особливо важливим у динамічному бізнес-середовищі.

Існує багато популярних онлайн-ресурсів для проведення опитувань, таких як Google Forms, Survio, Surveynuts та інші [2-4]. Ці платформи вже стали невід'ємною частиною дослідницьких процесів, дозволяючи не лише створювати опитування, але й збирати та аналізувати дані. Проте, спеціалізованих веб-ресурсів для проведення опитувань та дослідження спектрів емоцій опитанних ще не було розроблено.

Крім того, існують численні переваги використання веб-ресурсів для проведення опитувань:

1. Глобальний доступ - можливість охопити респондентів з різних куточків світу.
2. Економія часу - швидке створення та розповсюдження опитувань.
3. Гнучкість - можливість адаптувати опитування під різні потреби.
4. Кросплатформність - доступність на різних пристроях.
5. Синхронізація даних - автоматичне збереження та обробка відповідей.

Втім, існують також деякі недоліки:

1. Відсутність особистого контакту - складніше забезпечити емоційний зв'язок з респондентами.
2. Технічні проблеми - можливі збої в роботі інтернету або серверів.
3. Відсутність фізичного досвіду - не можна використовувати фізичні матеріали під час опитувань [5].

Розробка спеціалізованого веб-ресурсу для проведення опитувань, який забезпечить високу продуктивність, безпеку та зручність використання, є актуальним завданням. Це дозволить організаціям ефективно збирати та аналізувати дані, що сприятиме прийняттю обґрунтованих рішень та підвищенню якості управління.

Розробка веб-застосунку для проходження опитувань включає кілька ключових аспектів, зокрема створення серверної частини та розробку API для взаємодії з клієнтською частиною. Основна увага приділяється розробці серверної частини, яка забезпечує стабільну та надійну роботу веб-застосунку, а також створенню документації для API, що полегшить інтеграцію з клієнтською частиною.

Основні функції серверної частини веб-застосунку включають:

- Можливість авторизації та збереження анкет і відповідей.
- Можливість легко створювати нові опитування.
- Можливість аналізу зібраних даних у реальному часі.
- Оцінка емоційного стану учасників за допомогою спеціальних емоційних шкал.

Для реалізації веб-застосунку потрібно виконати такі завдання:

1. Вибір бекенд технологій: Обрати відповідні технології для реалізації серверної частини
2. Створення бази даних: Розробити структуру бази даних та реалізувати її для збереження анкет, відповідей та емоційних даних.

3. Автентифікація користувача: Реалізувати систему автентифікації користувачів з підтримкою логіну через пошту та пароль, а також через Google аккаунт.
4. Створення API: Розробити API для взаємодії з веб-застосунком, включаючи ендпоінти для створення, редагування, перегляду та видалення опитувань. Забезпечити належну документацію для API, щоб клієнтська частина могла легко інтегруватися з сервером.
5. Тестування API та різних модулів системи: Здійснити повне тестування розроблених модулів та API для забезпечення їх коректної роботи.

Розробка серверної частини передбачає використання сучасних технологій для створення надійного та масштабованого веб-застосунку.

Створений API надасть можливість інтеграції клієнтської частини з сервером, забезпечуючи стабільну взаємодію між користувачами та системою. Документація API полегшить процес розробки клієнтських застосунків, надаючи розробникам чіткі інструкції та приклади використання.

Отже, розробка серверної частини веб-застосунку для проходження опитувань є важливим кроком у створенні сучасного та ефективного інструменту для збору та аналізу даних, що сприятиме покращенню процесу прийняття рішень та управління інформацією.

Розробка спеціалізованого веб-ресурсу для проведення опитувань, який забезпечує високу продуктивність, безпеку та зручність використання, є актуальним завданням. Всі поставлені завдання були успішно виконані, що дозволило створити ефективний інструмент для збору та аналізу даних. Такий ресурс сприятиме прийняттю обґрунтованих рішень та підвищенню якості управління в різних сферах діяльності.

1.2 Висновок

У першому розділі було обґрунтовано необхідність розробки веб-застосунку для проведення опитувань, визначено основні вимоги до його

функціональності та технічні завдання. Розробка серверної частини та API, що забезпечують стабільну роботу та зручну інтеграцію з клієнтською частиною, є важливим кроком у створенні сучасного інструменту для збору та аналізу даних.

Основні завдання, включені в процес розробки, охоплюють вибір відповідних технологій для бекенду, створення бази даних, реалізацію системи автентифікації користувачів, розробку API для взаємодії з веб-застосунком, а також тестування розроблених модулів і API. Виконання цих завдань дозволить створити надійний і масштабований веб-застосунок, який задовольняє потреби сучасних користувачів.

Таким чином, розробка серверної частини веб-застосунку для проведення опитувань є важливим кроком у створенні ефективного інструменту для збору та аналізу даних, що сприятиме прийняттю обґрунтованих рішень та підвищенню якості управління інформацією.

2 ПРОЕКТУВАННЯ СЕРВЕРНОЇ ЧАСТИНИ ДЛЯ ПРОХОДЖЕННЯ ОПИТУВАНЬ

2.1 Опис інструментів розробки

Програмний комплекс збудовано на основі трирівневої архітектури, яка поєднує різні технології для кожного рівня. Це рішення було розроблено з метою забезпечення надійності та масштабованості, використовуючи найсучасніші технології.

На серверному рівні використовуються такі технології:

- TypeScript: мова програмування, яка надбудовується над JavaScript, додаючи типізацію та інші можливості, що сприяє написанню більш надійного та легко підтримуваного коду [6].
- Nest.js: фреймворк для побудови серверних додатків, який забезпечує модульну архітектуру та підтримку TypeScript [7].
- Sequelize: бібліотека для взаємодії з базою даних, яка спрощує процес доступу до даних та їх обробки [9].
- Node.js: середовище виконання JavaScript, яке дозволяє запускати серверний код на стороні користувача [9].

База даних побудована на основі PostgreSQL, потужної об'єктно-реляційної системи управління базами даних, яка забезпечує високу продуктивність, надійність та гнучкість у налаштуванні.

Таким чином, програмний комплекс поєднує в собі найсучасніші технології та архітектурні рішення, що гарантує його надійність, масштабованість та зручність у використанні.

2.1.1 Мова програмування JavaScript, фреймворки Nest.js.

JavaScript є динамічною, високорівневою мовою програмування, яка широко використовується для створення інтерактивних веб-сторінок. Завдяки

своїй універсальності JavaScript може виконуватися як на стороні клієнта, так і на стороні сервера. Це робить його ідеальним вибором для розробки веб-додатків, оскільки одна мова може використовуватися по всій технологічній стеку. JavaScript є однією з найпопулярніших мов програмування у світі, що забезпечує велику кількість ресурсів, бібліотек і фреймворків для розробників. Відносно простий синтаксис і низький поріг входу роблять його доступним для новачків, а також забезпечують високу продуктивність для досвідчених програмістів. Мова дозволяє динамічно маніпулювати об'єктами і створювати інтерактивні елементи на веб-сторінках, що значно покращує користувацький досвід. Крім того, JavaScript постійно оновлюється та вдосконалюється, підтримуючи сучасні стандарти ECMAScript, що забезпечує його актуальність і конкурентоспроможність.

Nest.js - це прогресивний фреймворк для побудови ефективних, масштабованих та надійних серверних додатків на базі Node.js. Використання TypeScript за замовчуванням дозволяє Nest.js забезпечувати статичну типізацію, що допомагає виявляти помилки на етапі компіляції і робить код більш зрозумілим та підтримуваним. Модульна архітектура Nest.js дозволяє розділяти додаток на окремі функціональні блоки, що полегшує розробку, тестування та підтримку коду. Фреймворк забезпечує високий рівень абстракції, використовуючи декоратори для визначення контролерів, сервісів та інших компонентів, що спрощує розробку.

Nest.js підтримує інтеграцію з різними базами даних завдяки ORM (Object-Relational Mapping), такими як TypeORM та Sequelize, що дозволяє легко працювати з базами даних, як-от PostgreSQL, MySQL та MongoDB. Вбудована підтримка для побудови REST API та GraphQL забезпечує зручні інструменти для створення API, що робить його ідеальним для розробки сучасних веб-додатків. Завдяки використанню Node.js, Nest.js забезпечує високу продуктивність та низьку затримку при обробці запитів, що є критично важливим для масштабованих серверних додатків.

Вибір Nest.js для розробки серверної частини веб-ресурсу обумовлений його перевагами. Модульність Nest.js дозволяє розділяти додаток на окремі модулі, кожен з яких відповідає за певну функціональність, що значно спрощує розробку, тестування та підтримку коду. Використання TypeScript забезпечує кращу типізацію, зменшує кількість помилок під час розробки і робить код більш зрозумілим та підтримуваним. Nest.js забезпечує зручні інструменти для інтеграції з різними базами даних та сервісами, що дозволяє легко налаштовувати взаємодію з PostgreSQL та іншими компонентами системи.

Фреймворк також забезпечує високу продуктивність та гнучкість, дозволяючи масштабувати додаток відповідно до зростаючих вимог. Активна підтримка спільнотою і постійне оновлення Nest.js забезпечує відповідність сучасним стандартам розробки програмного забезпечення.

Таким чином, використання JavaScript у поєднанні з фреймворком Nest.js для реалізації серверної частини веб-ресурсу для проведення опитувань дозволяє забезпечити високу надійність, масштабованість та зручність розробки. Ці технології надають необхідні інструменти для створення ефективного та сучасного веб-додатка, що відповідає всім сучасним вимогам веб-розробки.

2.1.2 Користувацький токен автентифікації JWT.

JSON Web Token (JWT) є стандартом відкритого доступу (RFC 7519) для створення токенів доступу, які дозволяють безпечно передавати інформацію між сторонами у вигляді JSON-об'єкта. Цей стандарт широко використовується для автентифікації та авторизації користувачів у веб-додатках.

JWT складається з трьох частин: заголовок (header), корисне навантаження (payload) і підпис (signature). Заголовок містить інформацію про тип токена (JWT) та алгоритм шифрування, який використовується для підпису токена (наприклад, HMAC SHA256 або RSA). Корисне навантаження містить заяви (claims), які можуть бути зарезервованими, публічними або приватними. Зарезервовані заяви є стандартними і визначені в RFC 7519, публічні заяви

можуть бути визначені користувачами, а приватні заяви використовуються для передачі специфічної інформації між сторонами. Підпис створюється шляхом кодування заголовка та корисного навантаження, а потім застосування алгоритму шифрування з використанням секретного ключа або пари відкритий/закритий ключ. Підпис використовується для перевірки цілісності повідомлення та автентифікації джерела.

Процес автентифікації з використанням JWT складається з кількох етапів. Користувач вводить свої облікові дані (логін і пароль) та відправляє їх на сервер. Сервер перевіряє облікові дані користувача. Якщо вони правильні, сервер створює JWT, який містить інформацію про користувача та його права доступу, і відправляє цей токен клієнту. Клієнт зберігає JWT, зазвичай у локальному сховищі або куках. Кожен раз, коли клієнт робить запит до захищеного ресурсу на сервері, він відправляє JWT у заголовку запиту. Сервер отримує запит з токеном, перевіряє його підпис та цілісність, а також перевіряє, чи не закінчився термін дії токена. Якщо токен валідний, сервер обробляє запит і відправляє відповідь клієнту[10].

Використання JWT має кілька переваг. Завдяки криптографічному підпису, JWT забезпечує цілісність і автентичність переданої інформації. Оскільки JWT є самодостатнім токеном, що містить всю необхідну інформацію для автентифікації, це зменшує навантаження на сервери автентифікації. JWT може використовуватися як для автентифікації, так і для авторизації, що робить його універсальним інструментом для управління доступом. Крім того, JWT може використовуватися в різних сценаріях, таких як веб-додатки, мобільні додатки, API та мікросервіси.

У проекті для забезпечення безпеки та управління доступом до ресурсів використовується JWT. Коли користувач проходить автентифікацію, сервер генерує JWT, який містить інформацію про користувача і його права доступу. Токен відправляється клієнту, який зберігає його і використовує для доступу до захищених ресурсів. Сервер перевіряє валідність токена на кожен запит, забезпечуючи тим самим безпеку і цілісність даних.

Таким чином, використання JWT для автентифікації в цьому проекті забезпечує високий рівень безпеки, масштабованість і зручність у розробці та експлуатації веб-ресурсу для проведення опитувань.

2.1.3 Система керування базами даних PostgreSQL

PostgreSQL є потужною об'єктно-реляційною системою керування базами даних, яка підтримує більшість стандартів SQL та багато сучасних функцій. Вона відзначається високою продуктивністю, стабільністю та гнучкістю, що робить її ідеальним вибором для різних додатків, від малих до великих.

У мому проекті для взаємодії з PostgreSQL використовується фреймворк Sequelize, який є ORM (Object-Relational Mapping) для Node.js [11]. Це дозволяє зручно працювати з базою даних у вигляді об'єктів, зберігаючи при цьому всі переваги реляційної бази даних.

Приклад налаштування підключення до PostgreSQL у нашому проекті наведено у наступному коді:

```
@Module({
  imports: [
    SequelizeModule.forRootAsync({
      imports: [ConfigModule],
      inject: [ConfigService],
      useFactory: (configService: ConfigService) => ({
        dialect: 'postgres',
        host: configService.get<string>('pghost'),
        port: configService.get<number>('pgport'),
        username: configService.get<string>('pgusername'),
        password: configService.get<string>('pgpassword'),
        database: configService.get<string>('pgdatabase'),
        models: [Survey, User, Question, Response],
        dialectOptions: {
          pool: false,
```

```
      },
    }),
  }),
  SequelizeModule.forFeature([Survey, User, Question, Response]),
  LoggingModule,
 ],
 providers: [DatabaseService],
 exports: [DatabaseService, SequelizeModule],
})
export class DatabaseModule {}
```

Цей код налаштовує підключення до бази даних PostgreSQL, використовуючи конфігураційний модуль для отримання налаштувань підключення, таких як хост, порт, ім'я користувача, пароль і назва бази даних. SequelizeModule дозволяє легко інтегрувати PostgreSQL з нашим додатком, використовуючи моделі для визначення структури таблиць.

PostgreSQL забезпечує збереження даних у окремих таблицях, що підвищує швидкість і гнучкість доступу до даних. Таблиці пов'язані між собою за допомогою відносин, що дозволяє поєднувати дані з різних таблиць у запитах. У нашому проекті ми використовуємо кілька моделей для роботи з базою даних: User, Survey, Question і Response. Приклад синхронізації таблиць наведено у наступному коді:

```
@Injectable()
export class DatabaseService {
 constructor(
  private readonly logger: LoggingService,

  @InjectModel(User)
  private userModel: typeof User,

  @InjectModel(Survey)
```

```
    private SurveyModel: typeof Survey,

    @InjectModel(Question)
    private QuestionModel: typeof Question,

    @InjectModel(Response)
    private responseModel: typeof Response,
  ) {
    User.sync({ force: false })
      .then(() => console.log('Users table synced'))
      .catch((error) => console.error('Error creating users table:', error));
    Survey.sync({ force: false })
      .then(() => console.log(Survey table synced'))
      .catch((error) => console.error('Error creating Survey table:', error));
    Question.sync({ force: false })
      .then(() => console.log(Question table synced'))
      .catch((error) => console.error('Error creating Question table:', error));
    Response.sync({ force: false })
      .then(() => console.log(Response table synced'))
      .catch((error) => console.error('Error creating Response table:', error));
  }
}
```

Цей код синхронізує моделі з базою даних, створюючи таблиці, якщо вони ще не існують, і забезпечує обробку помилок у разі невдачі.

PostgreSQL підтримує стандарт ACID (Atomicity, Consistency, Isolation, Durability), що забезпечує надійність і цілісність даних навіть у разі збоїв системи або відключення живлення. Це робить PostgreSQL відмінним вибором для критично важливих додатків, де безпека і цілісність даних є пріоритетом. Система підтримує потужні механізми обробки транзакцій, включаючи

створення точок збереження (savepoints) і контроль ізоляції транзакцій, що забезпечує високий рівень контролю над виконанням операцій [12].

PostgreSQL також пропонує широкі можливості для розширення. Вона підтримує створення власних типів даних, функцій, операторів і розширень, що дозволяє налаштовувати базу даних відповідно до специфічних вимог проекту. Підтримка реплікації та масштабування дозволяє створювати високонавантажені системи, які можуть обробляти великі обсяги даних і забезпечувати високу доступність.

Таким чином, використання PostgreSQL у нашому проекті забезпечує надійність, масштабованість та зручність у розробці, що є критичними факторами для успішного виконання проекту.

2.2 Опис архітектури серверу

Архітектура сервера для веб-ресурсу з проведення опитувань реалізована на базі фреймворку Nest.js та використовує кілька ключових технологій і підходів для забезпечення надійності та масштабованості додатка.

Для взаємодії клієнтів з сервером використовується API на основі HTTP-запитів. Формат даних для обміну між клієнтом і сервером - JSON. API підтримує стандартні HTTP-методи: GET для отримання даних, POST для створення нових ресурсів, PUT для оновлення існуючих ресурсів, DELETE для видалення ресурсів.

Контролери обробляють HTTP-запити, що надходять від клієнтів, та викликають відповідні сервіси для виконання необхідних дій. Сервіси містять бізнес-логіку додатка, виконують основні операції, такі як обробка даних, взаємодія з базою даних та виконання бізнес-правил.

Для передачі даних між клієнтом і сервером використовуються DTO (Data Transfer Objects). Вони також забезпечують валідацію та трансформацію цих даних.

Для забезпечення безпеки та обробки запитів використовуються Middleware та Guards. Middleware виконують додаткові функції обробки запитів, такі як автентифікація, тоді як Guards контролюють доступ до певних ресурсів [13].

База даних використовується для зберігання інформації про користувачів, опитування, питання та відповіді. Взаємодія з базою даних здійснюється через ORM, що дозволяє працювати з даними у вигляді об'єктів.

Загальна струкртура схема архітектури серверної частини (рис. 2.1)

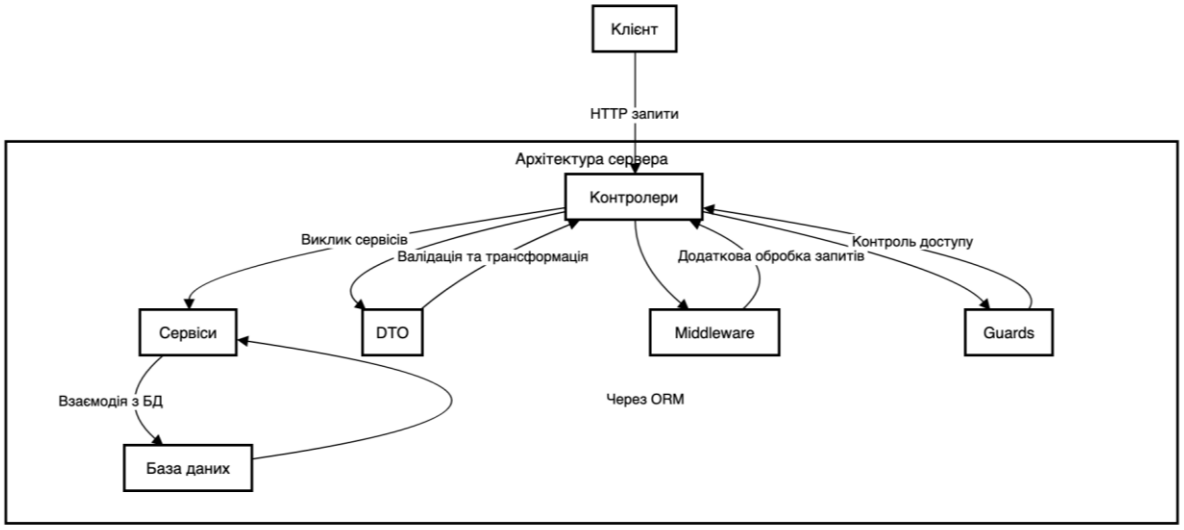


Рисунок 2.1 – Загальна структурна схема архітектури серверної частини

Архітектура сервера складається з кількох ключових компонентів: контролери, сервіси, DTO, middleware, guards та база даних. Контролери обробляють HTTP-запити та викликають сервіси для виконання дій. Сервіси виконують бізнес-логіку та взаємодіють з базою даних. DTO використовуються для передачі та валідації даних між клієнтом і сервером. Middleware та Guards забезпечують безпеку та обробку запитів. База даних зберігає дані та взаємодіє з сервером через ORM[13].

Таким чином, використання сучасних технологій та архітектурних підходів дозволяє створювати ефективні та масштабовані серверні додатки, які відповідають сучасним вимогам веб-розробки.

2.3 Розробка структури серверної частини WEB-застосунку для проведення опитувань

Backend частина веб-ресурсу для проведення опитувань базується на використанні двох основних технологій: React для фронтенду та Nest.js для бекенду. React забезпечує інтерактивний користувацький інтерфейс, а Nest.js відповідає за обробку запитів та взаємодію з базою даних

Модульність системи в Nest.js дозволяє розділяти додаток на окремі модулі, кожен з яких відповідає за певну функціональність. Це забезпечує зручність у розробці та підтримці додатка. У нашому випадку backend включає такі основні модулі (рис. 2.2) [14]:

AppModule - головний модуль додатка, який імпортує всі інші модулі та налаштовує їх взаємодію. Він є точкою входу в додаток і забезпечує ініціалізацію всіх інших компонентів.

AuthModule - модуль для автентифікації користувачів. Він включає в себе логіку для реєстрації, входу, виходу та захисту маршрутів за допомогою JWT (JSON Web Token). Цей модуль забезпечує безпечний доступ до захищених ресурсів і дозволяє керувати сесіями користувачів.

DatabaseModule - модуль для підключення до бази даних PostgreSQL. Він налаштовує підключення до бази даних, управляє міграціями та забезпечує доступ до даних через ORM (Object-Relational Mapping) або нативні SQL-запити.

SurveyModule - модуль для управління опитуваннями. Він містить логіку для створення, редагування, видалення та перегляду опитувань. Цей модуль також обробляє відповіді на опитування та забезпечує їх зберігання в базі даних.

ConfigModule - модуль для завантаження та керування конфігураціями додатка.

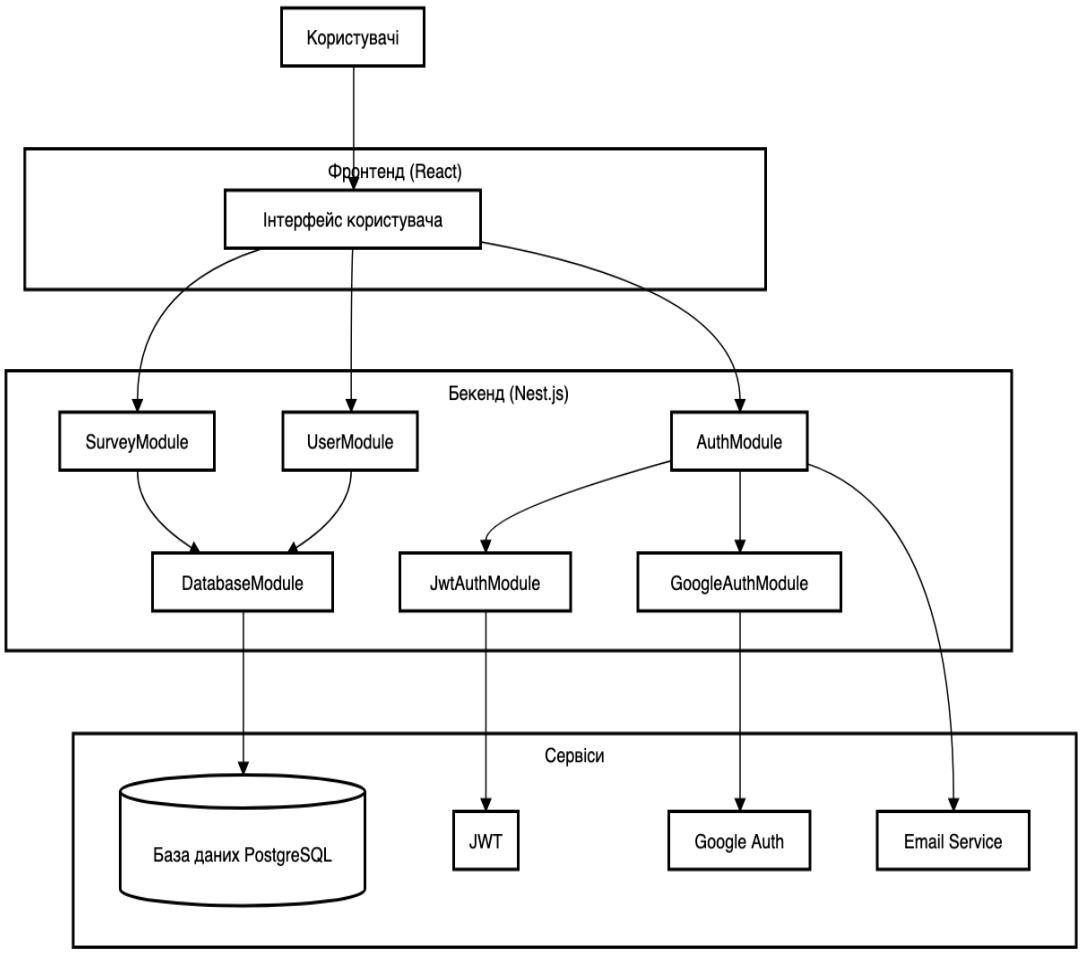


Рисунок 2.2 − Схема функціонування серверної частини

Ця схема надає загальне уявлення про взаємодію між фронтендом, бекендом та зовнішніми сервісами у додатку. Основні функціональні можливості бекенд включають авторизацію та автентифікацію користувачів, управління опитуваннями, обробку відповідей на опитування та відображення результатів.

Авторизація та автентифікація користувачів реалізується за допомогою JWT для захисту API. Це забезпечує надійну автентифікацію, оскільки кожен запит від користувача супроводжується токеном, який перевіряється на сервері. Підтримується логін через пошту та пароль, а також через Google аккаунт, що дозволяє користувачам обирати найбільш зручний спосіб входу до системи.

Управління опитуваннями включає такі функції, як створення нових опитувань, редагування та видалення існуючих опитувань, а також перегляд створених та взятих участь опитувань. Це дозволяє адміністраторам легко керувати контентом опитувань, а користувачам – зручно відповідати на них.

Обробка відповідей на опитування охоплює збереження відповідей учасників та обробку й аналіз емоційного стану учасників на основі відповідей. Збережені відповіді аналізуються для визначення емоційного стану, що допомагає отримувати більш детальну інформацію про учасників опитувань.
Відображення результатів включає надання результатів опитувань через API. Це дозволяє інтегрувати результати опитувань у різні додатки та сервіси, забезпечуючи їх доступність у реальному часі. Інтерактивні графіки та діаграми для аналізу даних забезпечують візуалізацію результатів, що полегшує їх інтерпретацію.

2.4 Розробка бази даних

Вся інформація системи, включаючи основні дані про відповіді, користувачів та опитування, зберігається в базі даних. Представлена UML-діаграма класів (рис. 2.3) надає візуалізацію структури бази даних, що допомагає у проектуванні та розумінні взаємозв'язків між різними сутностями системи. Використання UML-діаграм є важливим етапом, оскільки вони наочно демонструють, як різні компоненти системи взаємодіють між собою, що сприяє кращому розумінню її архітектури та спрощує процес розробки.

Розробка бази даних починається з аналізу вимог до зберігання та обробки даних. Цей процес включає оцінку потреб системи, визначення обсягу даних, які необхідно зберігати, частоти доступу до них, а також вимог до безпеки та резервного копіювання. На основі цього аналізу розробники визначають основні сутності, які будуть представлені в базі даних, а також їх взаємозв'язки.

Основними сутностями в даній системі є користувачі, опитування, питання та відповіді. Кожна з цих сутностей відображена у відповідній таблиці в базі даних, що забезпечує зручне зберігання та доступ до інформації. Користувачі взаємодіють з системою, створюючи та відповідаючи на опитування, кожне з яких може містити декілька питань. Відповіді на ці питання зберігаються окремо, що дозволяє легко аналізувати зібрані дані.

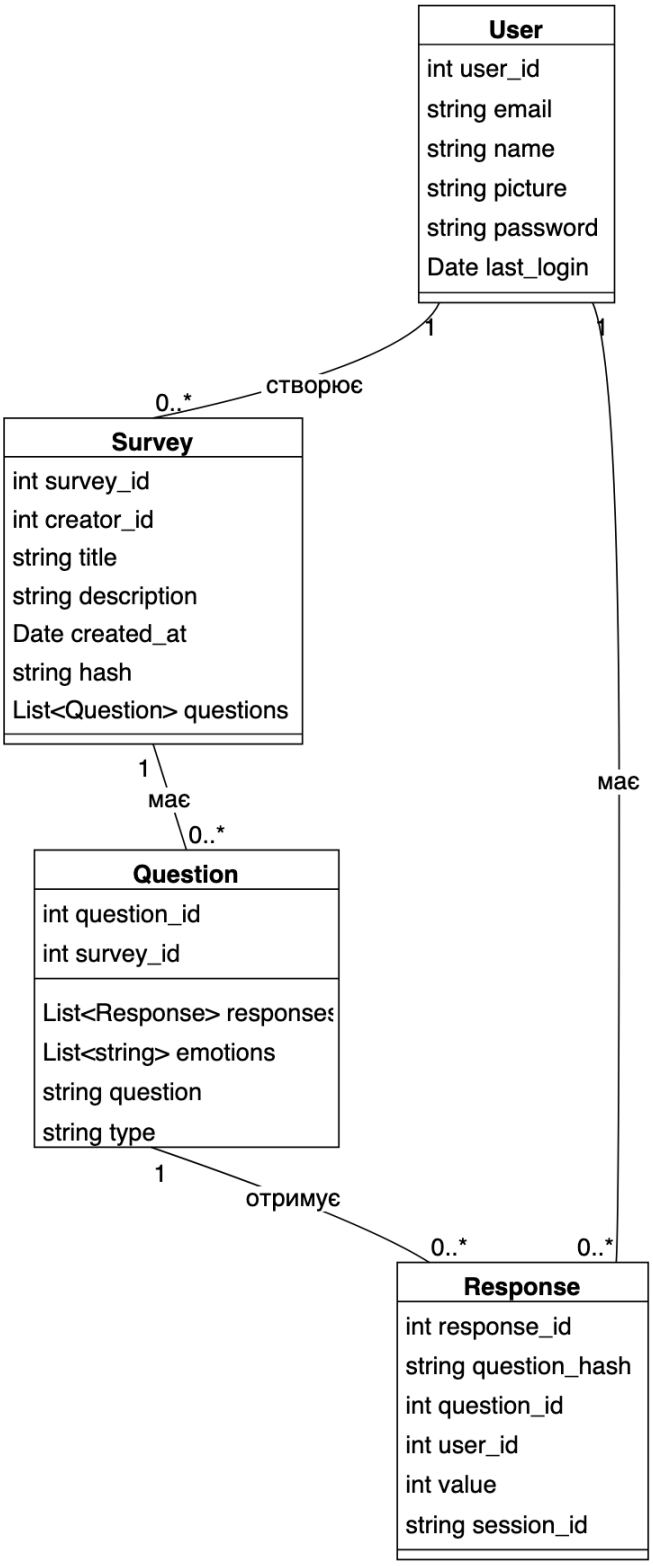


Рисунок 2.3 – Загальна UML-діаграма класів

Опис структури бази даних

База даних складається з наступних таблиць: “Користувач”, “Опитування”, “Питання”, “Відповіді”, “Користувач” (таблиця 2.1)

Таблиця 2.1 — Структура таблиці “Користувач”

Ім’я поля	**Тип і розмір поля**	**Опис поля**
user_id	int	Первинний ключ, автоінкремент
email	varchar	Унікальний email користувача
name	varchar	Ім’я користувача
picture	varchar	URL до зображення профілю
password	varchar	Хешований пароль користувача
last_login	date	Дата останнього входу

Таблиця складається з таких полів як ідентифікатор, email, ім’я, зображення профілю, пароль та дата останнього входу.

Таблиця “Опитування”

Таблиця “Опитування” містить такі поля, як ідентифікатор, ідентифікатор автора, назва, опис, дата створення та хеш опитування (таблиця 2.2).

Таблиця 2.2 — Структура таблиці “Опитування”

Ім’я поля	**Тип і розмір поля**	**Опис поля**
survey_id	int	Первинний ключ, автоінкремент
creator_id	int	Ідентифікатор автора (користувача)
title	varchar	Назва опитування
description	text	Опис опитування
created_at	date	Дата створення
hash	text	Унікальний хеш опитування

Таблиця “Опитування” має зв’язок з таблицею “Користувач” по ідентифікатору автора, так як виражає детальні дані, що стосуються кожного опитування.

Таблиця “Питання”

Таблиця “Питання” містить такі поля, як ідентифікатор, ідентифікатор опитування, тип питання, текст питання та емоції, пов’язані з питанням (таблиця 2.3).

Таблиця 2.3 — Структура таблиці “Питання”

Ім’я поля	**Тип і розмір поля**	**Опис поля**
question_id	int	Первинний ключ, автоінкремент
survey_id	int	Ідентифікатор опитування
type	varchar	Тип питання (text, image, video)
question	text	Текст питання
emotions	varchar[]	Масив емоцій, пов'язаних з питанням

Таблиця “Питання” має зв’язок з таблицею “Опитування” по ідентифікатору опитування.

Таблиця “Відповіді”

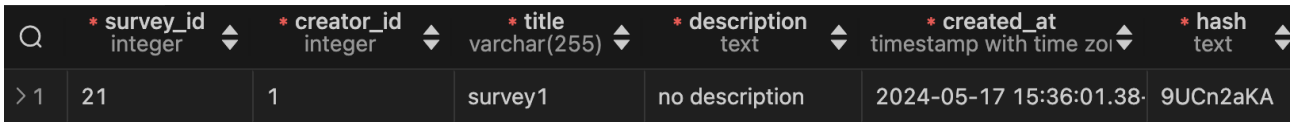
Таблиця “Відповіді” містить такі поля, як ідентифікатор, хеш питання, ідентифікатор питання, ідентифікатор користувача, значення відповіді та сесія (таблиця 4.4).

Таблиця 2.4 — Структура таблиці “Відповіді”

Ім’я поля	**Тип і розмір поля**	**Опис поля**
response_id	int	Первинний ключ, автоінкремент
question_hash	varchar	Хеш питання
question_id	int	Ідентифікатор питання
user_id	int	Ідентифікатор користувача
value	int	Значення відповіді
session_id	varchar	Ідентифікатор сесії

Таблиця “Відповіді” має зв’язок з таблицями “Користувач” та “Питання” по відповідним ідентифікаторам, виражаючи відповіді користувачів на конкретні питання.

Реляційна модель баз даних широко використовується для організації даних у вигляді таблиць. Фрагменти реляційної моделі дозволяють візуалізувати структуру та зв'язки між даними, полегшуючи розуміння і проектування баз даних (рис. 2.4)[15].



	* survey_id integer	* creator_id integer	* title varchar(255)	* description text	* created_at timestamp with time zo	* hash text
> 1	21	1	survey1	no description	2024-05-17 15:36:01.38	9UCn2aKA

Рисунок 2.4 – Фрагмент реляційної моделі бази даних.

Зв'язки між таблицями

- Таблиця “Користувач” має зв’язок один-до-багатьох з таблицею “Опитування”, оскільки один користувач може створити багато опитувань.

- Таблиця “Опитування” має зв’язок один-до-багатьох з таблицею “Питання”, оскільки одне опитування може містити багато питань.

- Таблиця “Питання” має зв’язок один-до-багатьох з таблицею “Відповіді”, оскільки одне питання може мати багато відповідей.

- Таблиця “Відповіді” має зв’язок багато-до-одного з таблицями “Користувач” та “Питання”, оскільки кожна відповідь належить одному користувачу і одному питанню.

Опис класів у програмному модулі

У програмному модулі, що розробляється, будуть такі класи: User, Survey, Question, Response, які слугують для відображення сутностей бази даних.

Опис класів

- Клас User містить поля для зберігання інформації про користувача, такі як user_id, email, name, picture, password, та last_login.

- Клас Survey представляє опитування та містить поля survey_id, creator_id, title, description, created_at, hash, та зв'язок з питаннями через поле questions.

- Клас Question представляє питання та містить поля question_id, survey_id, type, question, emotions, та зв'язок з відповідями через поле responses.

- Клас Response представляє відповідь на питання та містить поля response_id, question_hash, question_id, user_id, value, session_id, та зв'язки з Question і User.

Клас `User` має багато `Survey` (опитувань), які створює користувач. Клас `Survey` включає багато `Question` (питань), що належать до конкретного опитування. Клас `Question` включає багато `Response` (відповідей), що належать до конкретного питання. Клас `Response` асоційований з одним `User` та одним `Question`, представляючи відповідь користувача на конкретне питання.

Така структура дозволяє ефективно організувати взаємодію між різними частинами системи, забезпечуючи логічну цілісність та зручність у розробці та підтримці коду.

2.5 Висновок

У цьому розділі розглянуто ключові аспекти розробки серверної частини веб-ресурсу для проведення опитувань, використовуючи сучасні технології Nest.js та PostgreSQL. Використання Nest.js забезпечує високу продуктивність, надійність і масштабованість системи завдяки своїй модульній архітектурі, яка спрощує процес розробки, тестування та підтримки додатку. Підтримка TypeScript зменшує кількість помилок у коді та підвищує продуктивність розробників.

У проекті використовується JSON Web Token (JWT) для автентифікації користувачів, що забезпечує безпечну передачу інформації між клієнтом і сервером, гарантуючи цілісність даних і автентичність джерела запитів, а також знижуючи навантаження на сервери автентифікації. Вибір PostgreSQL як системи керування базами даних обґрунтовано її високою продуктивністю, надійністю та можливостями для розширення. Використання ORM Sequelize у

поєднанні з Nest.js дозволяє легко працювати з базою даних, виконуючи всі необхідні операції з даними.

Представлена UML-діаграма класів ілюструє структуру системи та взаємозв'язки між її компонентами, що допомагає візуалізувати структуру програми, краще зрозуміти сутність проблеми та визначити важливі аспекти при розробці. Загалом, розробка серверної частини веб-ресурсу для проведення опитувань з використанням сучасних технологій забезпечує високу продуктивність, надійність та безпеку системи. Інтеграція JWT для автентифікації користувачів гарантує захист даних, а модульна архітектура Nest.js дозволяє легко розширювати функціональність додатку. Це сприяє створенню ефективного, зручного та безпечного веб-ресурсу для проведення опитувань, який відповідає сучасним вимогам та задовольняє потреби користувачів.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ СЕРВЕРНОЇ ЧАСТИНИ

3.1 Обгрунтування вибору середовища розробки

Для розробки веб-ресурсу для проведення опитувань, який використовує технології Nest.js, PostgreSQL та Docker, важливо обрати інтегроване середовище розробки (IDE), яке найкраще підходить для цих завдань. Розглянемо три основні IDE: Visual Studio, Visual Studio Code та JetBrains Rider.

Visual Studio є потужним інтегрованим середовищем розробки, яке підтримує багато мов програмування, включаючи C++, Python та JavaScript. Воно пропонує повний набір інструментів для розробки, налагодження та тестування програмного забезпечення. Visual Studio включає в себе можливості для розробки веб-, хмарних, настільних, мобільних та ігрових додатків. Однак, Visual Studio переважно орієнтоване на розробку на платформі .NET, що робить його менш зручним для проектів на базі Nest.js та PostgreSQL. Крім того, Visual Studio може бути важким для системи та вимагати багато ресурсів [16].

Visual Studio Code є легким та потужним редактором вихідного коду, який працює на платформах Windows, macOS та Linux. Він підтримує різні мови програмування, включаючи JavaScript, TypeScript, Python та багато інших. Основна перевага цього редактора полягає у його гнучкості завдяки підтримці численних розширень. Це робить його ідеальним вибором для розробки з використанням Nest.js, PostgreSQL та Docker [17].

Розширення для PostgreSQL дозволяє підключатися до бази даних, виконувати SQL-запити, переглядати структуру таблиць і дані безпосередньо з IDE. Розширення для Docker забезпечує зручне керування контейнерами, образами та оркестрацією, що спрощує розгортання та тестування додатків у контейнеризованому середовищі. Visual Studio Code також підтримує розширення для Nest.js, що допомагає в розробці, налагодженні та тестуванні додатків.

JetBrains Rider – це кросплатформне інтегроване середовище розробки для додатків на базі мови програмування JavaScript, TypeScript та інших. Rider побудований на базі платформи IntelliJ і включає в себе потужні інструменти для аналізу, редагування, рефакторингу та налагодження коду. Він підтримує різні типи проектів, включаючи веб-додатки на базі Nest.js [18].

Rider пропонує інтеграцію з Docker і базами даних, що дозволяє ефективно керувати контейнерами та базами даних безпосередньо з IDE. Однак, як і Visual Studio, JetBrains Rider є платним середовищем розробки з відносно високою ціною, що може бути недоліком для деяких розробників.

З урахуванням специфічних завдань проекту, таких як розробка на базі Nest.js, робота з PostgreSQL, використання та Docker для контейнеризації, найбільш підходящим середовищем розробки є Visual Studio Code. Цей вибір обумовлений легкістю, швидкістю роботи, широкою підтримкою розширень для необхідних технологій, а також зручною інтеграцією з Docker та PostgreSQL. Visual Studio Code забезпечує ефективний процес розробки, тестування та розгортання додатка, що робить його оптимальним вибором для цього проекту.

3.2 Розробка та реалізація серверної частини та маршрутизація API

Для реалізації взаємодії з базою даних у проекті використовувався модуль Sequelize у поєднанні з Nest.js. Це дозволяє створювати та керувати реляційними базами даних за допомогою об'єктно-реляційного маппінгу (ORM).

Підключення та налаштування модуля бази даних

Модуль бази даних включає конфігурацію підключення до бази даних PostgreSQL, а також визначення моделей бази даних. Ось код для створення модуля бази даних:

```
@Module({
  imports: [
    // Налаштування підключення до бази даних
    SequelizeModule.forRootAsync({
```

```
      imports: [ConfigModule],
      inject: [ConfigService],
      useFactory: (configService: ConfigService) => ({
        dialect: 'postgres',
        host: configService.get<string>('pghost'),
        port: configService.get<number>('pgport'),
        username: configService.get<string>('pgusername'),
        password: configService.get<string>('pgpassword'),
        database: configService.get<string>('pgdatabase'),
        models: [Survey, User, Question, Response],
        dialectOptions: {
          pool: false,
        },
      }),
    }),
    SequelizeModule.forFeature([Survey, User, Question, Response]),
    LoggingModule,
  ],
  providers: [DatabaseService],
  exports: [DatabaseService, SequelizeModule],
})
export class DatabaseModule {}
```

SequelizeModule.forRootAsync: Цей метод використовується для асинхронного налаштування підключення до бази даних. Він дозволяє динамічно отримувати конфігураційні параметри з використанням ConfigService.

imports: У цьому розділі імпортуються необхідні модулі, такі як ConfigModule для конфігурацій та LoggingModule для логування.

useFactory: Ця функція використовує ConfigService для отримання параметрів підключення до бази даних, таких як хост, порт, ім'я користувача, пароль та ім'я бази даних.

models: Тут перелічуються моделі бази даних, які будуть використовуватись у проекті. В даному випадку це Survey, User, Question, Response.

SequelizeModule.forFeature: Цей метод використовується для реєстрації моделей бази даних, щоб вони були доступні в інших частинах додатку.

Сервіс для роботи з базою даних забезпечує методи для взаємодії з моделями бази даних. Ось приклад коду для DatabaseService:

```
Injectable()
export class DatabaseService {
  constructor(
    private readonly logger: LoggingService,

    @InjectModel(User)
    private userModel: typeof User,

    @InjectModel(Survey)
    private SurveyModel: typeof Survey,

    @InjectModel(Question)
    private QuestionModel: typeof Question,

    @InjectModel(Response)
    private responseModel: typeof Response,
  ) {
    // Синхронізація моделей з базою даних
    User.sync({ force: false })
    Survey.sync({ force: false })
    Question.sync({ force: false })
    Response.sync({ force: false })
  }
```

```
  async findOrCreateUser(userData: any): Promise<User> {
    const [user, _] = await User.findOrCreate({
      where: { email: userData.email },
      defaults: {
        name: userData.name,
        picture: userData.picture,
        password: userData.password,
        last_login: new Date(),
      },
    });
    return user;
  }
}
```

DatabaseService: Сервіс, який забезпечує методи для взаємодії з моделями бази даних.

constructor: Конструктор сервісу, який інжектує моделі бази даних (User, Survey, Question, Response) та сервіс логування (LoggingService).

sync: Методи sync використовуються для синхронізації моделей з базою даних. Якщо таблиці ще не існують, вони будуть створені.

findOrCreateUser: Метод для пошуку або створення нового користувача на основі наданих даних.

findAllQuestiones, findAllSurveys: Методи для отримання всіх питань та опитувань відповідно.

findOneUser: Метод для пошуку користувача за його ідентифікатором.

saveNewQuestion, saveNewSurvey, saveNewUser: Методи для збереження нових записів у відповідні таблиці бази даних.

Таким чином, створення та налаштування модуля бази даних забезпечує ефективну взаємодію з базою даних PostgreSQL та використання моделей для зберігання та отримання даних.

Модуль SurveyModule відповідає за реєстрацію сервісу та контролера для обробки опитувань у додатку Nest.js. Він імпортує необхідні компоненти та надає їх для використання у проекті. Цей модуль організовує всі пов'язані з опитуваннями компоненти, щоб зробити їх доступними для всього додатка.

Реєстрація сервісу та контролера

Модуль SurveyModule забезпечує реєстрацію сервісу SurveyService та контролера SurveyController, які відповідають за обробку HTTP-запитів.

```
import { Module } from '@nestjs/common';
import { SurveyService } from './survey.service';
import { SurveyController } from './survey.controller';

@Module({
  providers: [SurveyService],
  controllers: [SurveyController],
})
export class SurveyModule {}
```

@Module: Декоратор, що позначає клас як модуль Nest.js.

providers: Масив провайдерів, доступних у модулі. У цьому випадку це SurveyService.

controllers: Масив контролерів, які обробляють HTTP-запити. У цьому випадку це SurveyController.

Контролер SurveyController відповідає за обробку HTTP-запитів, пов'язаних з опитуваннями. Він взаємодіє з сервісом SurveyService для виконання різних операцій з опитуваннями, таких як створення, отримання, оновлення та видалення опитувань.

Опис SurveyController

Контролер SurveyController використовує декоратори з пакета @nestjs/common для визначення маршрутів та захисту їх за допомогою аутентифікаційних гвардій.

@Controller('survey'): Декоратор, який вказує, що цей клас є контролером, і всі маршрути починаються з survey.

constructor(private readonly surveyService: SurveyService): Конструктор, який інжектує сервіс SurveyService для доступу до методів роботи з опитуваннями.

@Post('new'): Декоратор, що визначає маршрут POST-запиту для створення нового опитування. Використовується гвардія JwtAuthGuard для захисту маршруту.

@Get('created'): Декоратор для маршруту GET-запиту, який отримує всі створені користувачем опитування.

@Get('taken'): Декоратор для маршруту GET-запиту, який отримує всі опитування, які пройшов користувач.

Функція saveNewSurvey використовується для створення нового опитування. Вона отримує дані опитування від клієнта, зберігає їх у базі даних та повертає результат операції.

```
@Post('new')
@UseGuards(JwtAuthGuard)
saveNewSurvey(@Body() surveyData: any, @Req() req) {
  const userId = req.user.userId;
  return this.surveyService.saveSurvey(surveyData, userId);
}
```

У цій функції використовується декоратор @Post, що означає, що вона обробляє POST-запити на маршрут survey/new. Декоратор @UseGuards(JwtAuthGuard) захищає цей маршрут, дозволяючи доступ тільки аутентифікованим користувачам. Функція отримує тіло запиту через декоратор @Body та об'єкт запиту через декоратор @Req. Вона викликає метод saveSurvey з сервісу SurveyService, передаючи дані опитування та ідентифікатор користувача.

Функція myCreatedSurveys повертає всі опитування, створені поточним користувачем.

```
@Get('created')
@UseGuards(JwtAuthGuard)
async myCreatedSurveys(@Req() req) {
  const { userId } = req.user;
  return await this.surveyService.getCreatedSurveys(userId);
}
```

Функція myCreatedSurveys використовує декоратор @Get для обробки GET-запитів на маршрут survey/created. Вона також захищена гвардією JwtAuthGuard. Функція отримує ідентифікатор користувача з об'єкта запиту і викликає метод getCreatedSurveys з сервісу SurveyService, який повертає всі опитування, створені цим користувачем.

Функція myTakenSurveys повертає всі опитування, пройдені поточним користувачем.

```
@Get('taken')
@UseGuards(JwtAuthGuard)
async myTakenSurveys(@Req() req) {
  const { userId } = req.user;
  return await this.surveyService.getTakenSurveys(userId);
}
```

Функція myTakenSurveys обробляє GET-запити на маршрут survey/taken і захищена гвардією JwtAuthGuard. Вона отримує ідентифікатор користувача з об'єкта запиту і викликає метод getTakenSurveys з сервісу SurveyService, який повертає всі опитування, які пройшов цей користувач.

Функція takeSurvey повертає дані опитування за його хешем.

```
@Get('take/:surveyHash')
async takeSurvey(@Req() req) {
  return await this.surveyService.getSurvey(req.params.surveyHash);
}
```

Функція takeSurvey обробляє GET-запити на маршрут survey/take/:surveyHash, де :surveyHash є параметром маршруту. Вона викликає метод getSurvey з сервісу SurveyService, передаючи хеш опитування, і повертає відповідні дані опитування.

Функція submitSurvey використовується для збереження відповідей на опитування. Вона отримує дані відповідей від клієнта, зберігає їх у базі даних та повертає результат операції.

```
@Post('submit')
@UseGuards(JwtAuthGuard)
async submitSurvey(@Body() surveyData: any, @Req() req) {
  const hash = surveyData.surveyHash;
  const { userId } = req.user;
  const sessionId = uuidv4();
  try {
    for (const questionId in surveyData.emotions) {
      if (surveyData.emotions.hasOwnProperty(questionId)) {
        const emotions = surveyData.emotions[questionId];
        for (const emotion in emotions) {
          if (emotions.hasOwnProperty(emotion)) {
            const value = parseInt(emotions[emotion], 10);
            await Response.create({
              question_id: parseInt(questionId, 10),
              question_hash: hash,
              user_id: userId,
              session_id: sessionId,
              value,
            });
          }
        }
      }
```

```
    }
    return {
      success: true,
      message: 'Survey responses stored successfully.',
    };
  } catch (error) {
    console.error('Error submitting survey:', error);
    return { success: false, message: 'Failed to store survey responses.' };
  }
}
```

Функція submitSurvey обробляє POST-запити на маршрут survey/submit і захищена гвардією JwtAuthGuard. Вона отримує дані відповідей на опитування з тіла запиту і зберігає їх у базі даних за допомогою моделі Response.

Функція getSurveyResults повертає результати опитування за його ID.

```
@Get(':id/results')
@UseGuards(JwtAuthGuard)
async getSurveyResults(@Param('id') id: number) {
  return this.surveyService.getSurveyResults(id);
}
```

Функція getSurveyResults обробляє GET-запити на маршрут survey/:id/results, де :id є параметром маршруту. Вона викликає метод getSurveyResults з сервісу SurveyService, передаючи ID опитування, і повертає відповідні результати.

Функція getSurveyById повертає дані опитування за його ID.

```
@Get(':id')
@UseGuards(JwtAuthGuard)
async getSurveyById(@Param('id') id: number) {
  const survey = await this.surveyService.getSurveyById(id);
  if (!survey) {
    throw new NotFoundException('Survey not found');
```

```
  }
  return survey;
 }
```

Функція getSurveyById обробляє GET-запити на маршрут survey/:id, де :id є параметром маршруту. Вона викликає метод getSurveyById з сервісу SurveyService, передаючи ID опитування, і повертає відповідні дані опитування. Якщо опитування не знайдено, кидається виняток NotFoundException.

Функція updateSurvey оновлює дані опитування за його ID.

```
@Put(':id')
@UseGuards(JwtAuthGuard)
async updateSurvey(
  @Param('id') id: number,
  @Body() updateSurveyDto: UpdateSurveyDto,
) {
  const updatedSurvey = await this.surveyService.updateSurvey(id, updateSurveyDto);
  if (!updatedSurvey) {
    throw new NotFoundException('Survey not found');
  }
  return updatedSurvey;
}
```

Функція updateSurvey обробляє PUT-запити на маршрут survey/:id, де :id є параметром маршруту. Вона викликає метод updateSurvey з сервісу SurveyService, передаючи ID опитування та дані для оновлення, і повертає оновлені дані опитування. Якщо опитування не знайдено, кидається виняток NotFoundException.

Функція updateQuestion оновлює дані питання в опитуванні за його ID.

```
@Put(':surveyId/questions/:questionId')
@UseGuards(JwtAuthGuard)
async updateQuestion(
```

```
    @Param('surveyId') surveyId: number,
    @Param('questionId') questionId: number,
    @Body() updateQuestionDto: UpdateQuestionDto,
  ) {
    const updatedQuestion = await this.surveyService.updateQuestion(surveyId, questionId, updateQuestionDto);
    if (!updatedQuestion) {
      throw new NotFoundException('Question not found');
    }
    return updatedQuestion;
  }
```

Функція updateQuestion обробляє PUT-запити на маршрут survey/:surveyId/questions/:questionId, де :surveyId та :questionId є параметрами маршруту. Вона викликає метод updateQuestion з сервісу SurveyService, передаючи ID опитування, ID питання та дані для оновлення, і повертає оновлені дані питання. Якщо питання не знайдено, кидається виняток NotFoundException.

Функція getSurveyResultsAll повертає всі результати опитувань.

```
@Get(':id/results-all')
@UseGuards(JwtAuthGuard)
async getSurveyResultsAll() {
  return this.surveyService.getAllSurveyResults();
}
```

Функція getSurveyResultsAll обробляє GET-запити на маршрут survey/:id/results-all і захищена гвардією JwtAuthGuard. Вона викликає метод getAllSurveyResults з сервісу SurveyService, який повертає всі результати опитувань.

Ці функції контролера SurveyController забезпечують повний набір операцій для створення, отримання, оновлення та збереження результатів опитувань у системі.

Після розгляду функцій контролера SurveyController стає зрозуміло, що для виконання різноманітних операцій з опитуваннями потрібен сервіс, який буде забезпечувати логіку цих операцій. Цей сервіс називається SurveyService.

SurveyService включає в себе такі основні функції:

saveSurvey: Створення та збереження нового опитування разом з його питаннями.

- getSurvey: Отримання опитування за його хешем.
- getCreatedSurveys: Отримання всіх опитувань, створених певним користувачем.
- getTakenSurveys: Отримання всіх опитувань, які пройшов користувач.
- getSurveyById: Отримання опитування за його ідентифікатором.
- updateSurvey: Оновлення даних опитування за його ідентифікатором.
- updateQuestion: Оновлення даних питання в опитуванні за його ідентифікатором.
- getSurveyResults: Отримання результатів опитування за його ідентифікатором.
- getAllSurveyResults: Отримання результатів всіх опитувань.

Ці функції забезпечують необхідну логіку для взаємодії з базою даних, включаючи створення, отримання, оновлення та збереження результатів опитувань.

Для забезпечення безпеки та управління доступом у нашому проекті використовується JWT (JSON Web Token). JWT дозволяє створювати токени доступу, які можуть бути використані для автентифікації та авторизації користувачів.

Підключення та налаштування JWT модуля

Почнемо з опису модуля JwtAuthModule, який включає в себе налаштування JWT, використовуючи бібліотеку @nestjs/jwt.

```
@Module({
  imports: [
    JwtModule.registerAsync({
```

```
      imports: [ConfigModule],
      inject: [ConfigService],
      useFactory: (configService: ConfigService) => {
        const jwtSecret = configService.get<string>('jwtSecret');
        if (!jwtSecret) {
          throw new Error('JWT secret key is not defined in configuration.');
        } else {
          console.log('JWT secret key is defined in configuration.');
        }

        return {
          secret: jwtSecret,
          signOptions: { expiresIn: '24h' },
        };
      },
    }),
  ],
  providers: [JwtAuthGuard, JwtStrategy, ConfigService],
  exports: [JwtAuthGuard, JwtStrategy, JwtModule],
})
export class JwtAuthModule {}
```

Цей модуль забезпечує асинхронне налаштування JWT, використовуючи ConfigService для отримання секретного ключа (jwtSecret). Основні компоненти, що використовуються (рис. 3.1):

JwtModule.registerAsync: Використовується для асинхронного налаштування JWT модуля.

useFactory: Функція, яка забезпечує конфігурацію для JWT, включаючи секретний ключ і опції підпису.

Наступні компоненти визначають JwtAuthGuard та JwtStrategy, які використовуються для автентифікації запитів.

JwtAuthGuard

```
import { Injectable } from '@nestjs/common';
import { AuthGuard } from '@nestjs/passport';

@Injectable()
export class JwtAuthGuard extends AuthGuard('jwt') {}
```

JwtAuthGuard є класом, який розширює `AuthGuard` з пакету @nestjs/passport і використовується для захисту маршрутів, що потребують автентифікації. Цей клас дозволяє використовувати JWT для аутентифікації користувачів, забезпечуючи, щоб лише авторизовані користувачі могли отримати доступ до захищених ресурсів.

JwtStrategy

```
typescript
import { Injectable } from '@nestjs/common';
import { ConfigService } from '@nestjs/config';
import { PassportStrategy } from '@nestjs/passport';
import { ExtractJwt, Strategy } from 'passport-jwt';

@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy) {
  constructor(private readonly configService: ConfigService) {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      secretOrKey: configService.get<string>('jwtSecret'),
      ignoreExpiration: false,
    });
  }

  async validate(payload: any) {
```

```
    return { userEmail: payload.userEmail, userId: payload.userId };
  }
}
```

JwtStrategy є класом, що реалізує стратегію JWT для Passport. Він використовує `passport-jwt` для вилучення та валідації токенів з заголовка авторизації. Стратегія визначає, як отримати та перевірити JWT, використовуючи такі ключові параметри:

- jwtFromRequest: Вказує, як вилучати JWT з запиту. В даному випадку, токен вилучається з заголовка авторизації як Bearer токен.

- secretOrKey: Секретний ключ, що використовується для підпису та валідації JWT. Він отримується з конфігураційного сервісу ConfigService.

- ignoreExpiration: Вказує, чи потрібно ігнорувати термін дії токена. У даному випадку встановлено false, що означає, що прострочені токени не будуть прийматися.

Метод validate викликається після валідації JWT і повертає об'єкт payload, який містить дані користувача (наприклад, userEmail та userId). Цей об'єкт буде доступний у запиті як req.user, що дозволяє легко отримувати інформацію про аутентифікованого користувача в обробниках маршрутів.

Ці компоненти інтегруються в додаток NestJS для забезпечення захисту маршрутів за допомогою JWT. Наприклад, у контролері можна використовувати JwtAuthGuard, щоб захистити певний маршрут

```
import { Controller, Get, UseGuards } from '@nestjs/common';
@UseGuards(JwtAuthGuard)
getProtectedResource() { return “protected”;
}
```

Таким чином, лише аутентифіковані користувачі з дійсним JWT зможуть отримати доступ до цього маршруту. Це забезпечує безпечний та масштабований спосіб аутентифікації користувачів у вашому додатку.

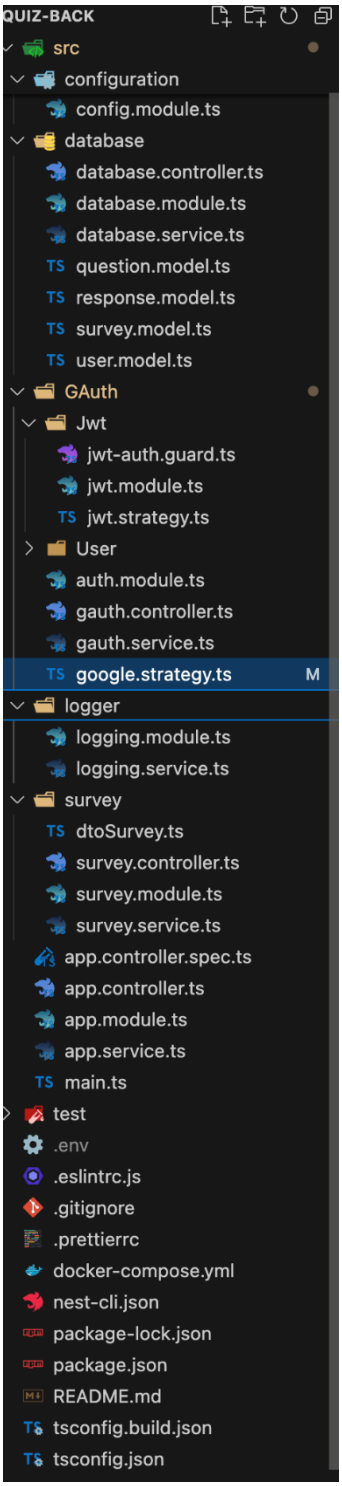


Рисунок 3.1 – Структура проекту у Visual Studio Code

3.3 Розробка API документації

Розробимо документацію HTTP-запитів, пов'язаних з опитуваннями [19].

Створення нового користувача

Запит: POST auth/signup

Опис: Цей ендпоїнт дозволяє створити нового користувач. Запит повинен містити дані користувача.

Вхідні параметри

{
"name":"Vitalii",
"email":"youremail@gmail.com",
"password":"123456789",
"picture":0
}

Параметри:

- name: Ім'я користувача.
- email: Електронна адреса користувача.
- password: Пароль користувача.
- picture: Ідентифікатор аватару користувача (0, якщо не використовується, буде стандартна картинка)

Позитивна відповідь:

{
"success":true,
"code":400,
"accessToken":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyRW1haWwiOiJ5b3VyZW1haWxAZ21haWwuY29tIiwidXNlcklkIjoyLCJpYXQiOjE3MTY5MjMxMjksImV4cCI6MTcxNzAwOTUyOX0.yxDLB2j0VuMW2-8Pr7uJeBDOlgvekQYeCqWDi6tuM"
}

accessToken (string): JWT токен для аутентифікації користувача.

Негативна відповідь:

{
"success": false,
"msg": "Not provided param: name"
}

Отримати JWT токен через пошту пароль

Запит: POST /auth/login

Опис: Цей ендпоїнт дозволяє користувачеві увійти до системи. Запит повинен містити електронну пошту та пароль користувача.

Вхідні параметри:

{
"email": "youremail@gmail.com",
"password": "123456789"
}

Параметри:

- email: Електронна адреса користувача.
- password: Пароль користувача.

Позитивна відповідь:

{
"success": true,
"code": 200,
"accessToken":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyRW1haWwiOiJ5b3VyZW1ha
WxAZ21haWwuY29tIiwidXNlcklkIjoyLCJpYXQiOjE3MTY5MjQwNjYsImV4cCI
6MTcxNzAxMDQ2Nn0.jKfaNZs2rPVv1H9IveJEv59xtMoq3agUahw0NIOdsDE"
}

accessToken (string): JWT токен для аутентифікації користувача.

Негативна відповідь:

{
"success":false,
"code":401
}

Авторизація через гугл аккаунт

Запит: /auth/google/callback

Опис: Цей ендпоїнт дозволяє користувачеві увійти через гугл аккаунт. Після авторизації/регістрації, сервер перенаправить на домен фронтенду з JWT токеном

Позитивна відповідь:

Редірект на домен сайту з параметром ?token=TOKEN

token: JWT токен для аутентифікації користувача.

Негативна відповідь:

```
{
"success":false,
"msg" “error occurred“
}
```

Створення нового опитування

Запит: POST /survey/new

Опис: Цей ендпоїнт дозволяє створити нове опитування. Запит повинен містити дані опитування.

Вхідні параметри

```
{
  "questions": [
    {
      "type": "TYPE"
      "answer": "QUESTION",
      "emotions": [EMOTIONS]
    }
  ],
  "hash": "HASH",
  "title": "TITLE",
  "description": "DESCRIPTION"
}
```

questions: масив об'єктів питань.

TYPE: рядок, що визначає тип питання. Можливі значення: text, image, video.

- text: Текстове питання.
- image: Питання зображення.
- video: Відеопитання.

answer: рядок, що містить питання або посилання на зображення/відео для відповідного типу.

emotions: масив рядків, що містить емоції, які можна оцінити в питанні. Можливі значення: Радість, Смуток, Гнів, Страх, Здивування, Огида, Довіра, Очікування, та інші.

hash: рядок, унікальний хеш для опитування.

title: рядок, заголовок опитування.

description: рядок, опис опитування.

Заголовок:

Authorization: Bearer <your_jwt_token>

Вихідні параметри

Позитивна відповідь:

{
"survey_id": 25,
"creator_id": 1,
"title": "TITLE",
"description": "DESCRIPTION",
"created_at": "2024-05-25T10: 01:40. 577Z",
"hash": "HASH",
"updatedAt": "2024-05-25T10:01:40.5772",
"createdAt": "2024-05-2510:01:40. 5772"
}

Відповідь з помилками:

{

```
  "success": false,
  "message": "Missing required parameter: title"
}
```


Отримати усі створені опитування

Запит: GET /survey/created

Опис: Отримує всі опитування, створені поточним користувачем.

Заголовок:

Authorization: Bearer <your_jwt_token>

Параметри перевірки

Перевірка автентифікації користувача за допомогою JWT токена.

Вихідні параметри

Позитивна відповідь:


```
[
  {
    "survey_id": 21,
    "creator_id": 1,
    "title": "survey1",
    "description": "no description",
    "created_at": "2024-05-17T15:36:01.380Z",
    "hash": "9UCn2aKA",
    "createdAt": "2024-05-17T15:36:01.381Z",
    "updatedAt": "2024-05-18T13:18:50.379Z",
    "questions": [
      {
        "question_id": 7,
        "survey_id": 21,
        "type": "text",
        "question": "lets go changeit now",
        "emotions": ["Радість", "Гнів", "Страх"],
```

"createdAt": "2024-05-17T15:36:01.449Z",
"updatedAt": "2024-05-18T13:11:38.334Z"
}
]
}
]

Відповідь з помилками:

Якщо користувач не автентифікований:

{
"statusCode": 401,
"message": "Unauthorized"
}

Збереження відповідей на опитування

Запит: POST /survey/submit

Опис: Зберігає відповіді користувача на опитування.

{
"surveyHash": "sfxG87DH",
"emotions": {
"1": {
"Радість": "55",
"Смуток": "2"
}
}
}

- surveyHash: рядок, унікальний хеш для опитування.
- emotions: об'єкт, де ключами є ідентифікатори питань, а значеннями-об'єкти з емоціями та їх оцінками.

Заголовок:

Authorization: Bearer <your_jwt_token>

Параметри перевірки

Перевірка автентифікації користувача за допомогою JWT токена.

Перевірка наявності surveyHash та emotions.

Позитивна відповідь:

```
{
  "success": true,
  "message": "Survey responses stored successfully."
}
```

Відповідь з помилками:

```
{
  "success": false,
  "message": "Failed to store survey responses."
}
```

Якщо користувач не автентифікований:

```
{
  "statusCode": 401,
  "message": "Unauthorized"
}
```

Оновлення питання в опитуванні

Запит: PUT /survey/:surveyId/questions/:questionId

Опис: Оновлює дані питання в опитуванні за його ID.

Вхідні параметри:

```
{
  "type": "text",
  "question": "Новий текст питання",
  "emotions": ["Радість", "Смуток"]
}
```

- type: рядок, новий тип питання. Можливі значення: text, image, video.
- question: рядок, новий текст питання.

- emotions: масив рядків, нові емоції для оцінки.

Параметри перевірки

Перевірка автентифікації користувача за допомогою JWT токена.

Перевірка правильності форматів surveyId та questionId.

Вихідні параметри

Позитивна відповідь:


```
[
  {
    "survey_id": 21,
    "creator_id": 1,
    "title": "survey1",
    "description": "no description",
    "created_at": "2024-05-17T15:36:01.380Z",
    "hash": "9UCn2aKA",
    "createdAt": "2024-05-17T15:36:01.381Z",
    "updatedAt": "2024-05-18T13:18:50.379Z",
    "questions": [
      {
        "question_id": 7,
        "survey_id": 21,
        "type": "text",
        "question": "lets go changeit now",
        "emotions": ["Радість", "Гнів", "Страх", "Здивування", "Огида", "Довіра",
"Очікування"],
        "createdAt": "2024-05-17T15:36:01.449Z",
        "updatedAt": "2024-05-18T13:11:38.334Z"
      }
    ]
  }
]
```

Відповідь з помилками:

```
{
  "statusCode": 404,
  "message": "Question not found"
}
```

Отримання всіх результатів опитування

Запит: GET /survey/:id/results-all

Опис: Отримує всі результати опитувань.

Параметри:

- id: number

Заголовок:

Authorization: Bearer <your_jwt_token>

Параметри перевірки:

Перевірка автентифікації користувача за допомогою JWT токена.

Перевірка правильності формату id.

Вихідні параметри

Позитивна відповідь:

```
[
 {
  "survey": {
   "id": 1,
   "title": "Тайтл",
   "description": "Оцінка тварин",
   "createdAt": "2023-05-01T00:00:00.000Z",
   "updatedAt": "2023-05-02T00:00:00.000Z"
  },
  "questions": [
   {
    "questionId": 1,
```

```
        "questionText": "Кіт",
        "questionType": "text",
        "emotions": ["Радість", "Смуток", "Гнів"]
      }
    ],
    "responses": {
      "sessionId1": [
        {
          "questionId": 1,
          "value": 5,
          "createdAt": "2023-05-02T00:00:00.000Z",
          "responseId": 1,
          "sessionId": "sessionId1",
          "user": {
            "id": 1,
            "name": "Користувач"
          }
        }
      ]
    }
  }
]
```


Відповідь з помилками:

Якщо опитування не знайдено:


```
{
  "statusCode": 404,
  "message": "Survey not found"
}
```


Якщо користувач не автентифікований:


```
{
```

"statusCode": 401,
"message": "Unauthorized"
}

3.4 Тестування API за допомогою Postman

Для тестування API у цьому проекті використовувався Postman - потужний інструмент, що дозволяє розробникам легко надсилати запити до серверів і отримувати відповіді [20]. Тестування API за допомогою Postman включає кілька важливих етапів, які забезпечують повну перевірку функціональності кінцевих точок.

Спершу, необхідно завантажити та встановити Postman з офіційного сайту. Після встановлення можна створити новий акаунт або увійти в існуючий, щоб мати змогу зберігати та організовувати запити.

Для зручності організації тестів, створив нову колекцію в Postman. Колекції дозволяють групувати запити за певними критеріями, що полегшує управління тестами. Для створення колекції достатньо натиснути на кнопку "New", вибрати "Collection" і дати їй назву, наприклад, "Survey API Tests".

У межах цієї колекції додав кілька запитів для тестування різних кінцевих точок нашого API. Перший запит, який створив, перевіряє функцію реєстрації нового користувача та отримання JWT токену. Для цього використовується метод POST. Вибрав метод POST, ввів URL запиту (http://localhost:3300/auth/signup), перейшов до вкладки "Body" та встановив формат "raw" і тип "JSON". Далі ввів JSON-дані, необхідні для реєстрації користувача (рис. 3.1):

{
"name": "Gara",
"email": "gara@gmail.com",
"password": "123456789",
"picture": 0

}

На зображенні нижче показано приклад запиту у Postman для реєстрації користувача:

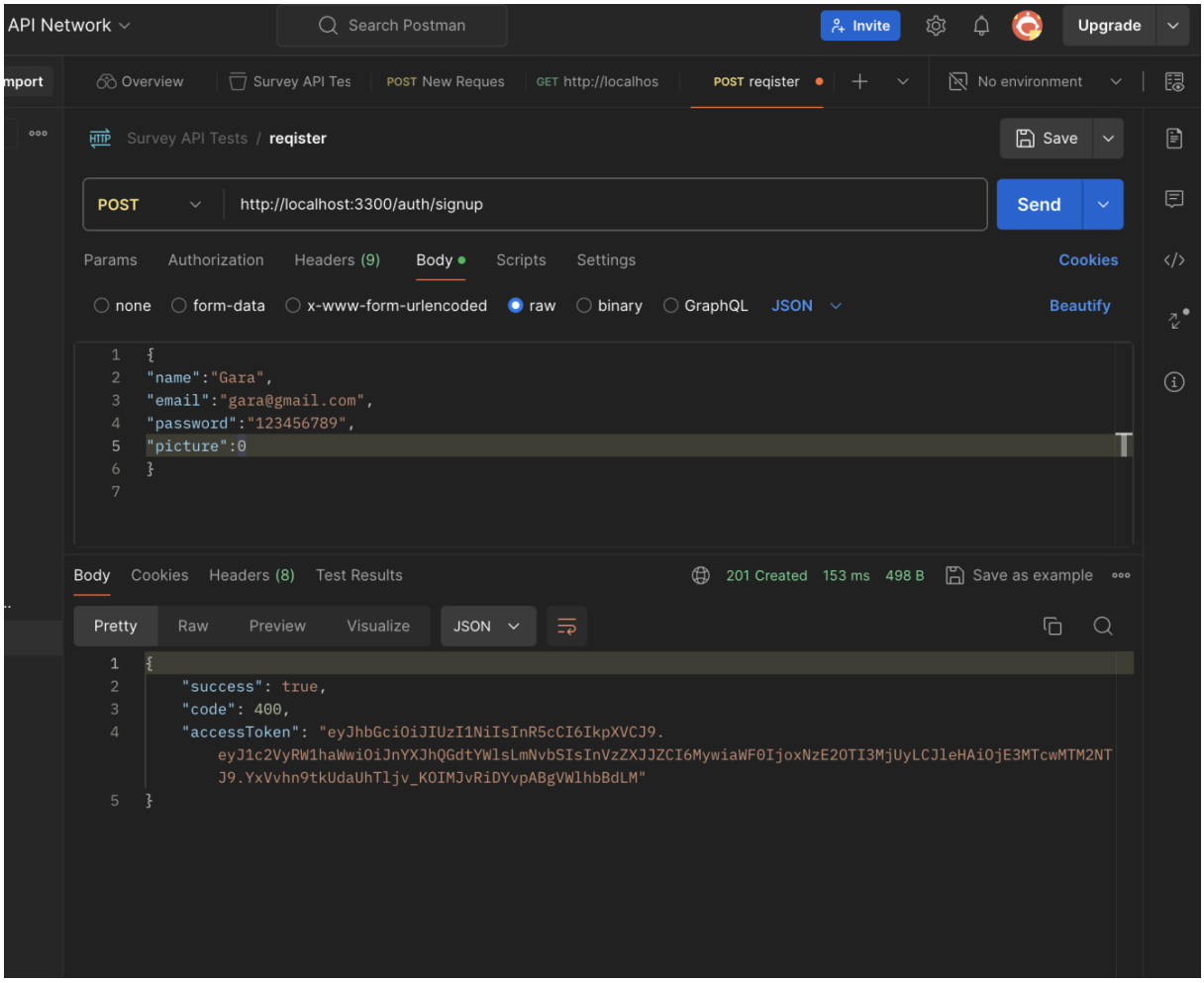


Рисунок 3.1 - Результат API запиту GET /auth/signup

Цей запит повертає JWT токен, який нам потрібен для наступних запитів.

Наступний запит перевіряє функцію створення нового опитування. Для цього також використовується метод POST. Вивів URL запиту (/survey/new), перейшов до вкладки "Body" та встановив формат "raw" і тип "JSON". Далі ввів JSON-дані, необхідні для створення опитування: {

"questions": [

{

"type": "image",

"answer":

"https://www.google.com/url?sa=i&url=https%3A%2F%2Fwww.nationalgeographic.

com%2Fanimals%2Fmammals%2Ffacts%2Fdomestic-dog&psig=AOvVaw0O5RHyMWuQ-1Z39WrAGt96&ust=1716717261955000&source=images&cd=vfe&opi=89978449&ved=0CBEQjRxqFwoTCKjltKbEqIYDFQAAAAAdAAAAABAE",
"emotions": [
"Радість",
"Смуток",
"Гнів",
"Страх",
"Здивування",
"Огида",
"Довіра",
"Очікування"
]
},
{
"answer": "Кіт",
"type": "text",
"emotions": [
"Радість",
"Смуток",
"Гнів",
"Страх",
"Здивування",
"Огида",
"Довіра",
"Очікування"
]
}
],

"hash": "sfxG87DH",

"title": "Тайтл",

"description": "Оцінка тварин"

}

Для авторизації у заголовках запиту додав заголовок Authorization з JWT токеном: Authorization: Bearer <your_jwt_token>. Натиснувши "Send", відправив запит і перевірив відповідь від сервера, яка повинна була підтвердити успішне створення опитування (рис. 3.2).

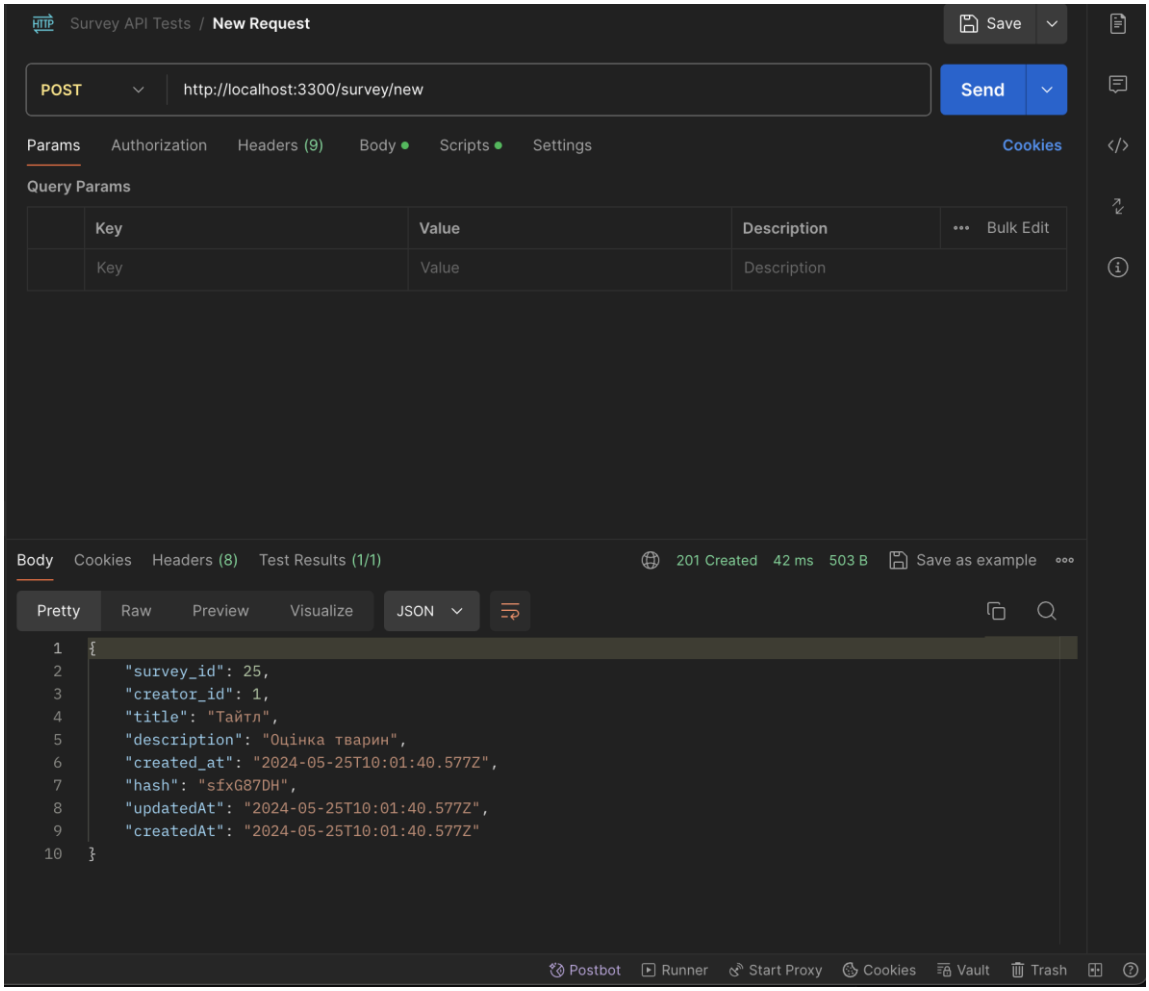


Рисунок 3.2 – Результат API запиту GET /survey/new

Наступний запит, який додав до колекції, перевіряв отримання створених опитувань. Для цього використовується метод GET. Вибрав метод GET, ввів URL запиту (/survey/created), додав заголовок авторизації та відправив запит, отримуючи список створених опитувань (рис. 3.3).

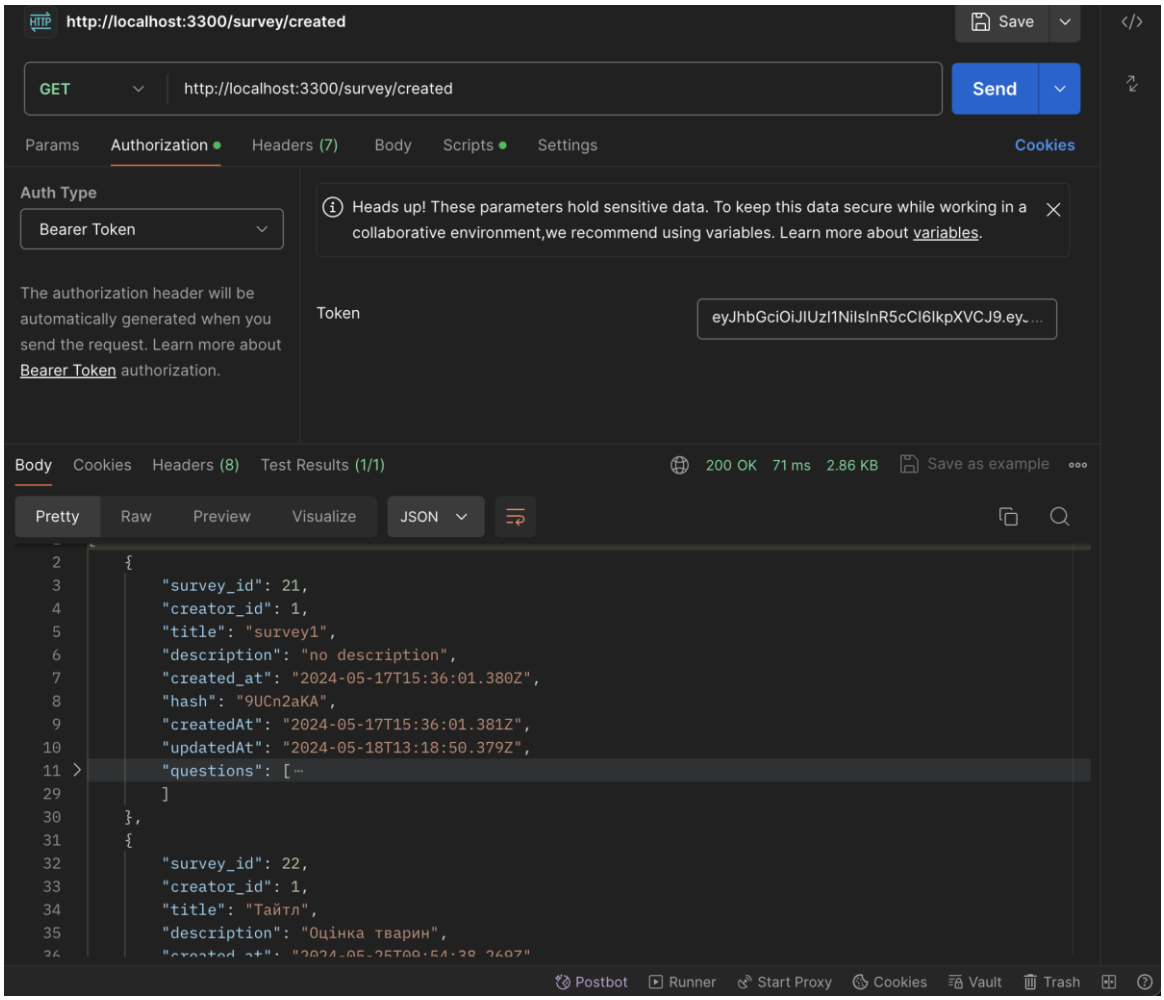


Рисунок 3.3 – Результат API запиту GET /survey/created

Після кожного запиту перевіряв відповіді від сервера. Відповіді мали містити відповідні статус-коди, такі як 200 OK для успішних запитів, а також коректні дані. Це допомагало переконатися, що API працює правильно та відповідає очікуванням.

Postman також дозволяє автоматизувати тестування за допомогою скриптів. У вкладці "Tests" кожного запиту додавав тести для автоматичної перевірки відповідей. Наприклад, для перевірки статус-коду відповіді використовував наступний код:

```
pm.test("Status code is 200", function () {
pm.response.to.have.status(200);
});
```

Цей процес дозволив мені створити комплексну тестову стратегію для API, забезпечуючи його надійність і коректність функціонування.

3.5 Висновок

Розробка серверної частини веб-ресурсу для проведення опитувань за допомогою технологій Nest.js, PostgreSQL та Docker була успішно завершена, що підтверджується результатами проведеного тестування та аналізу функціональності системи. Обране середовище розробки Visual Studio Code забезпечило легкість і швидкість роботи, а також підтримку необхідних розширень для Nest.js, PostgreSQL та Docker, що зробило його оптимальним для виконання задач проекту.

Створення API для взаємодії з базою даних PostgreSQL через модуль Sequelize у поєднанні з Nest.js дозволило ефективно реалізувати реляційні бази даних через об'єктно-реляційний маппінг (ORM). Це забезпечило стабільну та надійну роботу серверної частини веб-застосунку. Сервіс забезпечує повний набір функцій для роботи з опитуваннями, включаючи створення, отримання, оновлення та видалення опитувань, а також збереження та обробку відповідей користувачів, що дозволило досягти поставлених цілей щодо забезпечення зручності використання та надійності системи.

Розроблений модуль SurveyModule організовує всі компоненти, пов'язані з опитуваннями, роблячи їх доступними для всього додатка, що забезпечило зручну структуру для управління опитуваннями та їхніми компонентами. Документація API, розроблена для веб-ресурсу, дозволяє легко інтегрувати клієнтську частину з сервером, забезпечуючи стабільну взаємодію між користувачами та системою. Це полегшує процес розробки клієнтських застосунків, надаючи чіткі інструкції та приклади використання.

Тестування API за допомогою Postman підтвердило правильність роботи кінцевих точок, відповідність очікуваним результатам та стабільність системи. Тестування включало перевірку автентифікації, створення та отримання опитувань, збереження відповідей та оновлення даних.

Таким чином, розробка серверної частини веб-ресурсу для проведення опитувань була успішно завершена, досягнувши всіх поставлених цілей.

Система забезпечує високу продуктивність, надійність та зручність використання, що сприяє ефективному збору та аналізу даних. Такий ресурс є актуальним та необхідним інструментом для організацій, які прагнуть оперативно отримувати зворотний зв'язок від широкої аудиторії, що дозволяє приймати обґрунтовані рішення та підвищувати якість управління в різних сферах діяльності. Усі етапи проекту були успішно виконані, а розроблений веб-ресурс відповідає заявленим вимогам та очікуванням.

ВИСНОВКИ

У цій бакалаврській дипломній роботі було успішно реалізовано серверну частину веб-ресурсу для проведення опитувань, що включає інтеграцію з Nest.js, PostgreSQL та Docker. Всі задачі, поставлені на початку роботи, виконані в повному обсязі.

Обґрунтування доцільності розробки показало актуальність створення такого ресурсу, оскільки він дозволяє автоматизувати та спростити процес збору та аналізу спектру емоцій від користувачів. Було детально проаналізовано та обрано оптимальні технології для реалізації поставлених задач, що включали використання Visual Studio Code як основного середовища розробки.

Проектування веб-ресурсу охоплювало розробку архітектури системи, UML-діаграми класів та опис методів. Було створено модулі, що забезпечують функціональність бази даних, обробку опитувань та аутентифікацію користувачів за допомогою JWT. Це дозволило чітко структурувати процеси і забезпечити їх надійну реалізацію.

Програмна реалізація включала написання коду для всіх запланованих модулів та сервісів з використанням Nest.js та PostgreSQL. Модуль для роботи з опитуваннями дозволяє створювати, зберігати та аналізувати опитування, а модуль аутентифікації забезпечує безпечний доступ до системи.

Тестування програми здійснювалось за допомогою інструменту Postman, що дозволило перевірити коректність роботи API. Було ретельно протестовано функціональність створення, отримання та збереження опитувань, а також обробку відповідей від користувачів. Результати тестування підтвердили, що веб-ресурс відповідає всім заданим вимогам та працює стабільно.

Розроблений веб-ресурс має розширений функціонал, що відрізняється від будь-яких інших веб-ресурсів для проходження опитувань. Це робить його унікальним та корисним інструментом для збору та аналізу спектру емоцій користувачів. Мета роботи була досягнута, і всі поставлені задачі виконані успішно.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Ганевич В. М., Гарук В.В., Колодний В.В. Аналіз передумов розробки WEB-застосунку для проходження опитувань. LIII Науково-технічна конференція факультету інтелектуальних інформаційних технологій та автоматизації. URL: https://conferences.vntu.edu.ua/index.php/all-fksa/all-fksa-2024/paper/view/20883 (дата звернення: 06.05.2024).
2. Google Forms. URL: https://www.google.com/forms/about/ (дата звернення: 28.05.2024).
3. Survio | Free Online Survey Software | Create your own survey in minutes. URL: https://www.survio.com/ (дата звернення: 28.05.2024).
4. Surveynuts | Free Online Survey Tool. URL: https://www.surveynuts.com/ (дата звернення: 28.05.2024).
5. eSurvey. Переваги онлайн опитувань. URL: https://esurvey.com.ua/info/поради-щодо-опитування/переваги-онлайн-опитувань-11 (дата звернення: 28.05.2024).
6. TypeScript: Typed JavaScript at Any Scale. URL: https://www.typescriptlang.org/ (дата звернення: 28.05.2024).
7. NestJS: A progressive Node.js framework for building efficient, reliable and scalable server-side applications. URL: https://nestjs.com/ (дата звернення: 28.05.2024).
8. Sequelize: An easy-to-use multi SQL dialect ORM for Node.js. URL: https://sequelize.org/ (дата звернення: 28.05.2024)
9. Node.js: JavaScript runtime built on Chrome's V8 JavaScript engine. URL: https://nodejs.org/ (дата звернення: 28.05.2024).
10. JSON Web Token (JWT) Authentication: Best Practices and When to Use It. URL: https://blog.logrocket.com/jwt-authentication-best-practices/ (дата звернення: 28.05.2024).

11. ORM (Object-Relational Mapping) [Електронний ресурс]. – Режим доступу: https://uk.wikipedia.org/wiki/Об%27єктно-реляційне_відображення (дата звернення: 23.05.2024).
12. Understanding PostgreSQL and its Strong Embrace of ACID Properties. URL: https://blog.logrocket.com/jwt-authentication-best-practices/ (дата звернення: 28.05.2024).
13. Auth. Authorization in NestJS with Roles and Guards. URL: https://auth0.com/blog/authorization-in-nestjs-with-roles-and-guards/ (дата звернення: 28.05.2024).
14. Docs NestJS - Модульність. URL: https://docs.nestjs.com/modules (дата звернення: 28.05.2024).
15. Реляційна модель даних у СУБД. URL: https://www.guru99.com/uk/relational-data-model-dbms.html (дата звернення: 28.05.2024)
16. Visual Studio. Visual Studio Development Features. URL: https://visualstudio.microsoft.com (дата звернення: 28.05.2024).
17. Visual Studio Code. URL: https://code.visualstudio.com (дата звернення: 28.05.2024).
18. JetBrains. JetBrains Rider: The Best IDE for .NET & ASP.NET. URL: https://www.jetbrains.com/rider/ (дата звернення: 28.05.2024).
19. Документування API пир розробці. URL: https://tqm.com.ua/ua/likbez/api-wiki/avtomatychna-dokumentatsiya-api-web-service (дата звернення: 28.05.2024).
20. FreeCodeCamp. Як використовувати API з Postman: покроковий посібник. URL: https://www.freecodecamp.org/news/how-to-use-an-api-with-postman/ (дата звернення: 28.05.2024).

ДОДАТКИ

ДОДАТОК А

ПРОТОКОЛ ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ НА НАЯВНІСТЬ ТЕКСТОВИХ ЗАПОЗИЧЕНЬ

Назва роботи: WEB-застосунок для проходження опитувань, частина 1. Серверна частина

Тип роботи: бакалаврська дипломна робота
(БДР, МКР)

Підрозділ кафедра комп'ютерних наук, ФІТА
(кафедра, факультет)

Показники звіту подібності Unicheck

Оригінальність 96.05 % Схожість 3.95 %

Аналіз звіту подібності (відмітити потрібне):

- ✓ Запозичення, виявлення у роботі, оформлені коректно і не містять ознак плагіату.
- □ Виявлені у роботі запозичення не мають ознак плагіату, але їх надмірна кількість викликає сумніви щодо цінності роботи і відсутності самостійності її виконання автором. Роботу направити на розгляд експертної комісії кафедри.
- □ Виявлені у роботі запозичення є недобросовісними і мають ознаки плагіату та/або в ній містяться навмисні спотворення тексту, що вказують на спроби приховування недобросовісних запозичень.

Особа, відповідальна за перевірку Озеранський В.С.

Ознайомлені з повним звітом подібності, який був згенерований системою Unicheck щодо роботи.

Автор роботи Гарук В. В.

Керівник роботи Колодний В. В.

ДОДАТОК Б
ЛІСТИНГ ПРОГРАМИ

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { ExpressAdapter } from '@nestjs/platform-express';
import * as express from 'express';

async function bootstrap() {
  const server = express();
  const app = await NestFactory.create(AppModule, new ExpressAdapter(server));

  // Enable CORS for all origins
  app.enableCors();

  await app.listen(3300);
}
bootstrap();


import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';

import { ConfigService } from '@nestjs/config';
import { LoggingService } from './logger/logging.service';
import { DatabaseModule } from './database/database.module';
import { MyConfigModule } from './configuration/config.module';
import { GoogleStrategy } from './GAuth/google.strategy';
import { AuthService } from './GAuth/gauth.service';
import { AuthModule } from './GAuth/auth.module';
import { JwtAuthModule } from './GAuth/Jwt/jwt.module';
import { GauthControlloer } from './GAuth/gauth.controller';
import { UserModule } from './GAuth/User/user.module';
import { SurveyModule } from './survey/survey.module';

@Module({
  imports: [
    DatabaseModule,
    MyConfigModule,
    AuthModule,
    JwtAuthModule,
    UserModule,
    SurveyModule,
```

```
  ],
  controllers: [AppController, GauthControlloer],
  providers: [
    AppService,
    AuthService,
    ConfigService,
    LoggingService,
    GoogleStrategy,
  ],
})
export class AppModule {}

import {
  IsString,
  IsArray,
  IsNotEmpty,
  ValidateNested,
  ArrayMinSize,
} from 'class-validator';
import { Type } from 'class-transformer';

export class UpdateSurveyDto {
  @IsString()
  @IsNotEmpty()
  title: string;

  @IsString()
  @IsNotEmpty()
  description: string;
}

export class UpdateQuestionDto {
  @IsString()
  @IsNotEmpty()
  type: string;

  @IsString()
  @IsNotEmpty()
  question: string;

  @IsArray()
  @IsString({ each: true })
  emotions: string[];
}
class QuestionDto {
```

```
  @IsString()
  @IsNotEmpty()
  type: string;

  @IsString()
  @IsNotEmpty()
  answer: string;

  @IsArray()
  @ArrayMinSize(1)
  emotions: string[];
}

export class CreateSurveyDto {
  @IsArray()
  @ArrayMinSize(1)
  @ValidateNested({ each: true })
  @Type(() => QuestionDto)
  questions: QuestionDto[];

  @IsString()
  @IsNotEmpty()
  hash: string;

  @IsString()
  @IsNotEmpty()
  title: string;

  @IsString()
  @IsNotEmpty()
  description: string;
}

import {
  Controller,
  Post,
  Body,
  Get,
  UseGuards,
  Req,
  NotFoundException,
  Param,
  Put,
} from '@nestjs/common';
import { SurveyService } from './survey.service';
```

```
import { JwtAuthGuard } from 'src/GAuth/Jwt/jwt-auth.guard';
import { Response } from 'src/database/response.model';
import {
  CreateSurveyDto,
  UpdateQuestionDto,
  UpdateSurveyDto,
} from './dtoSurvey';
import { v4 as uuidv4 } from 'uuid';

@Controller('survey')
export class SurveyController {
  constructor(private readonly surveyService: SurveyService) {}

  @Post('new')
  @UseGuards(JwtAuthGuard)
  saveNewSurvey(@Body() surveyData: CreateSurveyDto, @Req() req) {
    const userId = req.user.userId;
    return this.surveyService.saveSurvey(surveyData, userId);
  }
  @Get('created')
  @UseGuards(JwtAuthGuard)
  async myCreatedSurveys(@Req() req) {
    const { userId } = req.user;
    return await this.surveyService.getCreatedSurveys(userId);
  }

  @Get('taken')
  @UseGuards(JwtAuthGuard)
  async myTakenSurveys(@Req() req) {
    const { userId } = req.user;
    return await this.surveyService.getTakenSurveys(userId);
  }
  // @UseGuards(JwtAuthGuard)
  @Get('take/:surveyHash')
  async takeSurvey(@Req() req) {
    //validdata hash if it is not sql inection
    return await this.surveyService.getSurvey(req.params.surveyHash);
  }
  @Post('submit')
  @UseGuards(JwtAuthGuard)
  async submitSurvey(@Body() surveyData: any, @Req() req) {
    // Validate hash to prevent SQL injection
    console.log('surveyData', surveyData);
    console.log('req.user', req.user);
    const hash = surveyData.surveyHash;
```

```
  const { userId } = req.user;
  const sessionId = uuidv4();
  try {

    for (const questionId in surveyData.emotions) {
      if (surveyData.emotions.hasOwnProperty(questionId)) {
        const emotions = surveyData.emotions[questionId];

        for (const emotion in emotions) {
          if (emotions.hasOwnProperty(emotion)) {
            const value = parseInt(emotions[emotion], 10);

            await Response.create({
              question_id: parseInt(questionId, 10),
              question_hash: hash,
              user_id: userId,
              session_id: sessionId,
              value,
            });
          }
        }
      }
    }

    return {
      success: true,
      message: 'Survey responses stored successfully.',
    };
  } catch (error) {
    console.error('Error submitting survey:', error);
    return { success: false, message: 'Failed to store survey responses.' };
  }
}
@Get(':id/results')
@UseGuards(JwtAuthGuard)
async getSurveyResults(@Param('id') id: number) {
  return this.surveyService.getSurveyResults(id);
}

@Get(':id')
@UseGuards(JwtAuthGuard)
async getSurveyById(@Param('id') id: number) {
  const survey = await this.surveyService.getSurveyById(id);
  if (!survey) {
    throw new NotFoundException('Survey not found');
```

```
    }
    return survey;
  }

  @Put(':id')
  @UseGuards(JwtAuthGuard)
  async updateSurvey(
    @Param('id') id: number,
    @Body() updateSurveyDto: UpdateSurveyDto,
  ) {
    const updatedSurvey = await this.surveyService.updateSurvey(
      id,
      updateSurveyDto,
    );
    if (!updatedSurvey) {
      throw new NotFoundException('Survey not found');
    }
    return updatedSurvey;
  }

  @Put(':surveyId/questions/:questionId')
  @UseGuards(JwtAuthGuard)
  async updateQuestion(
    @Param('surveyId') surveyId: number,
    @Param('questionId') questionId: number,
    @Body() updateQuestionDto: UpdateQuestionDto,
  ) {
    const updatedQuestion = await this.surveyService.updateQuestion(
      surveyId,
      questionId,
      updateQuestionDto,
    );
    if (!updatedQuestion) {
      throw new NotFoundException('Question not found');
    }
    return updatedQuestion;
  }

  @Get(':id/results-all')
  @UseGuards(JwtAuthGuard)
  async getSurveyResultsAll() {
    return this.surveyService.getAllSurveyResults();
  }
}
```

```
import { Module } from '@nestjs/common';
import { SurveyService } from './survey.service';
import { SurveyController } from './survey.controller';

@Module({
  providers: [SurveyService],
  controllers: [SurveyController],
})
export class SurveyModule {}

import {
  BadRequestException,
  Injectable,
  NotFoundException,
} from '@nestjs/common';
import { Question } from 'src/database/question.model';
import { Survey } from 'src/database/survey.model';
import { Response } from 'src/database/response.model'; // Import the Response model
import { UpdateQuestionDto, UpdateSurveyDto } from './dtoSurvey';
import { User } from 'src/database/user.model';

@Injectable()
export class SurveyService {
  constructor() {}

  async saveSurvey(surveyData: any, userId: number) {
    // Create a new survey instance
    if (!surveyData.title) {
      throw new BadRequestException('Missing required parameter: title');
    }

    const newSurvey = new Survey();
    newSurvey.creator_id = userId;
    newSurvey.title = surveyData.title;
    newSurvey.description = surveyData.description;
    newSurvey.created_at = new Date();
    newSurvey.hash = surveyData.hash;

    // Save the survey to the database
    await newSurvey.save();

    // Save questions associated with the survey
    if (surveyData.questions && surveyData.questions.length > 0) {
      await Promise.all(
        surveyData.questions.map(async (questionData: any) => {
```

```
        const newQuestion = new Question();
        newQuestion.survey_id = newSurvey.survey_id;
        newQuestion.type = questionData.type;
        newQuestion.question = questionData.answer;
        newQuestion.emotions = questionData.emotions;
        console.log(newQuestion);
        await newQuestion.save();
      }),
    );
  }
  console.log('Survey saved successfully', newSurvey);

  return newSurvey;
}

async getSurvey(hash: string) {
  // Find the survey based on the provided hash
  const survey = await Survey.findOne({
    where: { hash },
    include: Question,
  });
  console.log('Survey found:', survey);
  return survey;
}
async getCreatedSurveys(userId: number) {
  // Find all surveys created by the user
  const surveys = await Survey.findAll({
    where: { creator_id: userId },
    include: Question,
  });
  console.log('Surveys found:', surveys);
  return surveys;
}

async getTakenSurveys(userId: number) {
  try {
    const responses = await Response.findAll({
      where: { user_id: userId },
      include: {
        model: Question,
        include: [Survey],
      },
    });
```

```
  const questionsMap = new Map();

  responses.forEach((response) => {
    const questionId = response.question_id;

    if (!questionsMap.has(questionId)) {
      questionsMap.set(questionId, {
        question: {
          question_id: response.question.question_id,
          survey_id: response.question.survey_id,
          type: response.question.type,
          question: response.question.question,
          emotions: response.question.emotions,
          createdAt: response.question.createdAt,
          updatedAt: response.question.updatedAt,
          survey: {
            survey_id: response.question.survey.survey_id,
            creator_id: response.question.survey.creator_id,
            title: response.question.survey.title,
            description: response.question.survey.description,
            created_at: response.question.survey.created_at,
            hash: response.question.survey.hash,
            createdAt: response.question.survey.createdAt,
            updatedAt: response.question.survey.updatedAt,
          },
        },
        responses: [],
      });
    }

    questionsMap.get(questionId).responses.push({
      response_id: response.response_id,
      question_hash: response.question_hash,
      question_id: response.question_id,
      user_id: response.user_id,
      value: response.value,
      createdAt: response.createdAt,
      updatedAt: response.updatedAt,
    });
  });
  const result = Array.from(questionsMap.values());

  console.log('Taken surveys:', result);
  return result;
} catch (error) {
```

```
    console.error('Error fetching taken surveys:', error);
    throw error;
  }
}

async getSurveyById(surveyId: number) {
  // Find the survey based on the provided surveyId
  const survey = await Survey.findByPk(surveyId, {
    include: Question,
  });
  console.log('Survey found:', survey);
  return survey;
}

async updateSurvey(
  id: number,
  updateSurveyDto: UpdateSurveyDto,
): Promise<Survey> {
  const survey = await this.getSurveyById(id);
  if (!survey) {
    throw new NotFoundException('Survey not found');
  }
  survey.title = updateSurveyDto.title;
  survey.description = updateSurveyDto.description;
  return survey.save();
}

async updateQuestion(
  surveyId: number,
  questionId: number,
  updateQuestionDto: UpdateQuestionDto,
): Promise<Question> {
  const question = await Question.findOne({
    where: { question_id: questionId, survey_id: surveyId },
  });
  if (!question) {
    throw new NotFoundException('Question not found');
  }
  question.type = updateQuestionDto.type;
  question.question = updateQuestionDto.question;
  question.emotions = updateQuestionDto.emotions;
  return question.save();
}

async getSurveyResults(surveyId: number) {
```

```
const survey = await Survey.findByPk(surveyId, {
  include: [
    {
      model: Question,
      include: [
        {
          model: Response,
          include: [User],
        },
      ],
    },
  ],
});

if (!survey) {
  throw new NotFoundException('Survey not found');
}

const surveyInfo = {
  id: survey.survey_id,
  title: survey.title,
  description: survey.description,
  createdAt: survey.createdAt,
  updatedAt: survey.updatedAt,
};

const questionsInfo = survey.questions.map((question: Question) => ({
  questionId: question.question_id,
  questionText: question.question,
  questionType: question.type,
  emotions: question.emotions,
}));

const responsesBySession = survey.questions.reduce(
  (acc, question: Question) => {
    question.responses.forEach((response: Response) => {
      if (!acc[response.session_id]) {
        acc[response.session_id] = [];
      }
      acc[response.session_id].push({
        questionId: question.question_id,
        questionText: question.question,
        value: response.value,
        createdAt: response.createdAt,
        responseId: response.response_id,
```

```
          sessionId: response.session_id,
          user: {
            id: response.user.user_id,
            name: response.user.name,
          },
        });
      });
      return acc;
    },
    {},
  );

  const groupedResponses = Object.keys(responsesBySession).map(
    (sessionId) => ({
      sessionId,
      responses: responsesBySession[sessionId],
    }),
  );

  return {
    survey: surveyInfo,
    questions: questionsInfo,
    responses: groupedResponses,
  };
}

async getAllSurveyResults() {
  const surveys = await Survey.findAll({
    include: [
      {
        model: Question,
        include: [
          {
            model: Response,
            include: [User],
          },
        ],
      },
    ],
  });

  const allResults = surveys.map((survey) => {
    return {
      survey: {
        id: survey.survey_id,
```

```
        title: survey.title,
        description: survey.description,
        createdAt: survey.createdAt,
        updatedAt: survey.updatedAt,
      },
      questions: survey.questions.map((question: Question) => ({
        questionId: question.question_id,
        questionText: question.question,
        questionType: question.type,
        emotions: question.emotions,
      })),
      responses: survey.questions.reduce((acc, question: Question) => {
        question.responses.forEach((response: Response) => {
          if (!acc[response.session_id]) {
            acc[response.session_id] = [];
          }
          acc[response.session_id].push({
            questionId: question.question_id,
            value: response.value,
            createdAt: response.createdAt,
            responseId: response.response_id,
            sessionId: response.session_id,
            user: {
              id: response.user.user_id,
              name: response.user.name,
            },
          });
        });
        return acc;
      }, {}),
    };
  });

  allResults.forEach((result) => {
    Object.keys(result.responses).forEach((sessionId) => {
      result.responses[sessionId].sort((a, b) => a.responseId - b.responseId);
    });
  });

  return allResults;
 }
}

// auth.module.ts
import { Module } from '@nestjs/common';
```

```
import { AuthService } from './gauth.service';
import { GoogleStrategy } from './google.strategy';
import { ConfigService } from '@nestjs/config';
import { DatabaseModule } from 'src/database/database.module';
import { JwtAuthModule } from './Jwt/jwt.module';
import { GauthControlloer } from './gauth.controller';

@Module({
  imports: [DatabaseModule, JwtAuthModule],
  providers: [AuthService, GoogleStrategy, ConfigService, GauthControlloer],
  exports: [AuthService, GauthControlloer],
})
export class AuthModule {}


/* eslint-disable @typescript-eslint/no-unused-vars */
import {
  Controller,
  Get,
  UseGuards,
  Req,
  Res,
  Post,
  Body,
} from '@nestjs/common';
import { Response } from 'express';
import { AuthGuard } from '@nestjs/passport';
import { AuthService } from './gauth.service';
import { User } from 'src/database/user.model';

@Controller('auth')
export class GauthControlloer {
  constructor(private readonly auth: AuthService) {}

  @Get('google')
  @UseGuards(AuthGuard('google'))
  // eslint-disable-next-line @typescript-eslint/no-empty-function
  async googleAuth(@Req() req) {}

  @Get('google/callback')
  @UseGuards(AuthGuard('google'))
  async googleAuthRedirect(@Req() req, @Res() res: Response) {
    const jwt = await this.auth.googleLogin(req);
    console.log('jwt ', jwt);
    res.redirect(`http://localhost:3000/storeToken?token=${jwt.access_token}`);
```

```
  }

  @Post('signup')
  async signup(@Body() body) {
    const existingUser = await User.findOne({
      where: { email: body.email },
    });
    if (existingUser) {
      return { success: false, code: 409 };
    }
    return await this.auth.signup(body);
  }
  @Post('login')
  async login(@Body() body) {
    return await this.auth.login(body);
  }
}

import { Injectable } from '@nestjs/common';
import { DatabaseService } from 'src/database/database.service';
import { JwtService } from '@nestjs/jwt';
import { User } from 'src/database/user.model';

@Injectable()
export class AuthService {
  constructor(
    private readonly database: DatabaseService,
    private jwtService: JwtService,
  ) {}

  async googleLogin(req) {
    const { user } = req;
    if (!user) {
      throw new Error('No user from Google');
    }

    const dbUser = await this.database.findOrCreateUser(user);

    const payload = { userEmail: dbUser.email, userId: dbUser.user_id };
    const access_token = this.jwtService.sign(payload);

    return {
      access_token: access_token,
    };
  }
```

```
  async signup(body) {
    const user = await User.create({
      name: body.name,
      email: body.email,
      picture: body.picture,
      password: body.password,
      last_login: new Date(),
    });
    const userData = user.dataValues;
    const payload = { userEmail: userData.email, userId: userData.user_id };
    const access_token = this.createJwt(payload);
    return {
      success: true,
      code: 400,
      accessToken: access_token,
    };
  }
  async login(body) {
    const existingUser = await User.findOne({
      where: { email: body.email },
    });
    const dataValues = existingUser.dataValues;
    if (existingUser && dataValues.password == body.password) {
      const payload = {
        userEmail: dataValues.email,
        userId: dataValues.user_id,
      };
      const access_token = this.createJwt(payload);

      return { success: true, code: 200, accessToken: access_token };
    }
    return { success: false, code: 404 };
  }

  createJwt(payload) {
    return this.jwtService.sign(payload);
  }
}

import { PassportStrategy } from '@nestjs/passport';
import { Strategy, VerifyCallback } from 'passport-google-oauth20';
import { config } from 'dotenv';

import { Injectable } from '@nestjs/common';
import { ConfigService } from '@nestjs/config';
```

```
config();

@Injectable()
export class GoogleStrategy extends PassportStrategy(Strategy, 'google') {
  constructor(private readonly configService: ConfigService) {
    super({
      clientID: configService.get<string>('googleAuthclientID'),
      clientSecret: configService.get<string>('googleAuthclientSecret'),
      callbackURL: 'http://localhost:3300/auth/google/callback',
      scope: ['email', 'profile'],
    });
  }
  async validate(
    accessToken: string,
    refreshToken: string,
    profile: any,
    done: VerifyCallback,
  ): Promise<any> {
    const { displayName, emails, photos } = profile;
    const user = {
      email: emails[0].value,
      name: displayName,
      picture: photos[0].value,
      password: 'google',
      accessToken,
      refreshToken,
    };
    done(null, user);
  }
}

import { Injectable } from '@nestjs/common';
import { AuthGuard } from '@nestjs/passport';

@Injectable()
export class JwtAuthGuard extends AuthGuard('jwt') {
}

import { Module } from '@nestjs/common';
import { JwtModule } from '@nestjs/jwt';
import { ConfigModule, ConfigService } from '@nestjs/config';
import { JwtStrategy } from './jwt.strategy';
import { JwtAuthGuard } from './jwt-auth.guard';
```

```
@Module({
  imports: [
    JwtModule.registerAsync({
      imports: [ConfigModule],
      inject: [ConfigService],
      useFactory: (configService: ConfigService) => {
        const jwtSecret = configService.get<string>('jwtSecret');
        if (!jwtSecret) {
          throw new Error('JWT secret key is not defined in configuration.');
        } else {
          console.log('JWT secret key is defined in configuration.');
        }

        return {
          secret: jwtSecret,
          signOptions: { expiresIn: '24h' },
        };
      },
    }),
  ],
  providers: [JwtAuthGuard, JwtStrategy, ConfigService],
  exports: [JwtAuthGuard, JwtStrategy, JwtModule],
})
export class JwtAuthModule {}

import { Injectable } from '@nestjs/common';
import { ConfigService } from '@nestjs/config';
import { PassportStrategy } from '@nestjs/passport';
import { ExtractJwt, Strategy } from 'passport-jwt';

@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy) {
  constructor(private readonly configService: ConfigService) {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      secretOrKey: configService.get<string>('jwtSecret'),
      ignoreExpiration: false,
    });
  }

  async validate(payload: any) {
    return { userEmail: payload.userEmail, userId: payload.userId };
  }
}
```

ДОДАТОК В

ГРАФІЧНА ЧАСТИНА

«WEB-ЗАСТОСУНКОК ДЛЯ ПРОХОДЖЕННЯ ОПИТУВАНЬ. ЧАСТИНА 1. СЕРВЕРНА ЧАСТИНА»

Виконав: студент 4 курсу, групи 3КН-20б
спеціальності 122 – Комп'ютерні науки
(шифр і назва напряму підготовки, спеціальності)
Гарук В. В.
(прізвище та ініціали)

Керівник: к.т.н., доцент каф. КН
Колодний В. В.
(прізвище та ініціали)
«07» 06 2024 р

Вінниця ВНТУ – 2024 рік

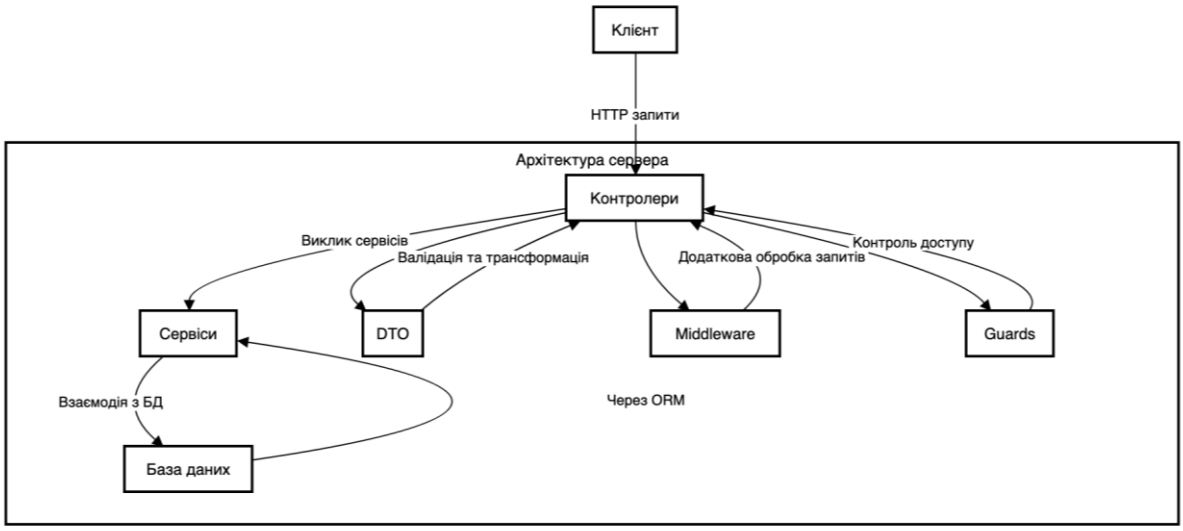


Рисунок В.1 - Загальна структурна схема архітектури серверної частини сервіса

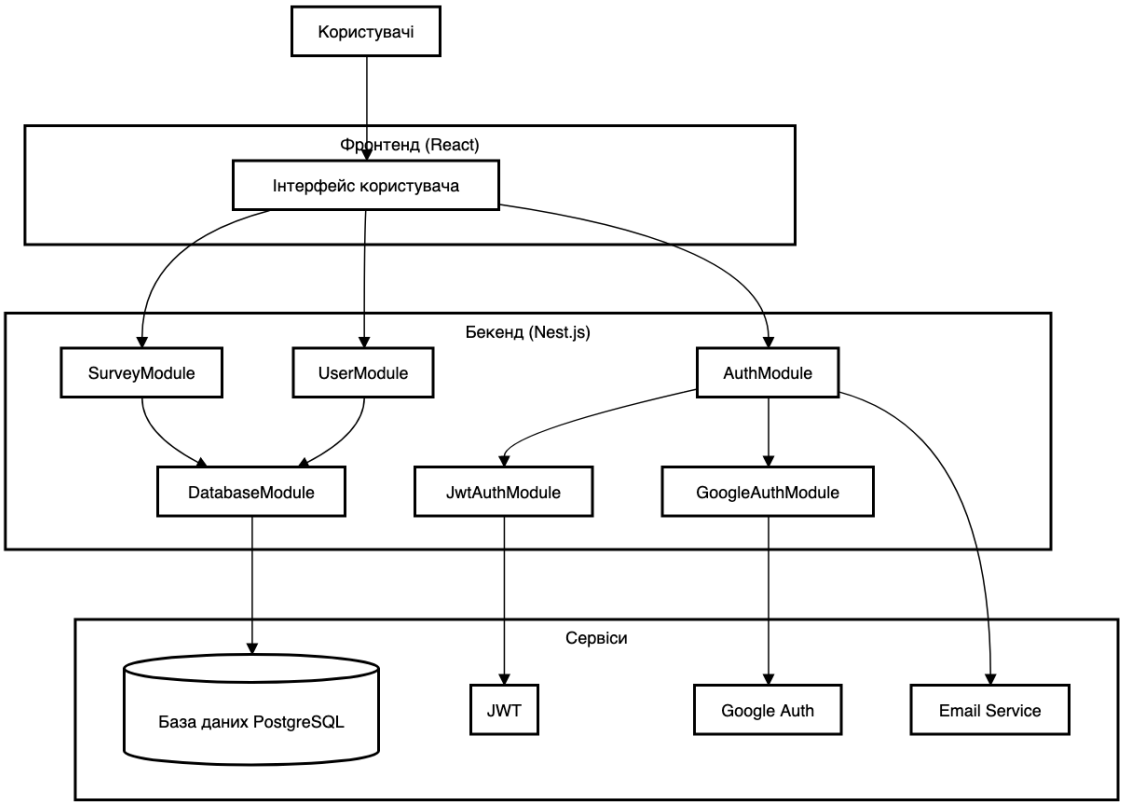


Рисунок В.2 – Схема функціонування серверної частини

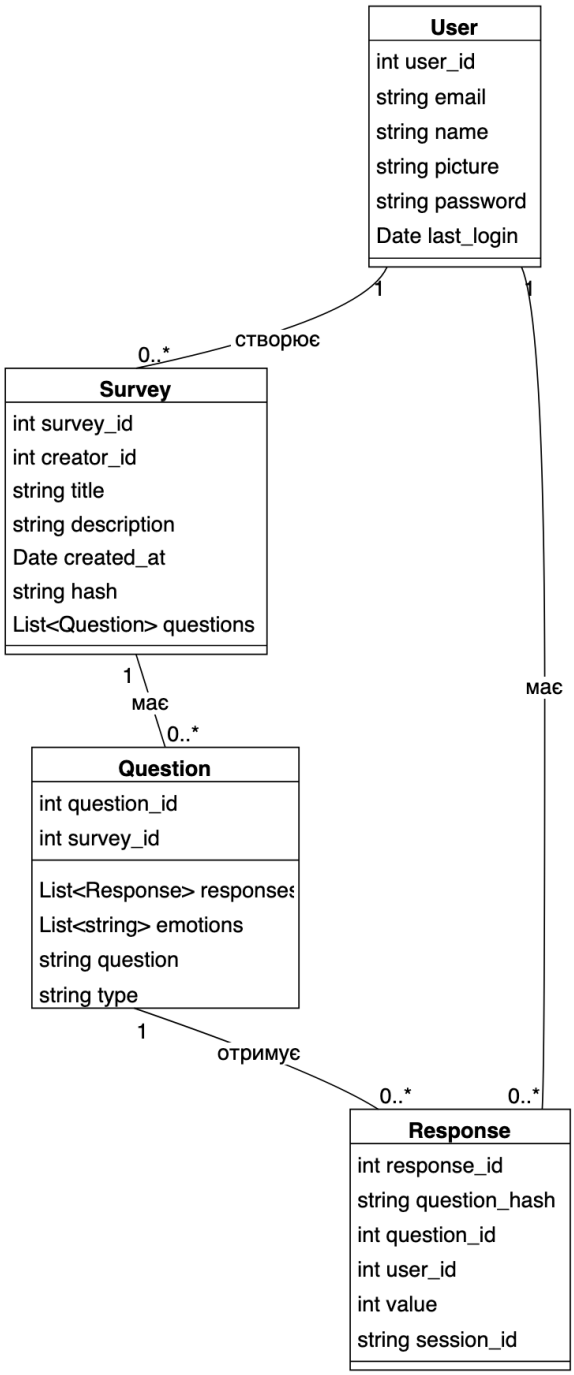


Рисунок В.3 – Загальна UML-діаграма класів

	* survey_id integer	* creator_id integer	* title varchar(255)	* description text	* created_at timestamp with time zo	* hash text
> 1	21	1	survey1	no description	2024-05-17 15:36:01.38	9UCn2aKA

Рисунок В.4 – Фрагмент реляційної моделі бази даних

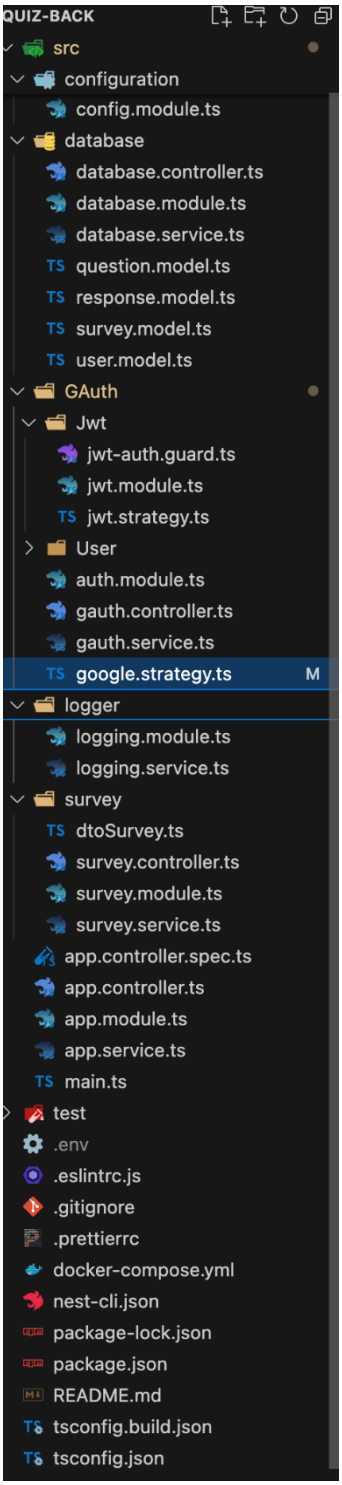


Рисунок В.5 – Структура проекту у Visual Studio Code

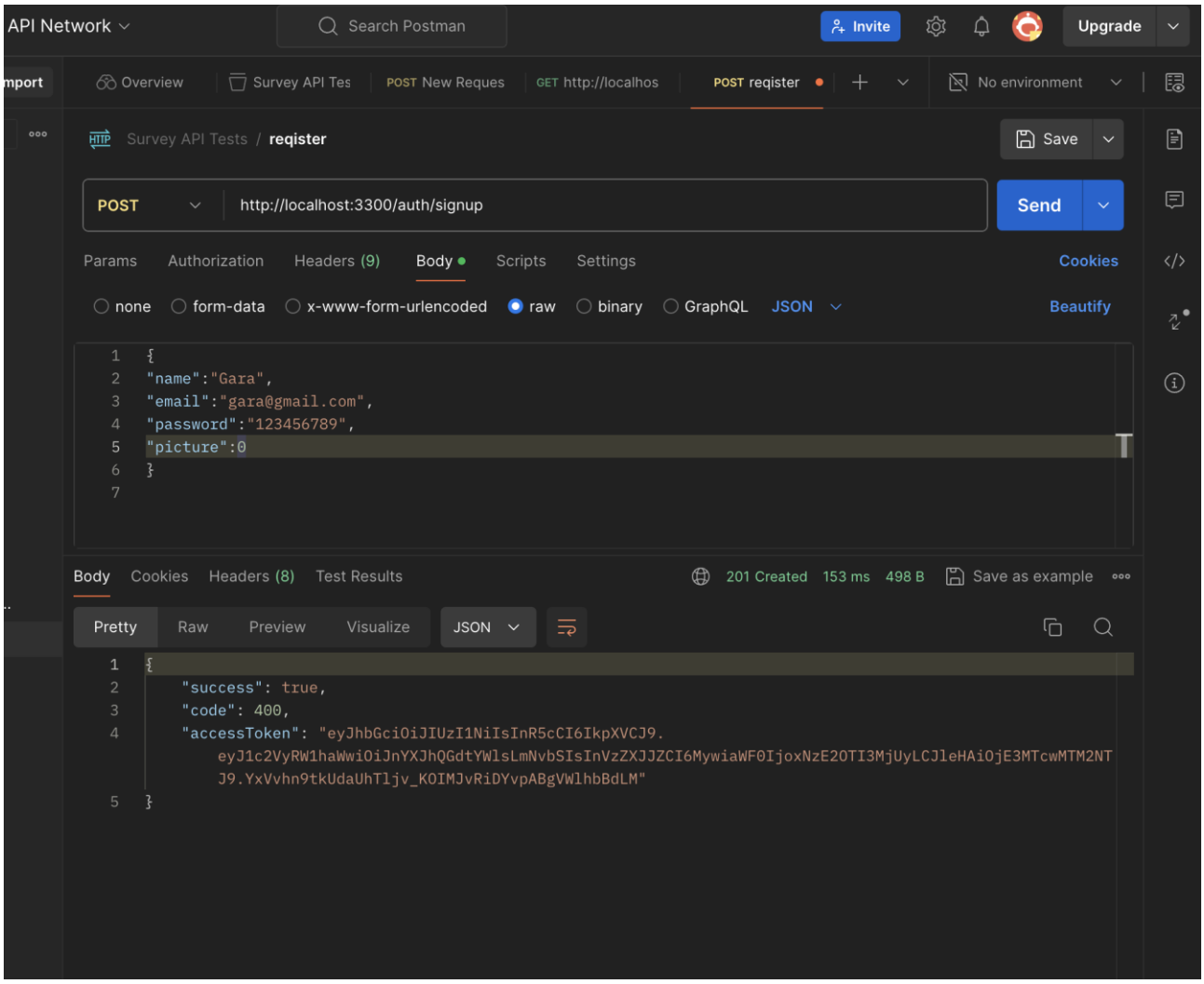


Рисунок В.6 – Результат API запиту GET /auth/signup

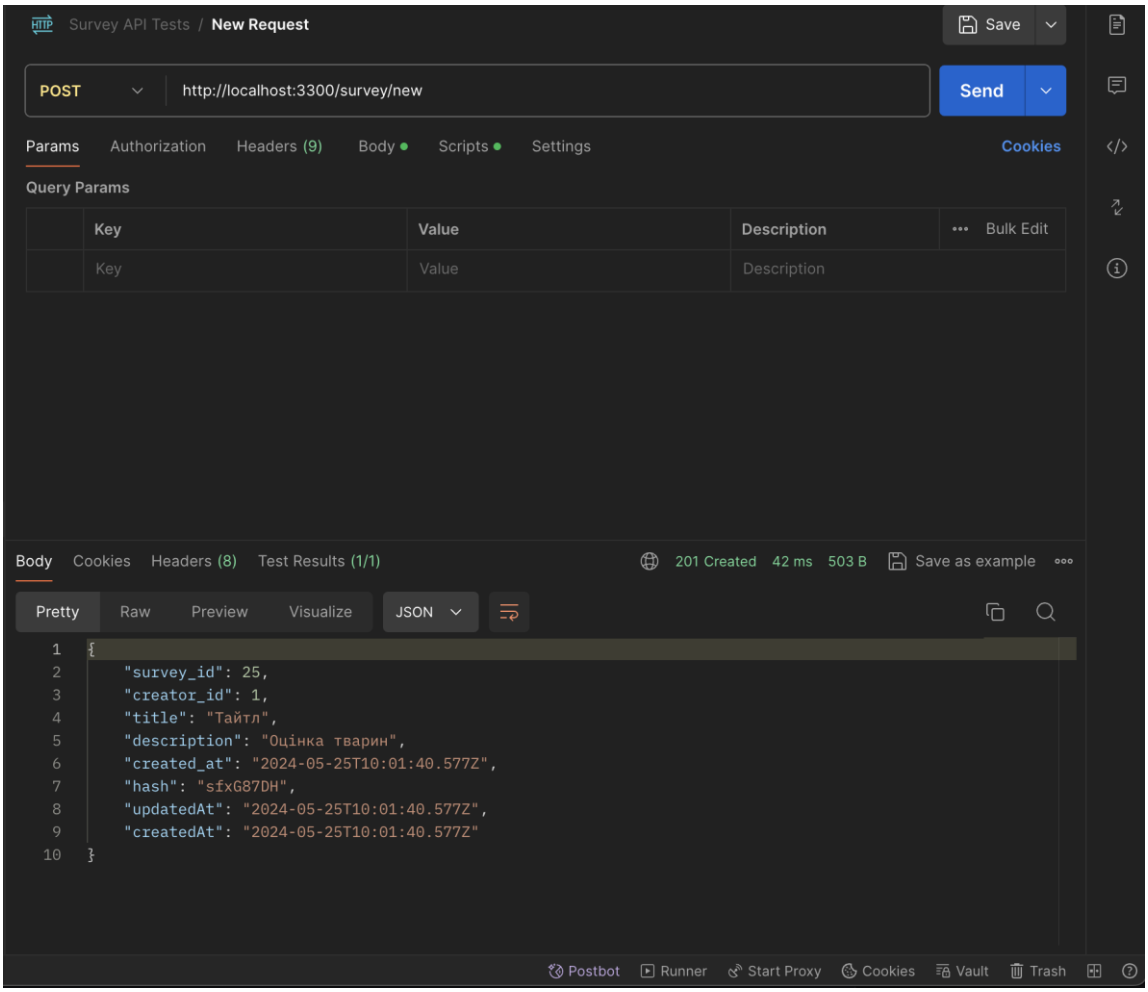


Рисунок В.7 – Результат API запиту GET /survey/created

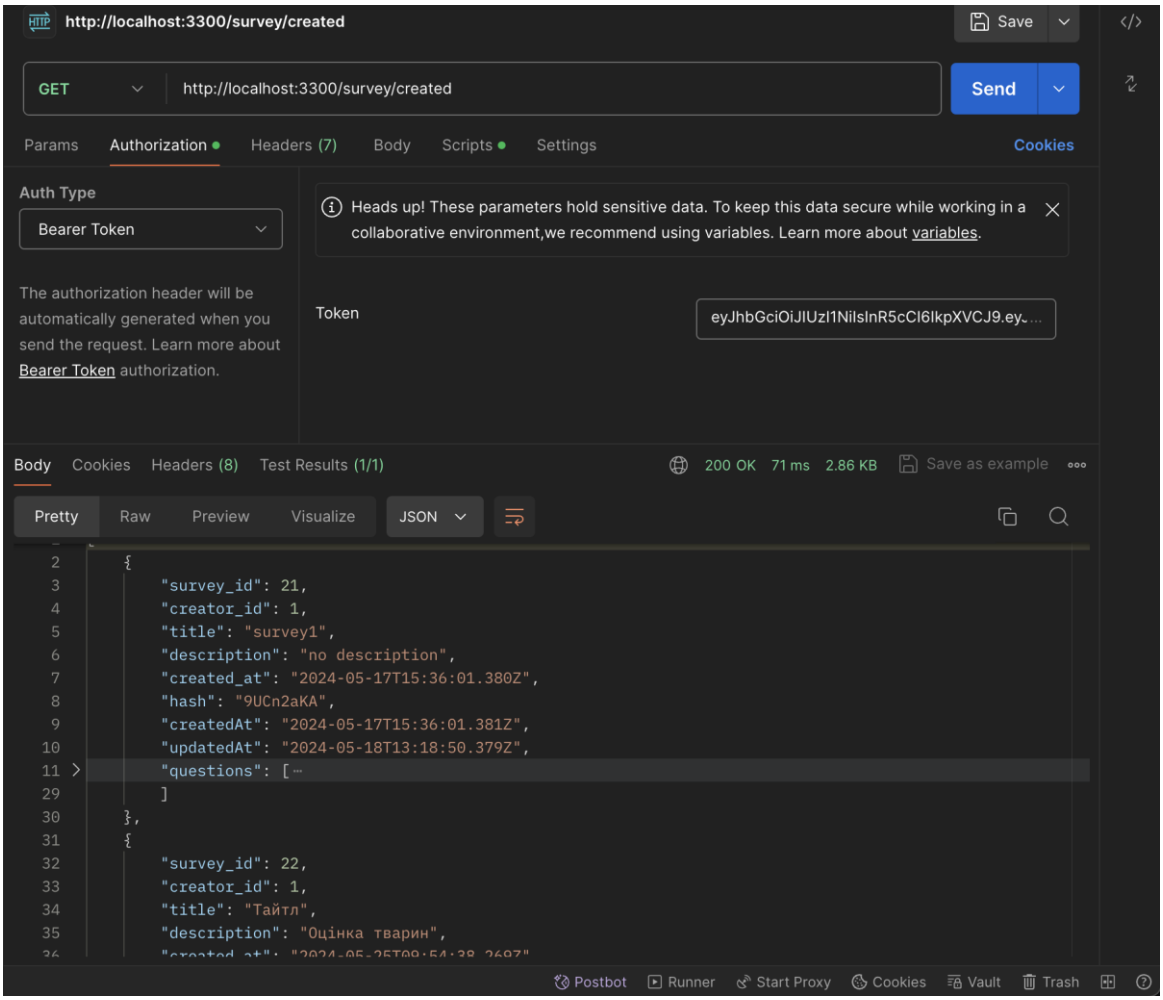


Рисунок В.8 – Результат API запиту GET /survey/created

ДОДАТОК Г
ІНСТРУКЦІЯ КОРИСТУВАЧА

Для того щоб використати серверну частину, потрібно звернутися до розділу 3.3 Розробка API документації. Для створення JWT токена можна використати запит GET /auth/signup (рис. В.6) чи GET /auth/login. Зберегти токен на клієнтській частни. Після цього ви зможете виконувати маршрутизацію будь яких запитів які описані в документації.