

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук і технологій

(повне найменування інституту, назва факультету)

Кафедра програмних засобів

(повне найменування кафедри)

Пояснювальна записка

до дипломного проекту (роботи)

магістр

(ступінь вищої освіти)

на тему ПРОЕКТУВАННЯ ТА РОЗРОБЛЕННЯ АРХІТЕКТУРИ СИСТЕМИ
ЗБЕРІГАННЯ ТА ОБРОБЛЕННЯ ДАНИХ ДЛЯ ОНЛАЙН ШКІЛ НА ОСНОВІ
МІКРОСЕРВІСІВ

DESIGN AND DEVELOPMENT OF THE MICROSERVICE-BASED
ARCHITECTURE FOR DATA PROCESSING AND STORAGE SYSTEM AT
ONLINE SCHOOLS

Виконав: студент 2 курсу, групи КНТ-121м
Спеціальності 121 Інженерія програмного
забезпечення

(код і найменування спеціальності)

Освітня програма (спеціалізація)

Інженерія програмного забезпечення

Соколовський Д.В.

(прізвище та ініціали)

Керівник Табунщик Г.В.

(прізвище та ініціали)

Рецензент Шитікова О.В.

(прізвище та ініціали)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»
 (повне найменування закладу вищої освіти)

Інститут, факультет ФКНТ
 Кафедра програмних засобів
 Ступінь вищої освіти магістр
 Спеціальність 121 Інженерія програмного забезпечення
(код і найменування)
 Освітня програма (спеціалізація) Інженерія програмного забезпечення
(назва освітньої програми (спеціалізації))

ЗАТВЕРДЖУЮ

Завідувач кафедри ПЗ, д.т.н, проф.
С.О. Субботін
 “ ” 20 року

З А В Д А Н Н Я

НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА

Соколовського Даніла Віталійовича

(прізвище, ім'я, по батькові)

1. Тема проєкту (роботи) Проектування та розроблення архітектури системи зберігання та оброблення даних для онлайн шкіл на основі мікросервісів.
Design and Development of the Microservice-Based Architecture for Data Processing and Storage System at Online Schools
 керівник проєкту (роботи) Табунщик Галина Володимирівна, професор,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
 затверджені наказом закладу вищої освіти від “04” листопада 2022 року № 363
2. Строк подання студентом проєкту (роботи) 8 грудня 2022 року
3. Вихідні дані до проєкту (роботи) технічне завдання

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1. Аналіз проблеми та постановка завдань дослідження.
2. Матеріали і методи. 3. Розробка архітектури програми. 4. Розробка програми. 5.
Експлуатація, тестування та експериментальне дослідження програми.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень) Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-5 Основна частина	Табунщик Г.В., професор		
Нормоконтролер	Камінська Ж.К., асистент		

7. Дата видачі завдання “01” вересня 2022 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз предметної області, розробка та затвердження технічного завдання	1 тиждень	Технічне завдання
2	Проектування структури системи	2 тиждень	Структурна схема
3	Розробка схеми функціонування системи та модулів програми	3 тиждень	Функціональна схема. Модулі
4	Створення програмного коду системи	4-5 тиждень	Текст програми
5	Тестування та відналагодження програми	6 тиждень	Працездатна система
6	Розробка розділів пояснювальної записки	7 тиждень	Пояснювальна записка
7	Розробка графічного матеріалу та захист дипломної роботи	8 тиждень	Графічний матеріал та дипломна робота

Студент

(підпис) Соколовський Д.В.
(прізвище та ініціали)

Керівник проєкту (роботи)

(підпис) Табунщик Г.В.
(прізвище та ініціали)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи бакалавра:
113с., 18 табл., 13 рис., 2 дод., 14 джерел.

ВЕБЗАСТОСУНОК, ОНЛАЙН НАВЧАННЯ, API, NODE, TYPESCRIPT,
JEST, DOCKER, EXPRESS, WEBSTORM.

Об'єкт дослідження – сервіси для онлайн навчання.

Предмет дослідження – програмні засоби реалізації сервісів для підтримки клієнтських застосувань для онлайн навчання.

Мета роботи – розробка сервісів для підтримки клієнтських застосувань для онлайн шкіл.

Матеріали, методи та технічні засоби: вебпрограмування, платформа Node.js, фреймворк Express, база даних PostgreSQL, середа тестування Jest, персональний комп'ютер з процесором Apple M1 під управлінням операційної системи macOS 12.6.

Результати. Створено API для підтримки клієнтських застосувань для онлайн навчання.

Висновки. Розроблено сервіси для клієнтських застосувань, які оптимізують процес навчання та викладання в онлайн школах.

Галузь використання – клієнтські застосування, розроблені для онлайн навчання та викладання.

ABSTRACT

Explanatory note to the diploma qualifying work of the master: 113 pages, 13 figures, 18 tables, 2 appendixes, 14 sources.

WEB APPLICATION, ONLINE LEARNING, API, NODE, TYPESCRIPT, JEST, DOCKER, EXPRESS, WEBSTORM.

Object of study: services for online learning.

Subject of study: software tools for the implementation of services to support customer applications for online learning.

The purpose of the work is to develop the services for supporting client applications for online learning.

Materials, methods, and tools: web programming, Node.js platform, Express framework, PostgreSQL database, Jest testing environment, personal computer with Apple M1 processor running macOS 12.6.

Results. API to support client applications for online learning was created.

Conclusions. Services for client applications that optimize the process of learning and teaching at online schools were created.

The field of use: customer applications designed for online learning and teaching.

ЗМІСТ

Перелік скорочень та умовних позначок	8
Вступ.....	9
1 Аналіз проблеми та постановка завдань дослідження	11
1.1 Аналіз предметної області	11
1.2 Огляд існуючих методів вирішення завдання	13
1.3 Технічне завдання	16
1.4 Висновки за розділом 1.....	16
2 Матеріали і методи.....	16
2.1 Дослідження підходів до тестування програмного забезпечення.....	16
2.2 Вибір між монолітним підходом та мікросервісною архітектурою.....	26
2.3 Тестування навантаження.....	30
2.4 Висновки за розділом 2.....	35
3 Опис програми.....	36
3.1 Вибір інструментарію розробки програмного забезпечення.....	36
3.2 Висновки за розділом 3.....	44
4 Розробка програми	45
4.1 Опис структури файлів	45
4.2 Основні рішення з реалізації компонентів системи	46
4.3 Захищення даних	52
4.4 Основні розроблені алгоритми	52
4.5 Опис допоміжних бібліотек, модулів та функцій	53
4.6 Висновки за розділом 4.....	55
5 Експлуатація, тестування та експериментальне дослідження програми	56
5.1 Призначення й умови застосування програми.....	56
5.2 Виняткові ситуації.....	57
5.3 Тестування та експериментальне дослідження програми	59
5.4 Висновки за розділом 5.....	61

Висновки	62
Перелік посилань.....	63
Додаток А Текст програми.....	65
Додаток Б Слайди презентації	104

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

API – Application Programming Interface;

BDD – Behavior Driven Development;

IT – Information Technology;

HTTP – HyperText Transfer Protocol;

MSA – Micro Service Architecture;

ORM – Object-Relational Mapping;

TDD – Test-Driven Development;

TLD – Test Last Development;

ПЗ – програмне забезпечення;

ПП – програмний продукт;

СУБД – система управління базами даних.

ВСТУП

У нашому столітті онлайн навчання стало дуже популярними, бо воно, безсумнівно, має багато переваг, можна навчатись не виходячи з дому або під час подорожі. Протягом останніх років онлайн навчання набувало популярності, але після подій у 2020 році, коли люди могли тижнями не контактувати один з одним, у всьому світі онлайн навчання стало порятунком, єдиним способом продовжувати розвиток, особливо для учнів та студентів, які тільки за допомогою онлайн навчання могли продовжувати взаємодіти з викладачами та не відставати від навчальної програми.

Враховуючи попит на онлайн навчання, школам та іншим навчальним закладам важливо використовувати сучасні інструменти для забезпечення якісного викладання онлайн. Деякі навчальні курси/школи використовують вже існуючі вебзастосунки, а інші віддають перевагу розробці власних застосунків, щоб мати в програмі весь необхідний функціонал та легко додавати те, чого не вистачає.

Звичайно, розробники вебзастосунків приділяють особливу увагу і клієнтській частині, і серверній. Користувачі сайтів бачать лише інтерфейс, але швидкість роботи форм, швидкість отримання та обробки даних, гарантія цілісності даних, все це залежить від якості розробленої бекенд частини. Так як клієнтська частина залежить від серверної, дуже важливо, щоб вебзастосунок мав якісно розроблені сервіси.

Саме тому треба розробити сервіси для підтримки клієнтських застосувань для онлайн навчання та викладання, додати можливість роботи з базою даних, використати для цього сучасні інструменти та написати такий код, який можна буде легко масштабувати.

Ця задача буде вирішуватися за допомогою аналізу предметної області та використанню сучасних інструментів, які допоможуть вирішити проблему. Пропонується використати мікросервісну архітектуру, Micro Service Architecture (MSA). Мікросервіси – це шаблон сервіс-орієнтованої архітектури, в якому

програми створюються як сукупність найменших різних незалежних сервісних одиниць. Це програмний підхід, який фокусується на розкладанні програми на однофункціональні модулі із чітко визначеними інтерфейсами. Ці модулі можуть бути незалежно розгорнуті та експлуатуватися невеликими групами, що володіють усім життєвим циклом служби. Мікросервісна архітектура дозволяє створювати програми у вигляді набору невеликих автономних сервісів, розроблених для бізнес-сфери [1].

У мікросервісній архітектурі додаток розбивається на ряд сервісів, що незалежно розгортаються та які взаємодіють за допомогою API-інтерфейсів. Завдяки такому підходу кожен окремий сервіс можна розгортати та масштабувати незалежно від інших. Завдяки цьому команди можуть частіше і швидше постачати об'ємні та складні програми. На відміну від монолітної програми, з мікросервісною архітектурою команди можуть швидше впроваджувати нові можливості та вносити зміни, при цьому їм не доводиться переписувати великі фрагменти існуючого коду [2].

Отже, з мікросервісною архітектурою можна полегшити масштабування, прискорити розробку та скоротити ітеративний цикл розробки сервісів.

1 АНАЛІЗ ПРОБЛЕМИ ТА ПОСТАНОВКА ЗАВДАНЬ ДОСЛІДЖЕННЯ

1.1 Аналіз предметної області

За останні роки кількість користувачів платформ для онлайн навчання значно зросла. В інтернеті можна знайти багато інструментів, щоб зробити процес викладання та навчання максимально комфортним: в Google Meet або Zoom викладачі проводять заняття, Google Classroom/Drive або Moodle використовують для поширення файлів серед учнів, в Moodle викладачі створюють тести для перевірки знань учнів. Але все ж деякі навчальні заклади, а особливо навчальні курси віддають перевагу використанню власних застосунків, в яких функціонал набагато ширший.

Серед приватних онлайн шкіл розповсюджено придбання та використання сторонніх API, до яких можна підключити власну клієнтську частину та при необхідності доповнити потрібними функціями. Тому розробка Application Programming Interface (API) є актуальною. API – це інтерфейс взаємодії між сайтом і сторонніми програмами і серверами. Розробники можуть скористатися API для отримання доступу до функціоналу сторонньої програми. API робить можливим роботу ресурсів, які використовують потенціал і потужність іншого сайту або програми. При використанні API скорочується необхідність створювати самостійно складні програми або додатки. Замість цього можна використовувати готові частини існуючих ресурсів, у яких є доступ до потрібної інформації і даних. Отже, створення API для підтримки клієнтських застосувань для онлайн шкіл є актуальною задачею.

Ця система матиме мікросервісну архітектуру, яка складається з окремих, слабо пов'язаних компонентів-сервісів, кожен з яких можна розробляти, розгортати, експлуатувати, змінювати та розгортати повторно, не порушуючи роботу інших сервісів та цілісність програми. Це дозволяє легко та швидко розгортати окремі можливості програми. В мікросервісній архітектурі розробники можуть експериментувати з новими можливостями та повертатися до попередньої версії, якщо щось піде не так. Це полегшує оновлення коду та прискорює

виведення нових можливостей на ринок. Крім того, так стає простіше шукати та виправляти помилки та баги в окремих сервісах.

При розробці API для онлайн шкіл також пропонується використання концепції CRUD. CRUD - це акронім від CREATE, READ, UPDATE, DELETE (Створити, Прочитати, Оновити, Видалити). CRUD потрібний для того, щоб розділити програмний код на окремі частини, кожна з яких відповідає за свої дії.

Операції CRUD призначені для редагування записів бази даних. Можна розглянути детальніше, що означає кожна операція:

- GET – метод GET дозволяє отримати інформацію із джерела/бази даних;
- POST – метод POST дозволяє вносити інформацію до джерела/бази даних;
- PUT – метод PUT дозволяє оновлювати існуючу інформацію у джерелі/базі даних;
- DELETE – метод DELETE видаляти існуючу інформацію із джерела/бази даних.

CRUD спрощує та полегшує проектування додатків, роблячи їх більш масштабованими, полегшує контроль безпеки, задовольняючи різні вимоги доступу.

Отже, якісно розроблені сервіси для онлайн шкіл значно оптимізують для них роботу функціоналу та процес зберігання та оброблення даних. API повинен бути логічно побудованим та масштабованим, надавати наступний функціонал:

- реєстрація та авторизація користувача;
- відновлення пароля;
- розподіл учнів по групах;
- розподіл користувачів по ролях (учень, вчитель, адміністратор);
- розміщення відеозаписів з занять в особистих кабінетах учнів;
- розміщення файлів різних форматів (.pdf, .docx, .pptx та інші) в особистих кабінетах учнів;

- завантаження учнем виконаних завдань (в різних форматах) до особистого кабінету;
- розміщення тестів (з впродюкованими або рандомними питаннями, з обмеженням часу та без, з додатковими коментарями до відповідей та без) в особистих кабінетах учнів;
- наявність не тільки типічних тестів, а і завдань в більш цікавому форматі, наприклад, зібрати із невпорядкованих частин речення;
- розміщення розкладу занять в особистих кабінетах учнів в залежності від групи;
- швидкий пошук за назвою.

Пропонується провести огляд існуючих систем зберігання та оброблення даних для онлайн шкіл. Приклади будуть розглянуті у наступному розділі.

1.2 Огляд існуючих методів вирішення завдання

1.2.1 Сайт Ruzuku

На сайті guzuku.com можна створювати власні курси, користуючись інструментами цього вебзастосунку. Перевагами даної програми є різноманітність функцій, наприклад, в тестах це реалізація можливості перездачі тесту, невпорядкований формат питань/відповідей, правильна відповідь може бути одна або декілька (рис. 1.1). Також при створенні курсів можна завантажити файли різних форматів. Якщо у онлайн школи немає суворих вимог до функціоналу, то їй буде достатньо цього сайту, щоб виконувати базові функції. При тестуванні сайту були виявлені недоліки, пов'язані із клієнтською частиною, наприклад, нагромадженість інтерфейсу, відсутність модальних вікон із підтвердженням перед видаленням елементів.

Quiz | View Results ✓ saved ✕

☐ Allow retaking after submission

☐ Shuffle question order

☐ Shuffle option order

Question 1

☒ Option 1 ✓

☐ Add option

☒ Choose Answers (5 points) ☐ Allow Multiple Answers

Рисунок 1.1 — Сторінка створення тесту

1.2.2 Сайт Master test

Іншим прикладом сайту, за допомогою якого можна проводити тестування/ділитись матеріалами з учнями, є master-test.net. Перше, на що приділяє увагу користувач, це застарілий дизайн. На сайті можна створювати тести, але функціонал елементарний, також по інтерфейсу цього вебзастосунку можна зрозуміти, що він не надає багато можливостей для користувачів (рис. 1.2, 1.3)

Добавить вопрос Изменить титул Проверить тест Результат Сохранить

Тест

Вопрос 1

Заголовок
Вопрос: Вопрос 1

☐ Дополнительно

Тип вопроса: Однозначный Ответ

Ответ

☐ ответ 1 Удалить

☐ ответ 2 Удалить

Добавить Ответ

Вес Вопроса: 1

Готово

Рисунок 1.2 – Створення тесту в Master Test

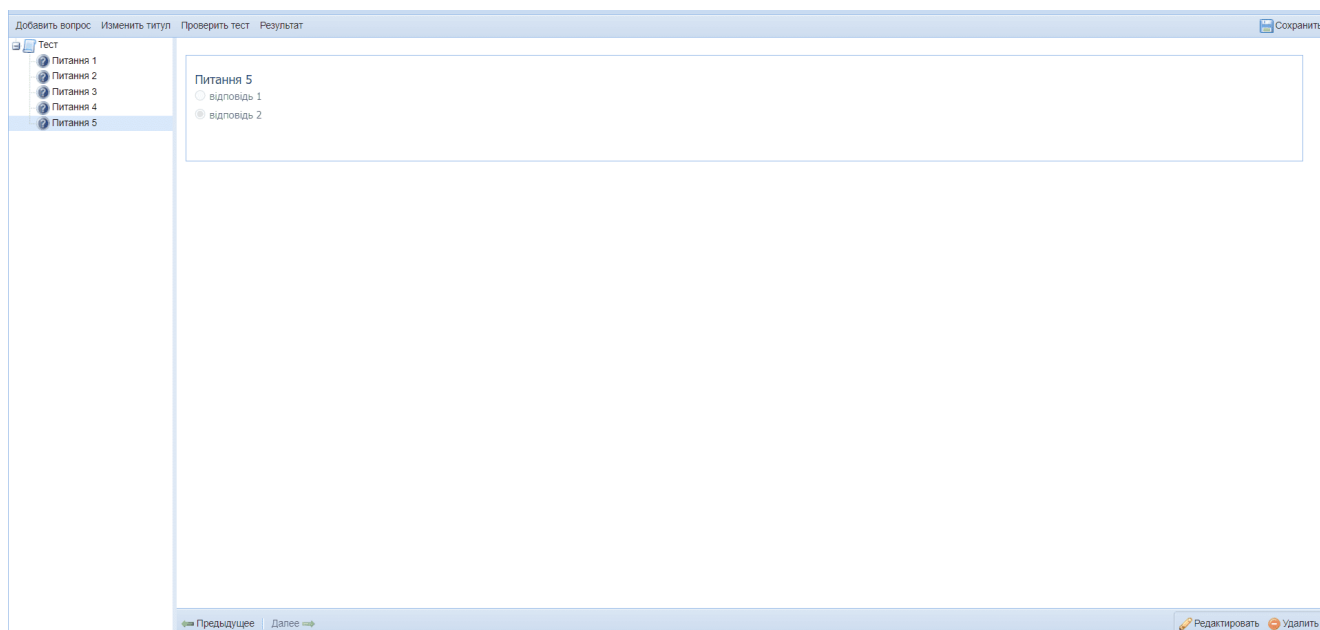


Рисунок 1.3 – Сторінка тесту в Master Test

1.2.3 Порівняння існуючих систем

На основі проведеного аналізу існуючих рішень, дані порівняння було внесено в табл. 1.1. Планується розробити власну систему з уникненням недоліків вже існуючих систем.

Таблиця 1.1 – Порівняння існуючих систем

Функції	Система	
	Ruzuku	Master Test
Створення курсів	так	так
Завантаження файлів різних форматів	так	ні
Перездача тесту	так	ні
Змішаний порядок питань/відповідей	так	ні
Зміна кількості правильних відповідей	так	так
Наявність розкладу занять для учнів	ні	ні
Пошук за назвою на сторінках	ні	ні
Завантаження учнем домашніх завдань (файли різних форматів)	ні	ні

1.3 Технічне завдання

Необхідно розробити архітектуру системи зберігання та оброблення даних для онлайн шкіл, програма має містити в собі:

- кінцеві точки (endpoints) для реєстрації, авторизації користувача, відновлення пароля;
- запит на отримання даних користувача і присутність обмежень (інші користувачі не мають доступу до особистих даних інших користувачів, виняток - адміністратор);
- запит на отримання списку домашніх завдань окремого користувача з фільтрацією та пошуком;
- запит на отримання списку навчальних файлів, які призначені для користувача, з фільтрацією та пошуком;
- запит на отримання розкладу користувача;
- CRUD для наступних сутностей: користувачі, групи, навчальні файли, тести та інші типи завдань, домашні завдання.

1.4 Висновки за розділом 1

Після аналізу предметної області було виявлено, що онлайн навчання поширено, як ніколи, тому реалізація API для онлайн школи – актуальна задача.

Програмний продукт дипломної роботи має поєднати в собі функціонал, який є у існуючих альтернативах та стати новим інструментом для підтримки клієнтської частини онлайн шкіл.

2 МАТЕРІАЛИ І МЕТОДИ

2.1 Дослідження підходів до тестування програмного забезпечення

Тестування програмного забезпечення – це один із найважливіших етапів створення продукту. Якщо не приділити тестуванню належної уваги, не можна досягти високої якості на виході. Всі успішні компанії, які займаються розробкою додатків для різних бізнес-цілей, мають у штаті висококваліфікованих тестувальників, які беруть участь в проекті на різних етапах. Тестування дозволяє знаходити та виправляти дефекти, тим самим підвищує якість продукту та знижує рівень ризику. При тестуванні також перевіряються і місця інтерфейсу користувача, де користувач може зробити помилку або неправильно зрозуміти висновки програми, а також стійкість системи до зловмисних дій. Але тестуванням займаються не тільки тестувальники. Існують техніки, які використовують програмісти при розробці програмного забезпечення, завдяки яким мінімізується кількість багів, наприклад, розробка через тестування (Test-Driven Development – TDD), розробка на основі поведінки (Behavior Driven Development – BDD), тестування після розробки (Test Last Development – TLD), розробка через приймальне тестування (Acceptance test-driven development – ATDD).

2.1.1 Дослідження Behavior Driven Development

Behavior Driven Development – це розробка на основі поведінки, що означає процес розробки програмного забезпечення, заснований на методології Agile.

Концепція BDD походить від розробки через тестування, процес розробки програмного забезпечення зосереджений на вимогах кінцевого користувача та його взаємодії з продуктом. Під час створення продукту на основі BDD враховуються наступні фактори:

- якою буде загальна поведінка програми для користувачів;

- наскільки добре основні функції програми працюватимуть для кінцевих користувачів;

- який відгук буде у аудиторії під час взаємодії з програмою.

Behavior Driven Development – доволі затребувана методологія розробки програмного забезпечення, яка використовується для розробки різноманітних програм для різних секторів, від B2B до SaaS. BDD базується на трьох стовпах:

- discovery (відкриття) – на цьому етапі розкривається проблема клієнта та її вирішення;

- formulation (формулювання) – критерії приймання перетворюються на приймальні випробування;

- automation (автоматизація).

Розробка на основі BDD визначається як процес, який зосереджується на вимогах і очікуваннях користувачів, сприяючи співпраці між розробниками, тестувальниками та керівниками проектів. Тобто концепція BDD полягає в тому, щоб дозволити бізнес-стейкхолдерам, розробникам, тестувальникам чітко зрозуміти поведінку програми та забезпечити співпрацю між технічними та нетехнічними командами. Розробка на основі поведінки допомагає розробляти орієнтовані на користувача продукти шляхом документування та проектування відповідно до очікувань користувача та взаємодії з продуктом.

Розглянемо переваги BDD:

- цей підхід можна також вважати Customer-driven product development, тобто розробкою продукту, керованою клієнтом. Завдяки ранньому врахуванню відгуків клієнтів і застосуванню методології Agile, BDD забезпечує створення вискоєфективного продукту, орієнтованого на клієнта, який не тільки відповідає вимогам клієнтів, але і перевищує їх;

- більша прозорість, а це є ключем до успіху для будь – якого підходу до розробки продукту, особливо при дотриманні позиції співпраці. Розробка, яка орієнтована на поведінку, забезпечує кращу видимість функцій системи, щоб і розробники, і тестувальники, і бізнес – стейкхолдери, і інші члени команди могли чітко розуміти, що розробляється, чи відповідають функції певному набору вимог

чи ні. Тому завдяки прозорості під час розробки можна уникнути ймовірності конфліктів чи двозначності;

- правильне визначення пріоритетів функцій. Це дійсно важливо, бо пріоритезація функцій продукту відіграє вирішальну роль у розробці програмного забезпечення;

- зменшення технічного обслуговування та ризиків проекту. Звісно, розробка будь-якого продукту пов'язана з ризиками, наприклад, збій функцій продукту, ймовірність неефективної розробки функцій, проблеми сумісності, витрати на перерозробку та час;

- краще узгодження із загальними бізнес-цілями. Оскільки цей підхід до розробки приділяє особливу увагу кожній функції, деталі про те, для якої мети вони служать, наскільки добре це підходить для поточного ринку, як це може сприяти збільшенню прибутку можна передати краще. В результаті, розробка продукту, який відповідає критеріям бізнесу, стає легшою [3].

Звісно, перелічені переваги доводять, що цей підхід до розробки ефективний, але все ж недоліків в BDD теж немало: цей підхід довгий та дорогий, тому його не так часто використовують в компаніях. Він незручний хоча б тим, що вже на етапі опрацювання вимог вимагає залучення фахівців тестування.

Розглянемо недоліки BDD:

- створення сценаріїв і підтримка файлів вимагає багато зусиль і часу. Тому це не підходить для короткого проекту, який необхідно завершити за короткий проміжок часу;

- необхідна хороша комунікація між людиною, яка розробляє код автоматизації, і особою, яка пише файли функцій. Особі, яка пише автоматизацію, потрібні ці файли та сценарії для розробки сценарію автоматизації. Якщо у них немає взаємного розуміння файлів, тоді важко розвивати проект;

- доволі важко перетворити оператори Given на інструкції з налаштування та сценарії, які переводитимуть систему у відомий стан перед виконанням операторів When;

— необхідно створити документ для проекту BDD, оскільки існує багато файлів і сценаріїв, які потрібно зрозуміти, щоб можна було створити документацію [4].

Пропонується в наступній частині роботи проаналізувати інші методи розробки, порівняти їх, обрати більш підходящий для цього проекту підхід та застосувати його.

2.1.2 Дослідження підходів Test Driven Development та Test Last Development

Розглянемо принципи TLD та TDD. Test Last Development – це тестування після реалізації функціоналу, зазвичай саме цю методику використовують програмісти при розробці програмного забезпечення. А Test-Driven Development – це техніка розробки програмного забезпечення, яка пропонує повторення коротких циклів розробки: спочатку пишеться тест, що покриває бажану зміну, потім пишеться код, який дозволить пройти тест, і потім проводиться рефакторинг нового коду відповідно до стандартів.

TDD існує з 1999 року і є методом розробки програмного забезпечення на основі тестування. Американський розробник Кент Бек «заново відкрив» TDD у 2003 році і застосував його як спосіб створення простих конструкцій. На сьогоднішній день, TDD – це процес розробки програмного забезпечення, який використовує дуже короткий цикл зворотного зв'язку (зображено на рис. 2.1), при якому розробники:

- створюють тест, який не може бути пройдено;
- пишуть робочий код, який пройде тест;
- роблять рефакторинг коду, написаного на другому кроці, використовуючи створений тест, доводять код до досконалості.



Рисунок 2.1 – Цикл TDD

Метою TDD є специфікація, а не валідація. Це один із способів продумати вимоги або дизайн до того, як функціональний код буде написано. TDD дозволяє написати чистий код, який працює.

Підхід TDD поєднує програмування з написанням тестів самим розробником, цим і відрізняється від інших методів. Ця концепція відновлює загальну повагу до тестів, створених програмістом. Розробка через тестування – це підхід до програмування, у якому програмування має вестися виходячи з тестування, а не навпаки. При традиційному підході до розробки програмного забезпечення замовник висуває вимоги до програми, після цього розробник створює код, який відповідає критеріям прийнятності користувальницької історії. Потім код буде об'єднаний та протестований тестувальником у середовищі QA або SIT. Тестер може виявити дефекти. У цьому випадку код вирушає на доопрацювання. Тобто існує цикл: написати код, запустити тести, виправити код. Test Driven Development перевертає цей цикл із ніг на голову. У ньому йдеться про те, що ми повинні спочатку створити тести, потім написати код, після чого виправити код. У Test Driven Development спочатку пишеться і запускається тест, потім пишеться код (до тих пір, поки він не пройде тест), а потім код

реорганізується. Це називається рефакторинг. Мета рефакторингу – це чистий код, який ідеально працює. Рефакторинг – це модифікація вихідного коду системи після незначних ізольованих змін, які призводять до зміни функціональності. Функціональність перевіряється під час рефакторингу шляхом багаторазового запуску тестів. Рефакторинг значно покращує читання коду, усуває дублювання, полегшує технічне обслуговування та розширення системи у майбутньому. Так як зміни невеликі, певні та контрольовані, їх також можна автоматизувати за допомогою інструментів рефакторингу, які вже інтегровані у багато популярних середовищ розробки, що значно підвищує продуктивність праці розробників.

Кент Бек підсумовує принцип роботи TDD в програмуванні в фразу «червоний, зелений, рефакторинг». Назви кольорів відносяться до інтерфейсу популярного автоматичного запуску тесту JUnit, де червоний колір вказує на те, що тест не пройдено, а зелений – тест пройшов успішно. Це основа тестового програмування, відповідно до якої процес починається з написання деяких тестів, і вони провалюються (червоний). Червона фаза змушує замислитись про вимоги, перш ніж почати писати код. Завдяки цьому система, над якою працюють, буде краще продумана. Виявляються невідповідності, що допомагають покращити роботу системи, перш ніж писати код. Після цього пишеться код, доки тести не будуть пройдені (зелений). На зеленій стадії додається мінімальний обсяг виробничого коду, який необхідний для проходження тестів. Потім код доопрацьовується. І так як тести вже готові, їх можна знову запустити і переконатися, що рефакторинг не вплинув на функціональність.

Кент Бек визначив два простих правила для TDD: писати новий бізнес-код потрібно лише у разі збою автоматизованого тесту; необхідно усунути будь-яке дублювання, яке буде знайдено. Ці два простих правила генерують складну індивідуальну та групову поведінку:

- розробляється працюючий код, що забезпечує зворотний зв'язок між рішеннями;
- програміст пише власні тести, тому що не може чекати 20 разів на день, щоб хтось інший написав їх для нього;

— середовище розробки повинне швидко реагувати на невеликі зміни (необхідний швидкий компілятор та набір регресійних тестів).

Переваги TDD:

— зменшення залежності від налагодження: так як спочатку увага приділяється написанню тестів, а потім - коду, який призначений для проходження тестів, життєвий цикл може значно скоротити потребу в налагодженні. Через те, що під час написання та кодування тесту необхідне глибше розуміння логічних та функціональних вимог, причину невдалого тесту зазвичай можна швидко розпізнати та усунути;

— аналізує досвід користувача: процес початкового осмислення і написання тесту змушує мозок програміста працювати навпаки: спочатку продумувати, як буде використовуватися функція і як вона може бути реалізована, а потім, як повинен бути написаний тест. Це спонукає враховувати особливості користувача;

— скорочує загальний час розробки: порівняно з традиційними методами написання коду при застосуванні TDD через тести загальна кількість рядків коду збільшується, але часто саме таке тестування виключає помилки в процесі і виявляє існуючі набагато раніше, запобігаючи виникненню проблем надалі [5].

Отже, між TDD та TLD було обрано TDD через його явні переваги. Розглянемо різницю між TDD та BDD, щоб обрати підхід для розробки програмного забезпечення (табл. 2.1).

Таблиця 2.1 – Порівняння TDD та BDD

TDD	BDD
Визначення	
TDD — це техніка розробки, яка зосереджена на роботі функцій.	BDD — це техніка розробки, спрямована на поведінку системи.
Контриб'ютор	
Розробники	Розробники, QA, клієнти
Фокус на	
Юніт тестуванні	Розумінні вимог

Враховуючи проведений аналіз, пропонується обрати методологію Test Driven Development, так як в даній роботі більше уваги приділяється саме технічній частині процесу розробки продукту. При розробці проекту буде використано підхід TDD. Отже, спочатку напишемо тести, щоб потім написати функціонал під тест. В даному коді два тест-кейси перевіряють правильність роботи авторизації, а саме: якщо введені правильні значення для авторизації, то ми повинні отримати токен доступу, інакше – повинні отримати код помилки (422) та повідомлення про те, які саме дані були введені некоректно.

```
describe('POST: login', () => {
  test('return access token', async () => {
    const result = await login('test@example.com', 'secret')
    expect(result).toHaveProperty('accessToken')
  })
  test('return throw if payload is not valid', async () => {
    const result = await login('test', 'secret')
    expect(result).toHaveProperty('code', 422)
    expect(result).toHaveProperty('message')
  })
})
```

Після того, як ми написали тести, ми можемо їх запустити: всі тести були провалені, але це означає, що ми все зробили правильно, бо метод ще не реалізовано. Потім приступаємо до реалізації методу авторизації. В першу чергу необхідно створити головний метод авторизації, в якому відбувається основна робота з отриманими даними, а саме: перевірка на наявність email-у в базі даних, хешування отриманого пароля та порівняння його з паролем, який лежить в базі даних; якщо дані валідні, за допомогою пакету 'jsonwebtoken' створюємо токен доступу та відправляємо користувачу:

```
import bcrypt from 'bcryptjs';
import jwt from 'jsonwebtoken';
import { SECRET } from '@config/index';
import { EntityReturn } from '@constants/Common';
import { EMAIL_CREDENTIAL_ERRORS, GENERIC_ERRORS } from '@constants/Error';
import { UserRole } from '@models/User';
import { EmailCredential, User } from '@models/index';
import { Transaction, doInTransaction } from '@services/Db';
export default async function login({
  email,
  password,
  transaction: tx,
}: {
  email: string;
  password: string;
  transaction?: Transaction;
}): Promise<EntityReturn> {
  return doInTransaction(async (transaction) => {
    try {
```

```

    const emailCredential = await
EmailCredential.scope('withPassword').findOne({
  where: { email },
  include: [{ model: User, where: { role: UserRole.user }, required: true
}],
  transaction,
});
if (!emailCredential) {
  await transaction.rollback();
  return {
    code: EMAIL_CREDENTIAL_ERRORS.NOT_FOUND,
    message: 'Incorrect email or password',
  };
}
const isMatch = bcrypt.compareSync(password, emailCredential.password);
if (!isMatch) {
  await transaction.rollback();
  return {
    code: EMAIL_CREDENTIAL_ERRORS.PASSWORD_NOT_MATCH,
    message: 'Password not match',
  };
}
await transaction.commit();
return {
  result: {
    accessToken: jwt.sign({ userId: emailCredential.userId }, SECRET.JWT),
  },
};
} catch (error: any) {
  await transaction.rollback();
  return {
    code: GENERIC_ERRORS.ERROR,
    message: 'Something went wrong',
  };
}
}, tx);
}

```

Далі необхідно створити Controller – головний метод для комунікацій з клієнтською частиною. В ньому відбуваються основні перевірки валідації, для який необхідно підключення до бази даних. Якщо введені дані валідні, ми передаємо їх в основний метод, який було описано вище. Після чого ми отримуємо відповідь і, в залежності від неї, ми віддаємо на клієнт токен доступу або текст помилки відповідно.

```

import { Request, Response } from 'express';
import StatusType from '@constants/StatusType';
import { getJsonError, getJsonResponse } from '@constants/ResponseType';
import { DEFAULT_JOI_RESPONSE, LOGIN_PAYLOAD } from '@constants/Validations';
import AuthEntity from '@entities/Auth';
export const login = async (req: Request, res: Response) => {
  const payload = (req.body && req.body.payload) || null;
  const { error, value } = (payload && LOGIN_PAYLOAD.validate(payload)) ||
DEFAULT_JOI_RESPONSE;
  if (error || !value || (!error && !value)) {
    return res.status(StatusType.DB_ERROR).json(getJsonError(error));
  }
  const data = await AuthEntity.login(value);
  if (data.result) {

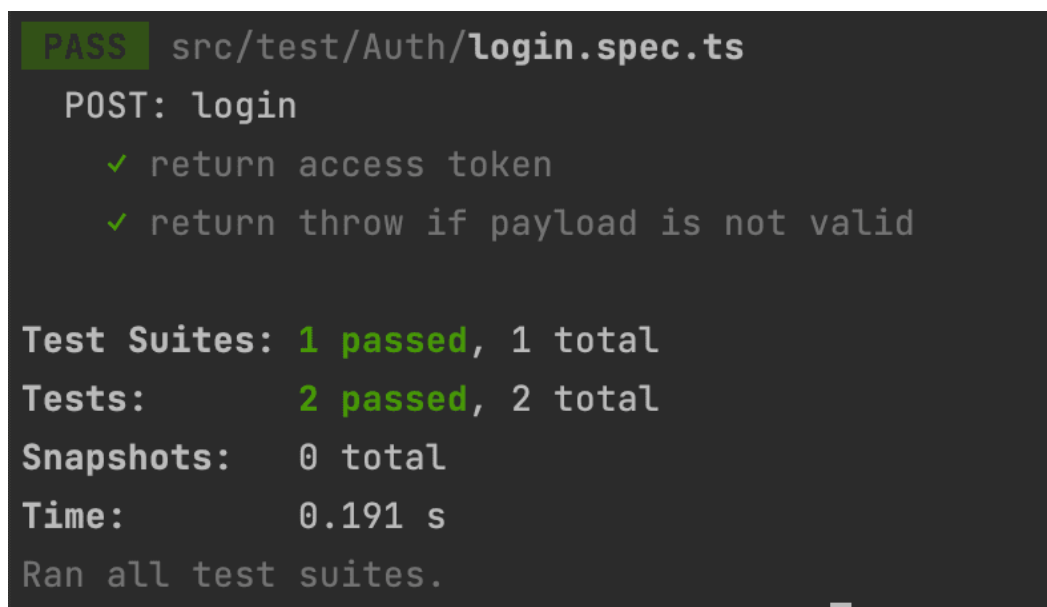
```

```

    return res.status(StatusCode.SUCCESS).json(getJsonResponse(data));
  } else {
    return res.status(StatusCode.DB_ERROR).json(getJsonError(data));
  }
};

```

Після цього, ми запускаємо тест (рис. 2.2) для перевірки роботи авторизації, і якщо всі тести виконані без помилок, то метод реалізовано коректно.



```

PASS src/test/Auth/login.spec.ts
  POST: login
    ✓ return access token
    ✓ return throw if payload is not valid

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        0.191 s
Ran all test suites.

```

Рисунок 2.2 – Результати запуску тесту

2.2 Вибір між монолітним підходом та мікросервісною архітектурою

Ще в процесі розробки програмного забезпечення необхідно мати на увазі, що при успішному запуску проекту, коли користувачів буде багато, з великою ймовірністю потрібно буде додавати на сайт новий функціонал, щоб задовольняти запитам користувачів. Коли програмне забезпечення стає більш об'ємним, над оновленнями потрібно працювати довше, а їх тестування стає складнішим. Щоб вирішити цю проблему, можна використати мікросервісну архітектуру, таким чином, всі «важкі» процеси можна буде винести в спеціальне оточення. Тобто ми беремо важкий процес, прибираємо його з додатку та запускаємо в окремому контейнері, де він працює сам по собі, а ми звертаємось до нього при необхідності. Мікросервіс, іноді називається сервісом, може бути як великим, так і зовсім маленьким, це не принципово.

Мікросервісна архітектура є протилежністю монолітного підходу розробки програмного забезпечення. При монолітному підході, який вважається класичним для розробки, вся робота додатку реалізується на одній кодовій базі, так простіше в процесі розробки, бо можна реалізовувати необхідний функціонал та не думати про проблеми комунікації компонентів. Додатки, які створені з застосуванням монолітного підходу, містять окремі класи та функції, які тісно пов'язані один з одним. При вилученні одного модуля будуть зміни у роботі всієї системи, що призведе до низки значних недоліків:

- масштабування окремого модуля неможливе, доведеться переробляти весь додаток;
- для внесення однієї зміни необхідно перезібрати всю програму;
- при поломці одного модуля, скоріш за все, перестане працювати весь сервіс;
- розробка обмежена попередньо обраними мовами програмування, фреймворками та іншими інструментами, не вийде використати навіть щось альтернативне;
- через складну структуру зв'язків між модулями вихід оновлень уповільнюється, бо кожне оновлення вимагає серйозного тестування на баги та регресії коду;
- коли додаток невеликий, його структура зрозуміла, але через деякий час рамки модулів стираються, виникає все більше неочевидних взаємозв'язків;
- складнощі виникають і з командою проекту, кожний учасник має розуміти всі бізнес-функції, також потрібно набагато більше часу на те, щоб додати нового програміста в роботу, йому потрібно буде вивчити весь код.

На початку 2010-х років розповсюдження хмарних сервісів частково призвело до розачарування розробників щодо класичного варіанту архітектури додатків. В якості альтернативи була запропонована архітектура мікросервісів, як розподілена система найпростіших модулів, які легко змінюються. Вони повинні працювати, як робітники конвеєра, тобто виконувати одну просту функцію та передавати завдання далі, при цьому мікросервіси вибудовуються не ієрархічно, а

симетрично. Сервіси зв'язуються між собою та з клієнтами використовуючи легкі протоколи, як http. Таким чином створюється проста для налаштування система, яку легко оновлювати. Мікросервіси незалежні від довкілля та мають високий рівень повторного використання, в цьому і полягає їх головна ідея. Кожний окремий сервіс можна програмувати саме тією мовою, яка буде зручна команді, це особливо корисно у великих проектах. Завдяки такій незалежності можна поділяти розробку компонентів системи між самостійними командами, а програміст, який прийшов у команду, вивче функціональні особливості тільки того мікросервісу, з яким він буде працювати, а не освоює систему цілком. Відмова одного сервісу не призводить до зупинення всієї системи. Коли помилку виправляють, необхідну зміну можна розгорнути лише для відповідного сервісу замість повторного налаштування всієї програми. Але звісно, для цього потрібно ще на етапі проектування продумати зв'язок між сервісами, щоб досягти максимальної незалежності один від одного. А у разі помилки користувачі повинні отримувати коректні сповіщення про тимчасову недоступність певного сервісу без шкоди для всієї системи.

Також великою перевагою використання мікросервісної архітектури є те, що вона дозволяє підвищити відмовостійкість. Всі чули про масштабні аварії в дата-центрах за останні роки, коли дані на серверах десятків тисяч клієнтів безповоротно втрачались. При використанні мікросервісної архітектури навіть якщо дата-центр згорить, програма продовжить працювати, вона просто переключиться на резервну копію [6].

Розглянемо відмінності між мікросервісним та монолітним додатком в таблиці 2.2.

Таблиця 2.2 – Порівняння мікросервісного та монолітного підходу

Мікросервісний додаток	Монолітний додаток
Діапазон технологій	
Кожний сервіс може бути написаним на своїй мові програмування, мати власний спосіб зберігання даних, необхідні бібліотеки.	Код монолітного додатку є єдиним цілим, тому вся команда розробників повинна дотримуватись вже обраних інструментів та методів.

Продовження таблиці 2.3

Мікросервісний додаток	Монолітний додаток
Швидкість релізів	
Для запуску нових функцій або оновлень вже існуючих, достатньо лише замінити один модуль програми. Завдяки цьому прискорюється розробка, можна частіше випускати оновлення.	Для того, щоб протестувати реліз та підготувати до нього додаток, необхідно оновити повністю всю систему, що може збільшити час налагодження та призвести до несподіваних збоїв.
Оптимізація додатку	
Управління функціональністю, ресурсами та інфраструктурою програми можна довірити різним сервісам та оптимізувати кожен із них окремо.	При оптимізації монолітного додатку необхідно не забувати про постійне враховування внутрішніх зв'язків між його модулями, бо оновлення хоча б одного модуля призведе до зміни системи в цілому.
Процес розробки	
Завдяки мікросервісній архітектурі можна вести гнучку розробку та за необхідності швидко змінити склад команди або вимоги до продукту.	Монолітний підхід більш вимогливий до порога входження, бо кожному новому розробнику в команді проекту доведеться повністю вивчити код програми та її функціональність.
Відмовостійкість	
При наявності проблем всередині одного сервісу, робота системи загалом не порушується, нові помилки не з'являються	Всі елементи монолітного додатку безпосередньо або опосередковано пов'язані один з одним, тому збій всередині одного модуля може призвести до повної відмови системи
Масштабованість	
При збільшенні навантаження на ресурси мікросервісів, вони автоматично масштабуються, а гнучкий процес розробки дозволяє посилювати команду додатковими працівниками.	При зміні хоча б одного блоку, буде необхідно масштабувати весь додаток.

Звісно, мікросервісна архітектура має багато переваг, але вона також не позбавлена недоліків, тому перед початком розробки програми необхідно розібрати і їх, щоб розуміти, з чим доведеться мати справу. Поділ великого продукту на самостійні сервіси здається простим та логічним, але розробка розподіленої системи – це доволі складний процес, як з технічної, так і з організаційної точок зору. Тому в деяких випадках переваги можуть обернутися недоліками, наприклад:

— простіше та ефективніше контролювати якість вирішення окремих бізнес-завдань, ніж оцінювати систему в цілому, але насправді не так легко налаштовувати робочі процеси великої команди розробників;

— можна буде зекономити тільки, якщо платити лише за ті ресурси, які споживають мікросервіси, але також необхідно передбачити витрати на впровадження хмарних технологій, окреме розгортання кожного нового сервісу та його покриття окремими моніторингами та тестами;

— збій одного сервісу хоч і не призведе до повної відмови додатку, але все ж будь яка розподілена система має інші слабкі місця: мережні затримки, потенційні проблеми зв'язку елементів програми один з одним, можлива неконсистентність даних.

Отже, після аналізу мікросервісного та монолітного підходу було вирішено, що використання мікросервісної архітектури більше підходить для розробки даного проекту.

2.3 Тестування навантаження

Традиційною, але не менш важливою частиною процесу розробки програмного забезпечення є тестування навантаження (load testing).

Тестування навантаження – це такий вид тестування, який визначає стабільність, працездатність та споживання ресурсів додатка при різних умовах навантажень та використання. При тестуванні навантаження імітується реальне навантаження на програму. Якщо не використовувати цей вид тестування перед запуском проекту, то велика ймовірність того, що в реальних умовах користування програмою вона не витримає та зламається. Тестування навантаження перевіряє, як само програмне забезпечення поводить себе при нормальних та при високих навантаженнях, завдяки цьому можна зрозуміти, чи зможе система витримати високі навантаження при високому попиті кінцевих користувачів. Зазвичай тестування навантаження застосовується, коли розробка програмного забезпечення наближається до завершення.

Тестування навантаження проводяться за допомогою end-to-end систем або менших компонентів, таких як сервери баз даних або брандмауерів. Такий тест вимірює швидкість або потужність системи чи компонента через час відповіді на запити. Коли час відповіді системних компонентів різко подовжується, або якщо вони стають нестабільними, це означає, що система, ймовірно, досягла максимальної робочої потужності. Коли таке трапляється, необхідно визначити слабкі місця програмного забезпечення та запропонувати рішення проблем.

Переваги тестування навантаження включають виявлення слабких місць перед релізом програми, масштабованість, скорочення часу простою системи, підвищення рівня задоволеності клієнтів і зниження витрат на те, щоб виправити помилки, які призвели до збоїв вже після релізу проекту, детальніше:

- виявлення слабких місць перед релізом програми. Оцінка повного програмного забезпечення або її частини перед релізом дозволяє знайти та усунути помилки своєчасно, мінімізуючи кількість проблем в майбутньому;

- підвищення рівня задоволеності клієнтів. Якщо час відповіді вебзастосунку короткий, то ймовірність того, що користувач побажає при необхідності знову відвідати сайт, набагато вища. Сучасні користувачі не люблять чекати, зараз в інтернеті можна знайти велику кількість аналогічних вебсайтів, тому якщо один сайт буде довго завантажуватись та не буде справлятися зі своїми задачами, користувачу буде простіше закрити цей сайт та знайти новий;

- зниження витрат. Виявлення проблем в коді на якомога ранньому етапі, а особливо перед релізом, значно зменшує вартість витрат, бо збої, які відбулися вже після запуску проекту призводять до неймовірних витрат компанії.

Також варто розібрати «best practices» (кращі практики) для тестування навантаження. Best practices – це досвід розробників та висновки про те, що варто робити при розробці, а що не варто. Тобто це набір дій та принципів, які приводять до найбільш ефективного результату. Найкращий інструмент та програмне забезпечення – це ще не все, що потрібно для успішного тестування

навантаження програми. Більш важливим є знання практик та вміння їх застосовувати під час навантажувального тестування. Наведемо приклади:

- визначте бізнес-цілі. Чітке розуміння майбутніх цілей щодо масштабу та обсягу допоможе розробити точні вказівки для інформування процесу;
- визначте ключові параметри продуктивності програми. Необхідно узгодити критерії відстеження, наприклад, час відповіді, пропускну здатність, використання ресурсів, максимальне навантаження користувачів, показники ефективності бізнесу;
- виберіть відповідний інструмент, який найкраще відповідає саме вашим потребам;
- створіть тестовий приклад, переконайтеся, що враховано як позитивні, так і негативні сценарії. Тестові випадки мають бути точними, також їх можна відстежити відповідно до вимог;
- зрозумійте своє оточення. Створюйте такі конфігурації, які подібні до типового процесу після релізу. Перевірте можливості різних систем, наприклад, безпеку, обладнання, мережі.
- виконуйте тести поступово. Однією з ключових цілей є визначення того, який саме обсяг навантаження призводить до невдачі;
- завжди майте на увазі кінцевих споживачів. Задоволеність клієнтів і відвідувачів має вирішальне значення для досягнення бізнес – метрик. Це впливає на їхню готовність повторно відвідати сайт [7].

При навантажувальному тестуванні важливо обрати інструмент, який найкраще відповіде потребам певного проекту. Зараз існують багато різноманітних інструментів для тестування навантаження, їх було проаналізовано та обрано Prometheus та Grafana. Будь-який, навіть маленький, IT- проект – це різні метрики. Наприклад, кількість зареєстрованих в тиждень, середня кількість активних користувачів в день, середній чек на клієнта, кількість активних користувачів, які користуються новим функціоналом. Із метрик можна витягти багато корисного, зрозуміти, куди треба рухатись далі.

Платформа Prometheus дозволяє візуалізувати метрики на основі виразів, створених в PromQL (мова запитів Prometheus), він відображає результати запиту в табличному або графічному форматі для оперативного усунення несправностей.

Отже, Prometheus – це центральний сервер, який збирає та зберігає дані. Дані постійно змінюються у часі, наприклад, трафік через мережевий інтерфейс, рівень заповнення диска, час відгуку сайту. А метрики є елементами даних. Сервер Prometheus із заданою періодичністю зчитує метрики та поміщає отримані дані в Time Series DB (це різновид баз даних, який зберігає часові ряди, тобто значення з прив'язкою до часу). Також, Prometheus надає інтерфейс для виконання запитів та візуалізації отриманих даних. Він працює за моделлю pull, тобто сам опитує endpoints, щоб отримати дані. На вузли для збирання інформації встановлюються експортери. Exporters – це процеси, які забезпечують збирання та їх передачу серверу Prometheus, дані передаються в форматі http.

Але використання тільки Prometheus може бути недостатньо, на рис. 2.3 зображено приклад відображення даних Prometheus, а на рис. 2.4 зображено типічну візуалізацію Grafana. Різниця у представленні інформації велика.



Рисунок 2.3 – Приклад відображення даних Prometheus

Основні обмеження Prometheus:

- не підтримує drag-and-drop віджети;

- не підтримує спільний доступ;
- не інтегрує події та сповіщення у свою візуалізацію;
- шаблони консолі потребують програмування на Go та інше.

Через ці причини Prometheus сам рекомендує використовувати інструмент із відкритим кодом Grafana як рішення для візуалізації.

Grafana — це багатоплатформна вебпрограма для аналітики та візуалізації, яка має багато інструментів для відображення всіх ваших метричних даних, незалежно від того, де вони знаходяться. За допомогою Grafana можна:

- створювати діаграми, графіки та карти;
- створювати інформаційні панелі та ділитися ними;
- налаштовувати правила сповіщень, які надсилають запит до одного або кількох джерел даних;
- керувати доступом користувачів.

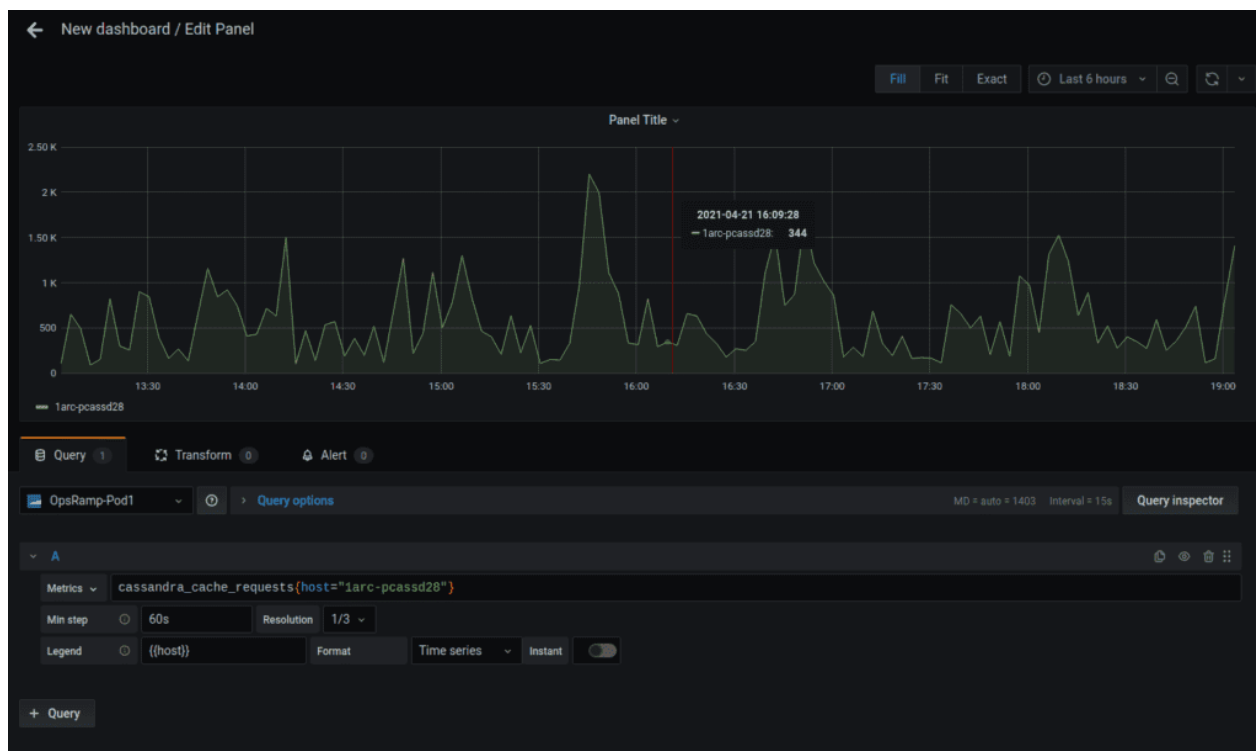


Рисунок 2.4 – Приклад типічної візуалізації Grafana

Отже Grafana є універсальною обгорткою для роботи з аналітичними даними, які зберігаються в різних джерелах. Grafana сама нічого не зберігає і не збирає, а лише виконує роль універсального клієнту для систем зберігання

метрик. Наприклад, за допомогою Grafana можна використовувати цифри і із традиційної бази PostgreSQL, і зі спеціалізованих аналітичних систем, як Influx або Prometheus. Grafana можна підключатись до будь-якого сховища статистичних даних та виконувати там певні запити. Запити конструюються у спеціальному зручному інтерфейсі, що допомагає зосередитись саме на даних, а не на правильності запитів у СУБД. Отримані результати Grafana показує у доступному вигляді: таблиці, графіки та інші формати відображення даних.

Єдине головне обмеження Grafana, про яке треба знати, полягає в тому, що необхідно налаштувати збір та зберігання даних окремо, і через це Grafana має обмежені можливості для кореляції між багатьма типами даних.

Виходячи з функцій окремо Prometheus, окремо Grafana, можна швидко зрозуміти, що ці два рішення здебільшого доповнюють один одного. Prometheus якісно збирає показники (метрики) та надає потужну мову запитів, а Grafana перетворює ці показники на значущі візуалізації. Також обидва вони сумісні з багатьма, навіть з більшістю типів джерел даних.

Отже, було вирішено після розробки проекту провести тестування навантаження та використовувати для цього такі інструменти як Prometheus та Grafana.

2.4 Висновки за розділом 2

Було розглянуто підходи до розробки програмного забезпечення та використано методологію TDD на практиці. TDD ґрунтується на написанні тестів до розробки функціоналу. Було з'ясовано, що цей підхід допомагає зменшити кількість багів після основного етапу розробки програмного забезпечення. Також було розібрано тестування навантаження та проаналізовано інші підходи до розробки програмного забезпечення.

3 ОПИС ПРОГРАМИ

3.1 Вибір інструментарію розробки програмного забезпечення

3.1.1 Вибір платформи

Основною платформою було обрано Node.js. Це програмна платформа, яка робить JavaScript мовою загального призначення. Її також вважають середовищем виконання JS. Node.js зв'язується з зовнішніми бібліотеками, викликає команди з коду і виконує роль вебсервера. Тобто додає до повністю фронтендової мови бекендову частину та допомагає створювати з його допомогою повноцінні програми, без задіяння браузера. А для розробки програмного забезпечення було обрано саме JS, тому що ця мова програмування задовольнювала всім потребам більше, ніж інші. Докладне порівнювання мов програмування наведено у таблицях 3.1 – 3.4.

Однією із найважливіших переваг Node.js є неблокуюча модель введення-виведення. Ця система керована подіями і працює асинхронно, вибудовуючи чергу за пріоритетністю. Їй легко впоратися з навантаженням, коли до сервера одночасно підключаються тисячі людей, так як немає потреби створювати окремий потік для кожного підключення. Правильний розподіл ресурсів допомагає витримати більшу кількість з'єднань.

Також перевагою Node.js над іншими платформами є те, що він розроблений спеціально для web, тому може вільно взаємодіяти з найбільш популярними базами даних та допомагає отримати низькорівневий доступ (http, udp, https, tcp). Кількість готових модулів і зовнішніх бібліотек Node.js завжди зростає, чому сприяє використання npm.

Швидкість – це теж про Node.js. Бо за допомогою Node.js не відніме багато часу створення робочого прототипу, який справляється з навантаженням.

Технологія стрімко поліпшується, над цим працюють багато програмістів зі всього світу. Процес координує фонд Node. Складно недооцінити роль ком'юніті, адже це допомога зі складнощами, велика кількість навчальних матеріалів, оперативно знайдені помилки у функціонуванні системи, можливість

обговорити ідеї або знайти команду. З "живою" спільнотою розвиток технології відбувається набагато швидше, і все завдяки відкритому коду. Популярність і переваги Node.js не залишилися непоміченими, його вже вибрали такі компанії, як Google, Ebay / PayPal, Netflix, Microsoft, Uber, LinkedIn, Yahoo, Walmart і інші [9].

Таблиця 3.1 – Парадигми

Можливість	Мова					
	C#	Go	Java	JavaScript	PHP	Python
Імперативна	+	+	+	+	+	+
Об'єктно-орієнтована	+	+/-	+	+	+	+
Функціональна	+/-	+/-	+/-	+/-	+/-	+

Таблиця 3.2 – Типізація

Можливість	Мова					
	C#	Go	Java	JavaScript	PHP	Python
Статистична типізація	+	+	+	-	-	-
Динамічна типізація	+	-	-	+	+	+
Явна типізація	+	+	+	-	+/-	+/-
Неявна типізація	+/-	-	-	+	+	+

Таблиця 3.3 – Функціональні можливості

Можливість	Мова					
	C#	Go	Java	JavaScript	PHP	Python
First class functions	+	-	-	+	+	+
Анонімні функції	+	-	+	+	+	+
Лексичні замикання	+	-	+	+	+	+
Часткове застосування	-	-	-	+	-	+
Карирування	+	-	-	+	+	+

Таблиця 3.4 – Інше

Можливість	Мова					
	C#	Go	Java	JavaScript	PHP	Python
Пітримка Unicode в ідентифікаторах	+	+	+	+	+/-	+
Значення параметрів за замовчуванням	+	+	-	+	+	+
Наявність бібліотек для роботи з графікою і мультимедіа	+	-	+	+	+/-	+

3.1.2 Вибір фреймворку

Зараз існує багато різноманітних фреймворків для роботи з Node.js, найпопулярніші – Express, Fastify, Koa. В табл. 3.5 наведено порівняльний аналіз фреймворків з метою обрати кращий варіант для проекту, який розроблюється.

Таблиця 3.5 – Порівняння фреймворків

Критерії	Фреймворк		
	Express	Fastify	Koa
Можливості	Простий, мінімалістичний, гнучкий і масштабований. Швидка розробка програми. Повністю настраюється. Легка інтеграція сторонніх служб.	Блискауча швидкість порівняно з іншими популярними фреймворками Node. Забезпечує повну інкапсуляцію для плагінів, автоматично аналізує JSON із відносно швидшим відтворенням і забезпечує швидку маршрутизацію.	Важить дуже мало (близько 600 рядків коду), що робить його дуже легким. Koa не використовує зворотні виклики, отже полегшує вхідний і низхідний потік елементів керування. Рефакторинг більшої бази коду в Koa легший завдяки модульності.
Документація	Велика кількість документації	Поки що не так багато документації	Середній рівень об'єму документації
Підтримка	Високий рівень підтримки спільноти	Слабка підтримка спільноти для дебагінгу	Низький рівень підтримки спільноти

Отже, після проведеного аналізу для платформи Node.js було обрано фреймворк Express. Це гнучкий та мінімалістичний вебфреймворк для програм Node.js, який надає широкий набір функцій для мобільних та вебдодатків [10].

Це лідер серед базових JavaScript-фреймворків. Express також називають "фреймворком для інших фреймворків", так як існує багато інших фреймворків, які побудовані з використанням Express. Express.js було створено в 2010 році, зараз він підтримується Node.js Foundation. Основна перевага використання Express.js – швидка розробка на стороні сервера. Основні особливості цього фреймворку наступні: надійний механізм маршрутизації для обробки динамічних URL-адрес, можливість додавати додаткове програмне забезпечення до будь-якої точки конвеєра обробки запитів, спрощення налагодження коду, наявність шаблонізатора для динамічного рендерингу сторінок HTML на стороні сервера, а не на стороні клієнта. Для Express характерний невеликий обсяг базового функціоналу. Завдяки такому мінімалізму програміст спочатку отримує в своє розпорядження легкий і швидкий інструмент, який при необхідності можна розширювати і розвивати. Також перевагами Express є те, що він добре документований, інтегрується з усіма найбільш популярними системами управління базами даних (MongoDB, MySQL та інші), також всі популярні движки шаблонів, такі як EJS, HAML, Pug сумісні з Express.js.

3.1.3 Вибір мови програмування

TypeScript – мова програмування на основі JavaScript. Він зародився в компанії Microsoft, але одразу став розвиватися як OpenSource. TypeScript з самого початку став швидко поширюватись через свою гнучкість і продуктивність. TypeScript – це строго типізована мова. Головна перевага суворой статичної типізації - можливість знайти низку помилок ще на етапі написання коду. Також TypeScript реалізує багато концепцій, які властиві об'єктно-орієнтованим мовам (наприклад, успадкування, поліморфізм, інкапсуляція, модифікатори доступу та інше). Завдяки TypeScript можна швидше та простіше писати великі комплексні програми, також їх легше підтримувати, розвивати, масштабувати та тестувати, ніж на стандартному JavaScript [11].

3.1.4 Вибір середовища розробки

WebStorm – обране середовище розробки. Інструмент розроблений компанією JetBrains, підходить і для frontend'a, і для створення додатків на Node.js. Webstorm підтримує повноцінне налагодження Node.js додатків, нове програмне забезпечення можна створити, використовуючи шаблон Node.js Express, а необхідні модулі можна легко встановити за допомогою npm – менеджера, який вбудовано в WebStorm, тому за допомогою WebStorm можна ефективно розробляти програми на Node.js. WebStorm має зручний і розумний редактор JavaScript, HTML і CSS, який підтримує також інші мови та фреймворки (TypeScript, CoffeeScript, Dart, Less, Sass, Stylus, Angular, React, Meteor та інші). WebStorm робить розробку проекту простою та зручною, забезпечує підсвічування та автодоповнення коду, його аналіз по ходу редагування, швидку навігацію та рефакторинг. Він має потужні інструменти інтеграції з системами керування версіями (GitHub, Git, Subversion, Perforce, Mercurial, CVS) та інструменти для налагодження, розуміє структуру проекту та код, може відстежувати помилки за допомогою різних систем (JSHint, Stylelint, ESLint, JSLint, TSLint,), пропонує їх вирішення. Вбудовані в WebStorm інструменти для тестування та роботи з проектом допомагають у розробці та роблять її зручнішою та продуктивнішою. Звичайно, через великий функціонал WebStorm важкий і вимагає багато ресурсів. Але враховуючи велику кількість переваг, стабільність роботи, приємний інтерфейс програми, можна сказати, що WebStorm – найкращий IDE для веброзробки. Отже, WebStorm має необхідні інструменти для комфортного написання коду і усунення дефектів.

3.1.5 Огляд особливостей обраної системи управління базами даних

Система управління базами даних (СУБД) – це набір програмних та апаратних засобів, за допомогою яких можна проектувати, налаштовувати та

адмініструвати бази даних (БД). СУБД гарантує безпеку, цілісність, безпеку зберігання даних і дозволяє видавати доступ до адміністрування БД.

В якості СУБД було обрано PostgreSQL. Це одна з найвідоміших серед усіх існуючих реляційних баз даних, вона підтримується протягом 30 років розробки, вона заснована на відкритому коді, тому систему бази даних постійно покращує профільна спільнота з більш ніж 600 учасників. PostgreSQL популярна серед розробників завдяки своїй винятковій гнучкості та цілісності. Наприклад, база даних PostgreSQL підтримує як реляційні, і нереляційні запити. PostgreSQL сумісне з усіма основними операційними системами. PostgreSQL підтримує текст, зображення, звуки та відео. Отже, база даних користується популярністю у користувачів та компаній з різними потребами.

Переваги PostgreSQL:

- доступ до потужних функцій: наприклад, відновлення на момент часу, попереджувальне ведення журналу, елементи деталізованого керування доступом, табличні простори, вкладені транзакції, оперативне резервне копіювання та багатоваріантне керування паралелізмом;

- надійність та відповідність вимогам: PostgreSQL відповідає властивостям атомарності, узгодженості, ізолюваності та довговічності (ACID) для транзакцій баз даних. Також PostgreSQL підтримує кілька мов для різних тригерів, атрибутів зовнішніх ключів, об'єднань та процедур, що зберігаються;

- різноманітні типи індексування та повнотекстовий пошук: методи індексування, включаючи індексування на основі дерев B+, узагальнений інвертований індекс та узагальнене дерево пошуку;

- ліцензія на програмне забезпечення з відкритим кодом, завдяки чому користувачі отримують більше гнучкості та можливостей для впровадження інновацій порівняно з комерційною системою баз даних;

- висока масштабованість: PostgreSQL може легко керувати великим обсягом даних;

— гнучкість: PostgreSQL сумісна з низкою найважливіших мов програмування і протоколів, користувачі зможуть працювати тією мовою, яку вони знають краще, без ризику виникнення системних конфліктів [12].

3.1.6 Вибір ORM

Sequelize – це Object-Relational Mapping - об'єктно-реляційне відображення/перетворення (ORM) для роботи з такими СУБД, як SQLite, MSSQL, Postgres, MySQL, MariaDB. ORM зручні тим, що дозволяють взаємодіяти з БД мовою програми (JavaScript), тобто без використання спеціально призначених для цього мов (SQL). Sequelize дозволяє програмістам писати менше коду, також це хороший інструментарій для міграцій. А так як Sequelize може працювати з різними SQL-базами даних, для визначення, з якою саме базою даних ми працюємо в даний момент, нам необхідно також встановити драйвер “pg” для роботи з PostgreSQL. Цей драйвер надає правила, за допомогою яких будуються валідні запити до БД.

ORM часто використовуються для роботи з СУБД. Серед переваг ORM виділяють:

- можливість автоматичного створення SQL-запитів,;
- наявність явного опису схеми БД, представленого в термінах будь-якої мови програмування, яка знаходиться і редагується в одному місці;
- не потрібно створювати нові SQL-запити при перенесенні на іншу систему управління базами даних, оскільки за це відповідає низькорівневий драйвер ORM;
- ORM дає можливість ізолювати код програми від деталей зберігання даних;
- можливість оперувати елементами мови програмування, тобто класами, об'єктами, атрибутами, методами, а не елементами реляційної моделі даних.

3.1.7 Вибір середи тестування

Для тестування було обрано фреймворк Jest. Jest працює з проектами, що використовують TypeScript, Node, React, Angular, Babel, Vue та інше. Jest націлений на роботу із коробки, без додаткових налаштувань, для більшості проектів на JavaScript. Тести Jest легко відстежують великі об'єкти. Jest ізольований: тести виконуються паралельно, запускаючись у власних потоках, задля досягнення максимальної продуктивності. Забезпечуючи тестам унікальний глобальний стан, Jest прагне безпомилково запускати тести у паралельних потоках. Щоб робити це швидко, Jest в першу чергу запускає провалені раніше тести і змінює черговість їх запуску, відштовхуючись від того, як довго виконується кожен тест. Jest добре документований, може бути розширений і вимагає мінімальної настройки, щоб відповідати вимогам розробника [13].

3.1.8 Вибір платформи для розгортання проекту

Docker - це програмна платформа для швидкої розробки, тестування і розгортання додатків. Контейнеризація за допомогою docker дозволяє вирішити багато проблем. Додаток розбивається на компоненти за функціями, які мають індивідуальну упаковку із залежностями, а потім можуть бути розгорнуті на архітектурі, відмінній від стандартної. Ця можливість спрощує оновлення окремих компонентів і масштабування програми в цілому. Перевагами контейнеризації docker є:

- готове середовище для запуску: контейнер разом із додатком одразу містить середовище для роботи. В одному середовищі ми пишемо додаток і тут його тестуємо, щоб переконатися в працездатності. Завдяки цьому можна завантажити готовий контейнер на будь-який сервер і бути впевненим, що програма запуститься нормально. Неважливо, де було написано код, запускатися він буде стабільно завдяки упаковці в контейнер;

— невелика вага: Docker-контейнер часто важить не більше кількох сотень мегабайтів, іноді сильно менше, а віртуальні машини можуть важити кілька гігабайт;

— можливість відкату. Будь-якої миті контейнер можна «скинути», відкотити до початкової версії. Завдяки цьому можна ефективніше тестувати програми та легше виправляти помилки. Також контейнер завжди можна перезавантажити, іноді це теж допомагає виправити помилку;

— ізольованість від довкілля: контейнер не має доступу до інформації на хості. Це робить роботу програми безпечнішою;

— зручні інструменти керування: образи docker зберігаються у зручному docker-реєстрі, звідки можна швидко запускати готові середовища та на їх основі налаштовувати свої. Сам Docker дозволяє управляти контейнерами: запускати, зберігати, редагувати, перезавантажувати.

3.2 Висновки за розділом 3

Для створення системи зберігання та оброблення даних для онлайн шкіл було обрано якісний та сучасний набір інструментів: платформа Node.js, фреймворк Express, мова програмування TypeScript, СУБД PostgreSQL, ORM Sequelize, Jest, Docker, WebStorm.

4 РОЗРОБКА ПРОГРАМИ

4.1 Опис структури файлів

Проект структуровано з використанням всіх правил для розробки API.

Папка `src` містить в собі папки `config`, `constants`, `controllers`, `entities`, `middlewares`, `models`, `routers`, `services`, `test` та файл `index.ts`, який є головним файлом для запуску проекту. В папці `config` зберігається файл `index.ts` (містить в собі конфігураційні дані), в папці `constants` зберігаються файли із константними значеннями. Папка `controllers` містить в собі папки `admin` (містить в собі контролери для адміністративної панелі), `common` (загальні контролери), `main` (контролери для клієнтської частини). Папка `admin` складається з наступних файлів: `Auth.ts` (контролери, які пов'язані з авторизацією), а `Group.ts`, `Homework.ts`, `Material.ts`, `Test.ts`, `User.ts`, `VideoLesson.ts` містять в собі CRUD для сутностей груп, домашніх завдань, навчальних матеріалів, тестів, користувачів, відеоуроків відповідно. Папка `common` містить в собі файл `File.ts`, в якому прописана логіка для завантаження файлів різних форматів. Папка `main` містить в собі файли `Auth.ts` (методи для авторизації), `Homework.ts` (методи для перегляду та відправки домашнього завдання), `Material.ts` (методи для перегляду навчальних матеріалів), `User.ts` (методи для перегляду і оновлення власних даних), `VideoLesson.ts` (методи для перегляду відеоуроків).

Папка `entities` складається з наступних папок: `Auth`, `Group`, `Material`, `Test`, `User`. Папка `Auth` складається з файлів `login.ts` (метод для авторизації користувача), `loginAdmin.ts` (для авторизації адміністратора). Папка `Group` містить в собі файли `createGroup.ts`, `deleteGroup.ts`, `getAllGroups.ts`, `getGroupById.ts`, `updateGroup.ts`, які містять в собі логіку для створення, видалення груп, отримання всіх груп, отримання групи за допомогою ідентифікатора, оновлення груп відповідно. Папка `Material` складається з файлів `createMaterial.ts`, `deleteMaterial.ts`, `getAllMaterials.ts`, `getMaterialById.ts`, `getMaterialsByGroupId.ts`, `updateMaterial.ts` створення, видалення навчальних матеріалів, отримання всіх матеріалів, отримання матеріалу за допомогою ідентифікатора, отримання матеріалів певної

групи, оновлення матеріалів відповідно. Папка Test містить в собі файли createTest.ts, deleteTest.ts, getAllTests.ts, getTestById.ts, updateTest.ts. Папка User містить в собі файли createUser.ts, deleteUser.ts, getAllUsers.ts, getUserById.ts, updateUser.ts.

Папка middlewares складається з файлів checkRole.ts (перевіряє роль користувача), uploadFile.ts (методи для завантаження файлу на сервер), verifyToken.ts (методи для верифікації токена доступу). Папка models містить в собі файли DoneHomework.ts, EmailCredentials.ts, Group.ts, GroupMaterial.ts, Homework.ts, Materials.ts, Test.ts, User.ts, UserProfile.ts, VideoLesson.ts, все це моделі сутностей бази даних.

Папка routers містить в собі файли admin.ts (містить шляхи для адміністративної панелі), main.ts (шляхи для клієнтської частини). Папка services складається з папок Auth, Db та файли Log.ts. (методи для логування), Sendgrid.ts (методи для роботи з поштовим сервісом). В папці Auth знаходиться папка Token з файлом index.ts, який містить в собі методи керування токеном. В папці Db знаходяться файли index.ts, utils.ts, які містять в собі методи для підключення до бази даних. В папці test знаходяться файли, які були розроблені при застосуванні підходу TDD.

4.2 Основні рішення з реалізації компонентів системи

4.2.1 Основні рішення щодо організації даних

Для маніпулювання даними використовується Sequelize, який відповідає за отримання даних з таблиць PostgreSQL. Розберемо структуру таблиць бази даних. В табл. 4.1 описані основні поля таблиці Users. Вона призначена для зберігання ролі користувача та дати реєстрації.

Таблиця 4.1 — Таблиця Users

Назва поля	Тип поля	Опис
ID	UUID	Ключове поле таблиці
role	Enum: teacher, student, admin	Роль користувача у системі

Продовження таблиці 4.1

Назва поля	Тип поля	Опис
created_at	Date	Дата створення
updated_at	Date	Дата оновлення

В табл. 4.2 описані основні поля таблиці Email Credentials. Вона необхідна для зберігання email-у та пароля.

Таблиця 4.2 — Email Credentials

Назва поля	Тип поля	Опис поля
ID	UUID	Ключове поле таблиці
user_id	UUID	FK User
email	Varchar	Email користувача
password	Varchar	Захешований пароль користувача
created_at	Date	Дата створення
updated_at	Date	Дата оновлення

В табл. 4.3 описані основні поля таблиці User Profile. Таблиця зберігає в собі особисті дані користувача.

Таблиця 4.3 — Таблиця User Profile

Назва поля	Тип поля	Опис
ID	UUID	Ключове поле таблиці
user_id	UUID	FK Users
group_id	UUID	FK Groups
firstName	Varchar	Ім'я користувача
lastName	Varchar	Прізвище користувача
phone	Varchar	Номер телефону користувача
created_at	Date	Дата створення
updated_at	Date	Дата оновлення

В табл. 4.4 описані основні поля таблиці Groups. В цій таблиці зберігаються назва групи та вчитель групи.

Таблиця 4.4 — Таблиця Groups

Назва поля	Тип поля	Опис поля
ID	UUID	Ключове поле таблиці
name	Varchar	Назва групи
teacher_id	UUID	FK Users
created_at	Date	Дата створення
updated_at	Date	Дата оновлення

В табл. 4.5 описані основні поля таблиці Materials. В цій таблиці зберігається інформація щодо навчальних матеріалів: назва навчального матеріалу, посилання на ресурс, який зберігається на нашому сервері.

Таблиця 4.5 — Таблиця Materials

Назва поля	Тип поля	Опис
ID	UUID	Ключове поле таблиці
name	Varchar	Назва матеріалу
file_url	Varchar	Шлях до файлу
file_name	Varchar	Назва файлу
created_at	Date	Дата створення
updated_at	Date	Дата оновлення

В табл. 4.6 описані основні поля сполучної таблиці Groups Materials. В цій таблиці зберігаються зовнішні ключі для реалізації зв'язку багато до багатьох.

Таблиця 4.6 — Таблиця Groups Materials

Назва поля	Тип поля	Опис поля
ID	UUID	Ключове поле таблиці
group_id	UUID	FK Groups
material_id	UUID	FK Materials

В табл. 4.7 описані основні поля таблиці Homeworks. В цій таблиці зберігаються назва домашнього завдання, зовнішні ключі до таблиць: матеріали, тести, відеоуроки та групи.

Таблиця 4.7 — Таблиця Homeworks

Назва поля	Тип поля	Опис поля
ID	UUID	Ключове поле таблиці
name	Varchar	Назва домашнього завдання
group_id	UUID	FK Groups
test_ids	[UUID]	Массив FK Tests
material_ids	[UUID]	Массив FK Materials
video_lesson_id	UUID	FK Video Lessons
created_id	Date	Дата створення
updated_id	Date	Дата оновлення

В табл. 4.8 описані основні поля таблиці Video Lessons. В цій таблиці зберігаються заголовок та опис відеоуроку, посилання на відео, яке зберігається у нас, та назва файлу.

Таблиця 4.8 — Таблиця Video Lessons

Назва поля	Тип поля	Опис поля
ID	UUID	Ключове поле таблиці
title	Varchar	Заголовок відеоуроку
description	Varchar	Опис відеоуроку
file_url	Varchar	Посилання на відеоурок
file_name	Varchar	Назва файлу відеоуроку
created_at	Date	Дата створення
updated_at	Date	Дата оновлення

В табл. 4.9 описані основні поля таблиці Tests. В цій таблиці зберігаються назва тесту, питання з відповідями (та іншими параметрами), час на виконання тесту.

Таблиця 4.9 — Таблиця Tests

Назва поля	Тип поля	Опис поля
ID	UUID	Ключове поле таблиці
name	Varchar	Назва тесту
questions	JSONB	Питання та відповіді тесту

Продовження таблиці 4.9

Назва поля	Тип поля	Опис поля
time	Number	Час на виконання тесту
created_at	Date	Дата створення
updated_at	Date	Дата оновлення

В табл. 4.10 описані основні поля таблиці Done Homeworks. В цій таблиці зберігаються зовнішній ключ на домашні завдання, на виконавця, посилання на файл виконаного домашнього завдання, відповіді на тести виконавця та поточний статус.

Таблиця 4.10 — Таблиця Done Homeworks

Назва поля	Тип поля	Опис поля
ID	UUID	Ключове поле таблиці
user_id	UUID	FK Users
homework_id	Varchar	FK Homeworks
done_homework_file_url	Varchar	Посилання на файл виконаного домашнього завдання
done_homework_file_name	Varchar	Назва файлу з виконаним домашнім завданням
tests_results	JSONB	Результати пройдених тестів
status	Enum: created, inProgress, needReview, reviewed, checked	Статус виконаного домашнього завдання
created_at	Date	Дата створення
updated_at	Date	Дата оновлення

Для повноцінної роботи програми, була розроблена БД, ER діаграма якої зображена на рис 4.1.

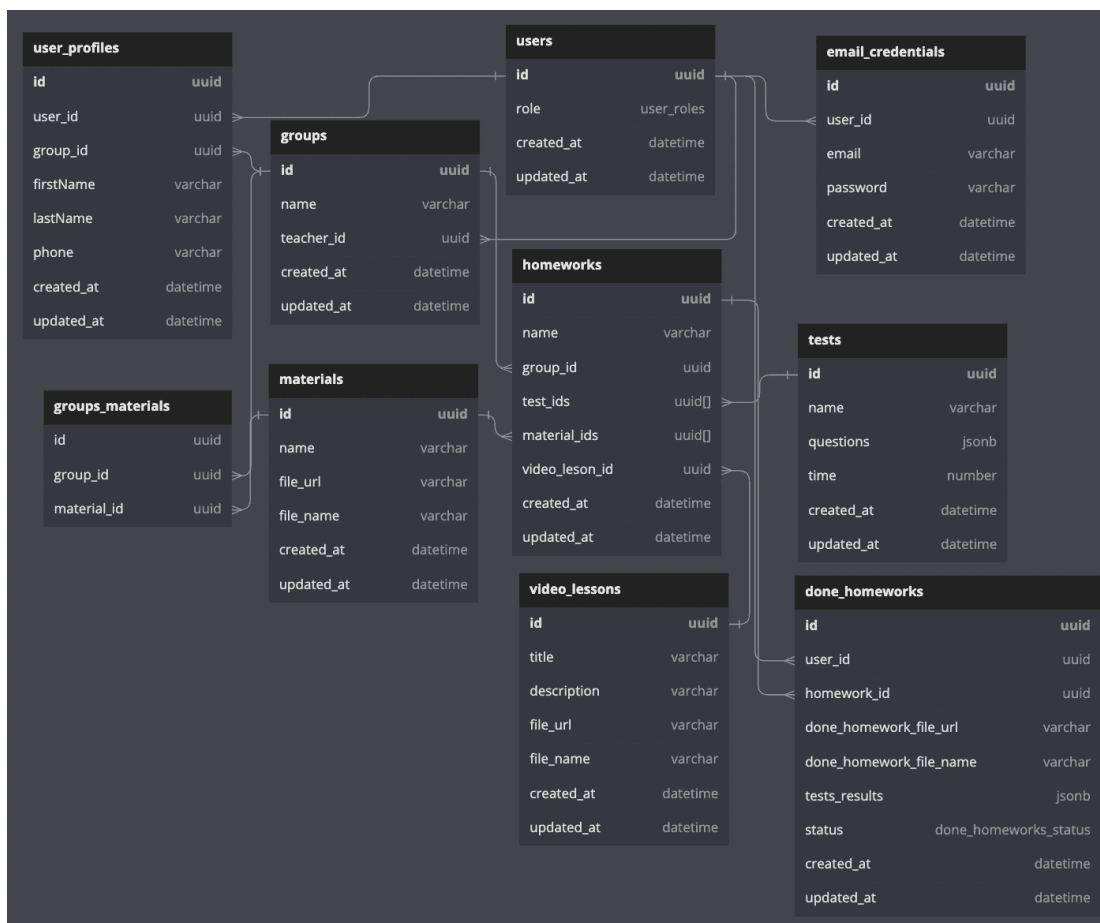


Рисунок 4.1 – ER діаграма БД

Виходячи з рисунку, були створені наступні таблиці, серед яких:

- Users – користувачі;
- Email credentials – дані для авторизації;
- User profile – дані в профілі користувача;
- Groups – список груп;
- Materials – список навчальних матеріалів;
- Groups materials – зовнішні ключі;
- Homeworks – домашні завдання;
- Video lessons – відеоуроки;
- Tests – тести для учнів;
- Done homeworks – виконані домашні завдання.

4.3 Захищення даних

Для захищення даних кожен, хто має доступ до системи, матиме свій логін і пароль. Пароль має бути не менше 6 символів та містити цифри та літери. Пароль хешується після введення даних користувачем при реєстрації, а потім зберігається в базі даних. Для цього було використано бібліотеку `bcryptjs`. Незареєстрований користувач має обмежений доступ до системи. Усі паролі зберігаються у базі даних, до яких матиме доступ тільки адміністратор системи.

4.4 Основні розроблені алгоритми

Основні алгоритми розробленої системи:

- вхід в адміністративну панель з попередньою валідацією (перевірка існування користувача та перевірка хеш-пароля);
- відправка повідомлень на електронну пошту користувача при реєстрації та при відновленні пароля;
- маніпуляції даними за допомогою адміністративної панелі;
- авторизація користувача з валідацією;
- відновлення пароля;
- фільтрація та пошук майже для кожного типу даних;
- пагінація даних та інше.

Наприклад, відправка повідомлень на пошту при відновлення пароля працює наступним чином. Якщо користувач забув пароль та бажає його відновити, вся необхідна для оновлення інформація відправляється на вказану ним пошту. Для реалізації відправки було використано поштовий сервіс `Sendgrid`. Для роботи з цим сервісом було створено акаунт та згенеровано `API key`, який було додано до бекенду. Для того, щоб `Sendgrid` відправляв повідомлення користувачам, необхідні їх електронні адреси та шаблон листа. Лист було попередньо реалізовано. Після того, як користувач отримує повідомлення на пошту, він натискає на відповідну кнопку, потім вводить новий пароль.

Для хешування було обрано бібліотеку `bcryptjs`, яка за допомогою хеш-функції із вхідними параметрами (пароль та `saltRounds`) хешує пароль. `SaltRounds` вважається «вартістю» хеша, тобто це час, який було затрачено на створення хеша; чим вище `saltRounds`, тим довше пароль буде хешуватись і тим важче буде його зламати. Захешований пароль зберігається в базі даних. Це дає гарантію того, що при зломі бази даних, зловмисники не зможуть отримати доступ до акаунтів користувачів. А коли при авторизації користувач вводить адресу пошти та пароль, сервісом перевіряється наявність поштової адреси користувача в базі даних (при існуванні виконується функція `compare`, яка приймає пароль, введений при авторизації, та хеш-пароль, який було попередньо збережено в базі даних). І у випадку, якщо паролі збігаються, користувачу повертається токен для подальшої роботи.

4.5 Опис допоміжних бібліотек, модулів та функцій

Окрім вже перелічених основних фреймворків та бібліотек, для розробки були використані додаткові інструменти, такі як `cors`, `joí`, `bunyan`, `multer`.

Cross-Origin Resource Sharing (CORS) - механізм, що використовує додаткові HTTP-заголовки, щоб дати можливість агенту користувача отримувати дозволи на доступ до вибраних ресурсів із сервера на джерелі (домені), відмінному від того, що сайт використовує на даний момент. Існує `cors` пакет від `node.js` для надання проміжного програмного забезпечення `Connect/Express`, яке можна використовувати для ввімкнення CORS із різними параметрами. Отже, `cors`, доступний в реєстрі `npm`, використовується для усунення помилок CORS в додатку `Node.js`.

Наступний пакет, який було також використано при розробці – це `Joí`. `Joí` — це `Node`-модуль для перевірки даних, його ми використали для валідації. `Joí` може перевірити дані будь-якого типу: від простих скалярних типів даних (рядки, числа або логічні значення) до складних значень, що складаються з кількох рівнів вкладених об'єктів і масивів. `Joí` також корисний, коли дані надходять із

невідомого джерела, наприклад, через загальнодоступний API, тоді Joi допомагає підтвердити, що дані містяться в потрібному форматі. Перевірка вхідних даних і відповідних дій у разі їх неправильності допомагає зробити додатки надійними та стабільними. Joi можна використовувати як окремий модуль у будь-якому додатку Node [14].

Наведемо приклад використання Joi для написання правил вхідних даних при створенні користувача, який було реалізовано в проєкті:

```
export const CREATE_USER_PAYLOAD = joi.object().keys({
  email: joi.string().trim().lowercase().email().required(),
  password: joi.string().trim().min(6).max(64).required(),
  firstName: joi.string().trim().required(),
  lastName: joi.string().trim().required(),
  phone: joi.string().trim().required(),
  groupId: joi.string().trim().uuid().required(),
});
```

Bunyan — це проста у використанні і продуктивна бібліотека логування JSON для Node. Використовується для ведення журналу JSON для сервісів Node.js. Наведемо приклад коду, реалізованого в проєкті, для ініціалізації логера для подальшого логування дій:

```
import { createLogger } from 'bunyan';
const log = createLogger({
  name: 'courses',
  level: 'trace',
});
export default log;
```

Наступний пакет, який теж було використано при розробці, — Multer. Це middleware для фреймворку Express для обробки multipart/form-data, потрібна насамперед при завантаженні файлів. Наведемо приклад коду, реалізованого в проєкті, для збереження файлу на сервері за допомогою Multer:

```
import multer from 'multer';
const storage = multer.diskStorage({
  destination: (req, file, cb) => {
    cb(null, 'static/');
  },
  filename: (req, file, cb) => {
    cb(null, `${Date.now()}-${file.originalname}`);
  },
});
export default multer({ storage });
```

4.6 Висновки за розділом 4

Було описано структуру файлів, допоміжні бібліотеки, розроблено основні алгоритми програми, які будуть використовуватись для реалізації поставлених задач. Було описано та заповнено всі таблиці, які потрібні для працездатності системи. Згенеровано ER діаграму даних, яка поєднує всі таблиці і реалізує логіку системи.

5 ЕКСПЛУАТАЦІЯ, ТЕСТУВАННЯ ТА ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ПРОГРАМИ

5.1 Призначення й умови застосування програми

Розроблена система призначена для забезпечення гнучкого механізму роботи онлайн школи. Розроблена адміністративна панель дозволяє виконувати маніпуляції з даними студентів, навчальних матеріалів, домашніх завдань, тестами і т.д. За допомогою розробленого сервісу онлайн школа може:

- реєструвати учнів;
- надавати можливість для авторизації користувача;
- надавати можливість для відновлення пароля;
- розподіляти учнів по групах;
- розподіяти користувачів по ролях;
- розміщувати відеозаписи з занять в особистих кабінетах учнів;
- розміщувати файли різних форматів (.pdf, .docx, .pptx та інші) в особистих кабінетах учнів;
- надавати можливість завантаження учнем виконаних завдань (в різних форматах) до особистого кабінету;
- розміщувати тести (з впродюкованими або рандомними питаннями, з обмеженням часу та без, з додатковими коментарями до відповідей та без) в особистих кабінетах учнів;
- надавати можливість розміщувати не тільки типічні тести, а і завдання в більш цікавому форматі, наприклад, зібрати із невпорядкованих частин речення;
- розміщувати розклад занять в особистих кабінетах учнів в залежності від групи;
- надавати можливість для пошуку та фільтрації.

Для звернення до програми потрібно виконати наступні кроки. Із кореневої папки необхідно виконати команду `yarn install` для встановлення необхідних пакетів бекенду. Потім потрібно заповнити даними для налаштувань файл «.env».

Далі необхідно запустити Docker Container, який ініціалізує базу даних. Потім за допомогою консолі запустити API. Для запуску проекту до складу технічних засобів має входити персональна

електронна обчислювальна машина (ПЕОМ), яка виконує роль сервера, який містить:

- оперативну пам'ять об'ємом 16 Гігабайт, не менше;
- HDD 100 Гігабайт, не менше;
- процесор Intel Core i3 1.4Hz, не менше;
- операційну систему Ubuntu 18.xx;
- платформу для розгортання проекту Docker 19.3.x;
- базу даних MongoDB 4.4.x;
- Node 14.x;
- Npm 6.14.x.

5.2 Виняткові ситуації

Були розроблені спеціальні функції для обробки виняткових ситуацій. Наприклад, при авторизації користувач має пройти валідацію, на рис. 5.1 зображено результат роботи кінцевої точки для авторизації. Для перевірки використовуються middleware, які перевіряють валідність тих даних, для перевірки яких не потрібне явне звернення до бази даних. У випадку, якщо валідація middleware пройшла успішно, переходимо до наступного етапу валідації, який знаходиться в сервісній частині програми. Ця валідація пов'язана з базою даних.

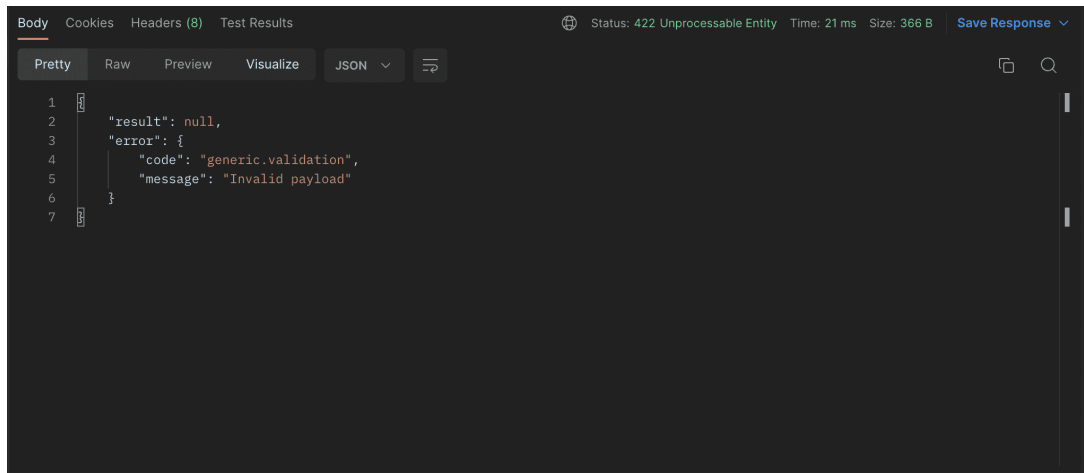


Рисунок 5.1 – Повідомлення про помилки при авторизації

Інший middleware перевіряє наявність та справжність токена (рис. 5.2). У випадку, якщо токен коректний, він розшифровується, та контролеру передається ID користувача для подальшої взаємодії із системою. А якщо він некоректний, то middleware відправляє повідомлення про помилку.

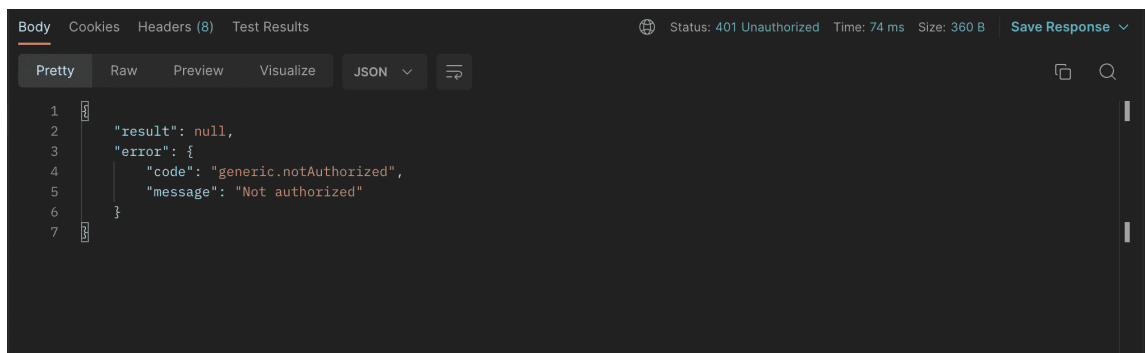


Рисунок 5.2 – Повідомлення про токен

Кінцеві точки, які реалізують функціонал пагінації, мають проміжний обробник, який перевіряє значення параметрів пагінації (поточна сторінка та кількість записів на одній сторінці). Згідно з правилами, прописаними в middleware, поточна сторінка не може бути негативною та не має бути символом в полі запиту. Кількість елементів на одній сторінці має такі ж обмеження, як і значення поточної сторінки, є тільки одна відмінність – кількість елементів на одній сторінці має діапазон від 1 до 100 (рис. 5.3). Це убезпечує роботу API у випадку, якщо користувач хоче одразу отримати дуже велику кількість даних, що потім може зламати роботу API.

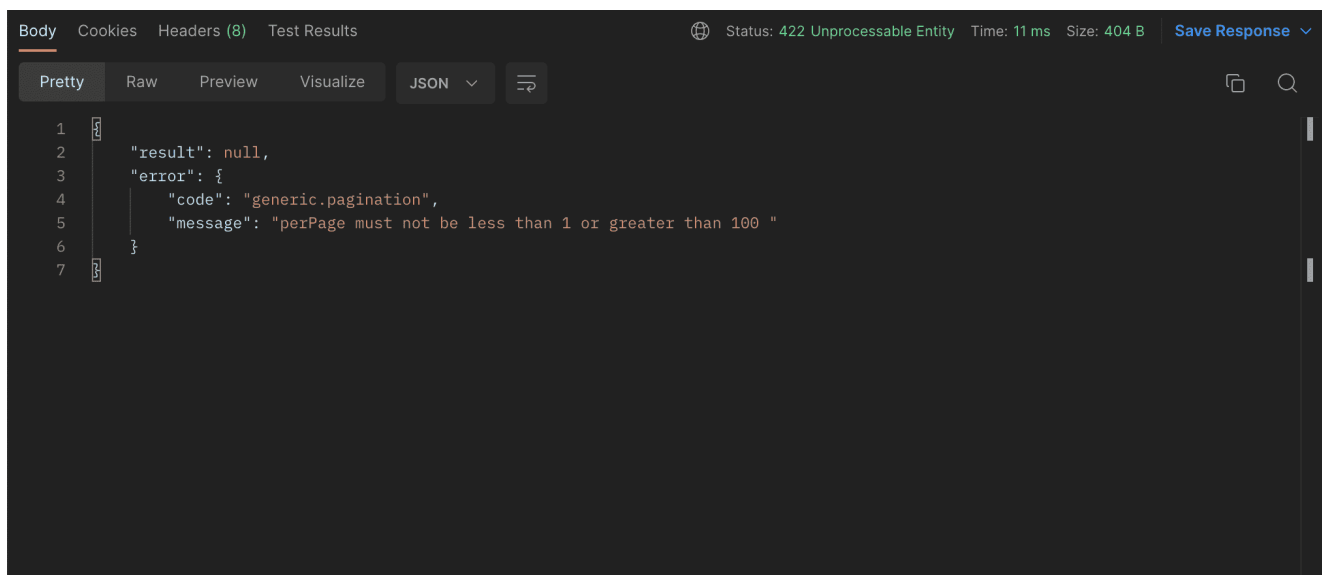


Рисунок 5.3 – Повідомлення про кількість елементів на одній сторінці

5.3 Тестування та експериментальне дослідження програми

Як вже було зазначено в розділі 2.1.2 при розробці проекту використовувався підхід Test Driven Development. Так як перед розробкою кожного блоку коду, писались тести, а після проходження тестів при необхідності в код вносились зміни для його оптимізації, при фінальному тестуванні системи в цілому кількість помилок, багів була мінімальна. Розробка велась у відповідності до циклу TDD. За допомогою фреймворку Jest були реалізовані тести, в яких була описана бізнес-логіка проекту, всі кейси, які може отримати користувач. Потім була створена сама бізнес логіка, також враховуючи можливі кейси, які може отримати користувач. Далі проводився рефакторинг, включаючи запуск тесту, який перевіряє, чи відповідає розроблений ендпоінт кейсам, які розписані в тестах. Якщо тести виконались без помилок, то це означає, що бізнес-логіка була написана коректно, та можна приступати до наступної задачі. А інакше – необхідно було перевірити, які саме кейси виконались з помилками та внести зміни до коду. Таким чином, всі описані дії повторювались для кожного логічного блоку коду. Тому можна зробити висновок, що використання підходу TDD – гарне рішення для мінімізації кількості помилок.

Для тестування програми окрім підходу TDD також було обрано тестування навантаження, як було зазначено в розділі 2.3.

Для тестування навантаження до кожної кінцевої точки було додано метод для збору метрик. Дані метрики передавались в Prometheus. Після цього були написані скрипти, які імітували користувачів. Ми могли задати кількість запитів в хвилину, які будуть опитувати API, імітуючи кількість користувачів в хвилину. Також було встановлено додаток Grafana, який було підключено до Prometheus за допомогою http протоколу. В Grafana було створено layouts, кожний з яких являє собою собою одну метрику (рис. 5.4). Кожний layout за допомогою спеціального запиту звертається до Prometheus задля отримання необхідних даних. Після всіх налаштувань можна було приступити до старту тестування навантаження: необхідно запустити API, Prometheus, Grafana, після чого було запущено скрипт, який імітує користувачів, та відкриваємо Grafana для візуального контролю метрик.

На рис. 5.4 зображено 3 метрики. Перша метрика- Query Count (кількість запитів). Так як платформа Node.js однопоточна, то максимальна кількість запитів в секунду – 10000, отже, щоб впевнитись, що API буде працювати стабільно, запити повинні відпрацьовувати швидко, щоб кількість запитів, які знаходяться в процесі обробки, не перевищувала 10000, інакше сервер почне погано працювати, або зовсім вимкнеться.

Друга метрика – завантаженість CPU (central processing unit, центральний процесор). Тут перевіряється наскільки вічислювальної потужності процесора вистачає на виконання задач. У випадку, якщо процесор буде працювати на максимум, це може призвести до заповільнення проекту. Тим самим необхідно буде оптимізувати роботу кінцевих точок, а якщо оптимізація запитів не вирішила проблему, тоді доведеться збільшувати потужність процесора.

Третя метрика – RAM (random access memory, пам'ять із довільним доступом). Вона відображає обсяг оперативної пам'яті, що використовується. Якщо завантаженість буде занадто високою, то треба діяти так само, як і при проблемах з CPU.

Як можна побачити на рис. 5.4, розроблений API добре справляється з навантаженням, і якщо кількість користувачів не буде перевищувати 10000 в секунду, то API буде працювати стабільно.



Рисунок 5.4 – Метрики

5.4 Висновки за розділом 5

У даному розділі було розглянуто основні призначення та умови застосування програми, були розглянуті виняткові ситуації які можуть виникнути при взаємодії з системою. Також було проведено тестування та експериментальне дослідження програми з використанням TDD підходу та тестування навантаження.

ВИСНОВКИ

В ході виконання кваліфікаційної роботи магістра, було проведено аналіз існуючих систем для онлайн навчання та викладання. Проведений аналіз показав, що подібні існуючі системи мають не весь функціонал, який зазвичай бажають мати онлайн школи. Тому було розроблено сервіси, які усувають головні недоліки існуючих систем, використовуючи сучасні інструменти.

Було складено технічне завдання, в якому зазначена мета розробки програми, призначення, основні вимоги до програми, порядок функціонування системи.

Розроблена система дозволяє виконувати:

- реєстрацію та авторизацію користувачів, відновлення пароля;
- розподіл учнів по групах, а користувачів по ролях;
- розміщення відеозаписів з занять, файлів різних форматів, тестів з різноманітними параметрами, розкладу занять в особистих кабінетах учнів, пошук та фільтрація;
- завантаження учнем виконаних завдань (в різних форматах) до особистого кабінету;
- маніпуляції усіма даними через адміністративну панель та інше.

Система може бути задіяна в будь-якій онлайн школі. Програма повністю дороблена та готова до використання.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Мікросервіси та SOA [Електрон. ресурс]. – Режим доступу:
<https://coderlessons.com/tutorials/veb-razrabotka/arkhitektura-veb-servisov/11-soa-protiv-mikroservisov#9>
2. Мікросервісна архітектура [Електрон. ресурс]. – Режим доступу:
<https://www.atlassian.com/microservices/microservices-architecture>
3. Behavoir Driven Development [Electronic resource]. – Access mode:
<https://www.spiceworks.com/tech/devops/articles/what-is-bdd/>
4. BDD vs TDD [Electronic resource]. – Access mode:
<https://www.fleekitsolutions.com/bdd-vs-tdd-pros-cons/>
5. Test Driven Development [Electronic resource]. – Access mode:
<https://freehost.com.ua/ukr/faq/articles/chto-takoe-tdd-test-driven-development/>
6. Monolithic vs Microservices [Electronic resource]. – Access mode:
<https://seedium.io/blog/monolithic-vs-microservices/>
7. Load Testing [Electronic resource]. – Access mode:
<https://stackify.com/what-is-load-testing/>
8. Prometheus vs Grafana [Electronic resource]. – Access mode:
<https://www.opsramp.com/guides/prometheus-monitoring/prometheus-vs-grafana/>
9. Node.js [Electronic resource]. – Access mode:
<https://artjoker.ua/ru/blog/v-chem-preimushchestva-nodejs/>
10. Express [Electronic resource]. – Access mode:
<https://expressjs.com/uk/>
11. TypeScript [Electronic resource]. – Access mode:
<https://metanit.com/web/typescript/1.1.php>
12. PostgreSQL [Electronic resource]. – Access mode:
<https://azure.microsoft.com/en-gb/resources/cloud-computing-dictionary/what-is-postgresql/>
13. Jest [Electronic resource]. – Access mode:
<https://jestjs.io/>

14. Joi [Electronic resource]. – Access mode:

<https://coderlessons.com/articles/veb-razrabotka-articles/hapi-js-v-deistvii-znakomstvo-s-dzhoi>

ДОДАТОК А
Текст програми

A.1 Файл /src/config/index.ts

```
function getEnv(env: string): string {
  return process.env?.[env] || '';
}

export const PORT = parseInt(getEnv('PORT'));

export const BACKEND = {
  URL: getEnv('BACKEND_URL'),
};

export const DATABASE = {
  USERNAME: getEnv('DATABASE_USERNAME'),
  PASSWORD: getEnv('DATABASE_PASSWORD'),
  DATABASE: getEnv('DATABASE_DATABASE'),
  HOST: getEnv('DATABASE_HOST'),
  PORT: parseInt(getEnv('DATABASE_PORT')),
};

export const SECRET = {
  JWT: getEnv('SECRET_JWT'),
};
```

A.2 Файл /src/constants/Error.ts

```
export const GENERIC_ERRORS = {
  ERROR: 'generic.error',
  VALIDATION: 'generic.validation',
  ACCESS_DENIED: 'generic.accessDenied',
  NOT_AUTHORIZED: 'generic.notAuthorized',
};

export const GROUP_ERRORS = {
  NOT_FOUND: 'group.notFound',
  ALREADY_EXIST: 'group.alreadyExist',
  COUNT_OF_MATERIALS_DOES_NOT_MACH: 'material.countOfMaterialsDoesNotMach',
};

export const EMAIL_CREDENTIAL_ERRORS = {
  NOT_FOUND: 'emailCredential.notFound',
  ALREADY_EXIST: 'emailCredential.alreadyExist',
  PASSWORD_NOT_MATCH: 'emailCredential.passwordNotMatch',
};

export const FILE_ERRORS = {
  CANNOT_UPLOAD: 'file.cannotUpload',
};

export const MATERIAL_ERRORS = {
  NOT_FOUND: 'material.notFound',
  ALREADY_EXIST: 'material.alreadyExist',
  COUNT_OF_GROUP_DOES_NOT_MACH: 'material.countOfGroupDoesNotMach',
};

export const TEST_ERRORS = {
  ALREADY_EXIST: 'test.alreadyExist',
};
```

```
export const USER_ERRORS = {
  NOT_FOUND: 'user.notFound',
};
```

A.3 Файл /src/constants/Validation.ts

```
/**
 * libs
 */
import joi from 'joi';
/**
 * constants
 */
import { GENERIC_ERRORS } from '@constants/Error';
/**
 * models
 */
import { UserRole } from '@models/User';

export const DEFAULT_JOI_RESPONSE = {
  value: null,
  error: {
    code: GENERIC_ERRORS.VALIDATION,
    message: 'Invalid payload',
  },
};

/**
 * auth
 */
export const LOGIN_PAYLOAD = joi.object().keys({
  email: joi.string().trim().lowercase().email().required(),
  password: joi.string().trim().min(6).max(64).required(),
});

/**
 * user
 */
export const CREATE_USER_PAYLOAD = joi.object().keys({
  email: joi.string().trim().lowercase().email().required(),
  password: joi.string().trim().min(6).max(64).required(),
  firstName: joi.string().trim().required(),
  lastName: joi.string().trim().required(),
  phone: joi.string().trim().required(),
  groupId: joi.string().trim().uuid().required(),
});
export const UPDATE_USER_PAYLOAD = joi.object().keys({
  userId: joi.string().trim().uuid().required(),
  email: joi.string().trim().lowercase().email().required(),
  role: joi
    .string()
    .trim()
    .valid(...Object.values(UserRole))
    .required(),
  firstName: joi.string().trim().required(),
  lastName: joi.string().trim().required(),
  phone: joi.string().trim().required(),
  groupId: joi.string().trim().uuid().required(),
});
export const GET_USER_PAYLOAD = joi.object().keys({
  userId: joi.string().trim().uuid().required(),
});
```

```

export const DELETE_USER_PAYLOAD = joi.object().keys({
  userId: joi.string().trim().uuid().required(),
});

/**
 * material
 */
export const GET_MATERIAL_PAYLOAD = joi.object().keys({
  materialId: joi.string().trim().uuid().required(),
});
export const CREATE_MATERIAL_PAYLOAD = joi.object().keys({
  title: joi.string().trim().required(),
  url: joi.string().trim().uri().required(),
  groupIds: joi.array().items(joi.string().trim().uuid().required()),
});
export const UPDATE_MATERIAL_PAYLOAD = joi.object().keys({
  materialId: joi.string().trim().uuid().required(),
  title: joi.string().trim().required(),
  groupIds: joi.array().items(joi.string().trim().uuid().required()),
});
export const DELETE_MATERIAL_PAYLOAD = joi.object().keys({
  materialId: joi.string().trim().uuid().required(),
});
export const GET_MATERIALS_BY_GROUP_ID_PAYLOAD = joi.object().keys({
  groupId: joi.string().trim().uuid().required(),
});

/**
 * group
 */
export const GET_GROUP_PAYLOAD = joi.object().keys({
  groupId: joi.string().trim().uuid().required(),
});
export const CREATE_GROUP_PAYLOAD = joi.object().keys({
  name: joi.string().trim().required(),
  materialIds: joi.array().items(joi.string().trim().uuid().required()),
});
export const UPDATE_GROUP_PAYLOAD = joi.object().keys({
  groupId: joi.string().trim().uuid().required(),
  name: joi.string().trim().required(),
  materialIds: joi.array().items(joi.string().trim().uuid().required()),
});
export const DELETE_GROUP_PAYLOAD = joi.object().keys({
  groupId: joi.string().trim().uuid().required(),
});

/**
 * tests
 */
const questionOptions = joi.object().keys({
  slug: joi.string().trim().lowercase().required(),
  answer: joi.string().trim().required(),
  description: joi.string().trim().required(),
});

const question = joi.object().keys({
  number: joi.number().min(1).required(),
  question: joi.string().trim().required(),
  answer: joi.string().trim().lowercase().required(),
  options: joi.array().items(questionOptions.required()).required(),
});

export const CREATE_TEST_PAYLOAD = joi.object().keys({
  title: joi.string().trim().required(),

```

```

    questions: joi.array().items(question.required()).required(),
  });
};

```

A.3 Файл /src/controllers/admin/Auth.ts

```

/**
 * libs
 */
import { Request, Response } from 'express';
/**
 * constants
 */
import StatusType from '@constants/StatusType';
import { getJsonError, getJsonResponse } from '@constants/ResponseType';
import { DEFAULT_JOI_RESPONSE, LOGIN_PAYLOAD } from '@constants/Validations';
/**
 * entities
 */
import AuthEntity from '@entities/Auth';

export const loginAdmin = async (req: Request, res: Response) => {
  const payload = (req.body && req.body.payload) || null;

  const { error, value } = (payload && LOGIN_PAYLOAD.validate(payload)) ||
    DEFAULT_JOI_RESPONSE;
  if (error || !value || (!error && !value)) {
    return res.status(StatusType.DB_ERROR).json(getJsonError(error));
  }

  const data = await AuthEntity.loginAdmin(value);
  if (data.result) {
    return res.status(StatusType.SUCCESS).json(getJsonResponse(data));
  } else {
    return res.status(StatusType.DB_ERROR).json(getJsonError(data));
  }
};

```

A.4 Файл /src/controllers/admin/Group.ts

```

/**
 * libs
 */
import { Request, Response } from 'express';
/**
 * constants
 */
import StatusType from '@constants/StatusType';
import { getJsonError, getJsonResponse } from '@constants/ResponseType';
import { DEFAULT_JOI_RESPONSE, LOGIN_PAYLOAD } from '@constants/Validations';
/**
 * entities
 */
import AuthEntity from '@entities/Auth';

export const loginAdmin = async (req: Request, res: Response) => {
  const payload = (req.body && req.body.payload) || null;

```

```

    const { error, value } = (payload && LOGIN_PAYLOAD.validate(payload)) ||
    DEFAULT_JOI_RESPONSE;
    if (error || !value || (!error && !value)) {
        return res.status(StatusType.DB_ERROR).json(getJsonError(error));
    }

    const data = await AuthEntity.loginAdmin(value);
    if (data.result) {
        return res.status(StatusType.SUCCESS).json(getJsonResponse(data));
    } else {
        return res.status(StatusType.DB_ERROR).json(getJsonError(data));
    }
};

```

A.5 Файл /src/controllers/admin/Material.ts

```

/**
 * libs
 */
import { Request, Response } from 'express'
/**
 * constants
 */
import StatusType from "@constants/StatusType";
import {getJsonError, getJsonResponse} from "@constants/ResponseType";
import {
    GET_MATERIAL_PAYLOAD,
    CREATE_MATERIAL_PAYLOAD,
    UPDATE_MATERIAL_PAYLOAD,
    DELETE_MATERIAL_PAYLOAD,
    DEFAULT_JOI_RESPONSE,
} from '@constants/Validations'
/**
 * entities
 */
import MaterialEntity from '@entities/Material';

export const createMaterial = async (req: Request, res: Response) => {
    const payload = (req.body && req.body.payload) || null;

    const { error, value } = (payload && CREATE_MATERIAL_PAYLOAD.validate(payload))
    || DEFAULT_JOI_RESPONSE;
    if (error || !value || (!error && !value)) {
        return res.status(StatusType.BAD_REQUEST).json(getJsonError(error));
    }

    const data = await MaterialEntity.createMaterial(value);
    if (data.result) {
        return res.status(StatusType.SUCCESS).json(getJsonResponse(data));
    } else {
        return res.status(StatusType.DB_ERROR).json(getJsonError(data));
    }
}

export const getAllMaterials = async (req: Request, res: Response) => {
    const data = await MaterialEntity.getAllMaterial({});

    if (data.result) {
        return res.status(StatusType.SUCCESS).json(getJsonResponse(data));
    }
}

```

```

    return res.status(StatusType.DB_ERROR).json(getJsonError(data));
  }

export const getMaterialById = async (req: Request, res: Response) => {
  const payload = req.params || null;

  const { error, value } = (payload && GET_MATERIAL_PAYLOAD.validate(payload)) ||
  DEFAULT_JOI_RESPONSE;
  if (error || !value || (!error && !value)) {
    return res.status(StatusType.BAD_REQUEST).json(getJsonError(error));
  }

  const data = await MaterialEntity.getMaterialById(value)
  if (data.result) {
    return res.status(StatusType.SUCCESS).json(getJsonResponse(data));
  }

  return res.status(StatusType.DB_ERROR).json(getJsonError(data));
}

export const updateMaterial = async (req: Request, res: Response) => {
  const params = req.params || null;
  const payload = (req.body && req.body.payload) || null;

  const joiPayload = {
    ...(params || {}),
    ...(payload || {}),
  };

  const { error, value } = (joiPayload &&
  UPDATE_MATERIAL_PAYLOAD.validate(joiPayload)) || DEFAULT_JOI_RESPONSE;
  if (error || !value || (!error && !value)) {
    return res.status(StatusType.BAD_REQUEST).json(getJsonError(error));
  }

  const data = await MaterialEntity.updateMaterial(value);
  if (data.result) {
    return res.status(StatusType.SUCCESS).json(getJsonResponse(data));
  }

  return res.status(StatusType.DB_ERROR).json(getJsonError(data));
}

export const deleteMaterial = async (req: Request, res: Response) => {
  const payload = req.params || null;

  const { error, value } = (payload && DELETE_MATERIAL_PAYLOAD.validate(payload))
  || DEFAULT_JOI_RESPONSE;
  if (error || !value || (!error && !value)) {
    return res.status(StatusType.BAD_REQUEST).json(getJsonError(error));
  }

  const data = await MaterialEntity.deleteMaterial(value);
  if (data.result) {
    return res.status(StatusType.SUCCESS).json(getJsonResponse(data));
  }

  return res.status(StatusType.DB_ERROR).json(getJsonError(data));
}

```

A.6 Файл /src/controllers/admin/User.ts

```

/**
 * libs
 */
import { Request, Response } from 'express';
/**
 * constants
 */
import StatusType from '@constants/StatusType';
import { getJsonError, getJsonResponse } from '@constants/ResponseType';
import {
  CREATE_USER_PAYLOAD,
  DEFAULT_JOI_RESPONSE,
  DELETE_USER_PAYLOAD,
  GET_USER_PAYLOAD,
  UPDATE_USER_PAYLOAD
} from '@constants/Validations';
/**
 * entities
 */
import UserEntity from '@entities/User';
import { UserRole } from '@models/User';

export const createUser = async (req: Request, res: Response) => {
  const payload = (req.body && req.body.payload) || null;

  const { error, value } = (payload && CREATE_USER_PAYLOAD.validate(payload)) ||
  DEFAULT_JOI_RESPONSE;
  if (error || !value || (!error && !value)) {
    return res.status(StatusType.BAD_REQUEST).json(getJsonError(error));
  }

  const data = await UserEntity.createUser({
    ...value,
    role: UserRole.user,
  });

  if (data.result) {
    return res.status(StatusType.SUCCESS).json(getJsonResponse(data));
  }

  return res.status(StatusType.DB_ERROR).json(getJsonError(data));
};

export const getAllUsers = async (req: Request, res: Response) => {
  const data = await UserEntity.getAllUsers({});

  if (data.result) {
    return res.status(StatusType.SUCCESS).json(getJsonResponse(data));
  }

  return res.status(StatusType.DB_ERROR).json(getJsonError(data));
};

export const getUserById = async (req: Request, res: Response) => {
  const payload = req.params || null;

  const { error, value } = (payload && GET_USER_PAYLOAD.validate(payload)) ||
  DEFAULT_JOI_RESPONSE;
  if (error || !value || (!error && !value)) {
    return res.status(StatusType.BAD_REQUEST).json(getJsonError(error));
  }

```

```

    }

    const data = await UserEntity.getUserById(value);
    if (data.result) {
        return res.status(StatusType.SUCCESS).json(getJsonResponse(data));
    }

    return res.status(StatusType.DB_ERROR).json(getJsonError(data));
}

export const updateUser = async (req: Request, res: Response) => {
    const params = req.params || null;
    const payload = (req.body && req.body.payload) || null;

    const joiPayload = {
        ...(params || {}),
        ...(payload || {}),
    };

    const { error, value } = (joiPayload &&
    UPDATE_USER_PAYLOAD.validate(joiPayload)) || DEFAULT_JOI_RESPONSE;
    if (error || !value || (!error && !value)) {
        return res.status(StatusType.BAD_REQUEST).json(getJsonError(error));
    }

    const data = await UserEntity.updateUser(value);
    if (data.result) {
        return res.status(StatusType.SUCCESS).json(getJsonResponse(data));
    } else {
        return res.status(StatusType.DB_ERROR).json(getJsonError(data));
    }
};

export const deleteUser = async (req: Request, res: Response) => {
    const payload = req.params || null;

    const { error, value } = (payload && DELETE_USER_PAYLOAD.validate(payload)) ||
    DEFAULT_JOI_RESPONSE;
    if (error || !value || (!error && !value)) {
        return res.status(StatusType.BAD_REQUEST).json(getJsonError(error));
    }

    const data = await UserEntity.deleteUser(value);
    if (data.result) {
        return res.status(StatusType.SUCCESS).json(getJsonResponse(data));
    } else {
        return res.status(StatusType.DB_ERROR).json(getJsonError(data));
    }
};

```

A.7 Файл /src/controllers/common/File.ts

```

/**
 * libs
 */
import { Request, Response } from 'express';
/**
 * configs
 */
import { BACKEND } from '@config/index';
/**

```

```

    * constants
    */
import { FILE_ERRORS } from '@constants/Error';
import StatusType from '@constants/StatusType';
import { getJsonError, getJsonResponse } from '@constants/ResponseType';

export const uploadFile = async (req: Request, res: Response) => {
  if (req.file) {
    return res.status(StatusType.SUCCESS).json(
      getJsonResponse({
        result: `${BACKEND.URL}/${req.file.path}`,
      })
    );
  } else {
    return res.status(StatusType.DB_ERROR).json(
      getJsonError({
        code: FILE_ERRORS.CANNOT_UPLOAD,
        message: 'Cannot upload file',
      })
    );
  }
};

```

A.8 Файл /src/controllers/main/Auth.ts

```

/**
 * libs
 */
import { Request, Response } from 'express';
/**
 * constants
 */
import StatusType from '@constants/StatusType';
import { getJsonError, getJsonResponse } from '@constants/ResponseType';
import { DEFAULT_JOI_RESPONSE, LOGIN_PAYLOAD } from '@constants/Validations';
/**
 * entities
 */
import AuthEntity from '@entities/Auth';

export const login = async (req: Request, res: Response) => {
  const payload = (req.body && req.body.payload) || null;

  const { error, value } = (payload && LOGIN_PAYLOAD.validate(payload)) ||
    DEFAULT_JOI_RESPONSE;
  if (error || !value || (!error && !value)) {
    return res.status(StatusType.DB_ERROR).json(getJsonError(error));
  }

  const data = await AuthEntity.login(value);
  if (data.result) {
    return res.status(StatusType.SUCCESS).json(getJsonResponse(data));
  } else {
    return res.status(StatusType.DB_ERROR).json(getJsonError(data));
  }
};

```

A.9 Файл /src/controllers/main/Material.ts

```
/**
 * libs
 */
import { Request, Response } from 'express';
/**
 * constants
 */
import StatusType from '@constants/StatusType';
import { getJsonError, getJsonResponse } from '@constants/ResponseType';
import { DEFAULT_JOI_RESPONSE, GET_MATERIALS_BY_GROUP_ID_PAYLOAD } from '@constants/Validations';
/**
 * entities
 */
import MaterialEntity from '@entities/Material';

export const getAllMaterials = async (req: Request, res: Response) => {
  const payload = {
    groupId: req.body.user.profile.group.id || null,
  };

  const { error, value } = (payload &&
GET_MATERIALS_BY_GROUP_ID_PAYLOAD.validate(payload)) || DEFAULT_JOI_RESPONSE;
  if (error || !value || (!error && !value)) {
    return res.status(StatusType.BAD_REQUEST).json(getJsonError(error));
  }

  const data = await MaterialEntity.getMaterialsByGroupId(value);
  if (data.result) {
    return res.status(StatusType.SUCCESS).json(getJsonResponse(data));
  }

  return res.status(StatusType.DB_ERROR).json(getJsonError(data));
};
```

A.10 Файл /src/controllers/main/User.ts

```
/**
 * libs
 */
import { Request, Response } from 'express';
/**
 * constants
 */
import StatusType from '@constants/StatusType';
import { GENERIC_ERRORS } from '@constants/Error';
import { getJsonError, getJsonResponse } from '@constants/ResponseType';

export const me = async (req: Request, res: Response) => {
  const { user } = req.body;

  if (user) {
    return res.status(StatusType.SUCCESS).json(getJsonResponse({ result: { user } }));
  }
};
```

```

return res.status(StatusType.DB_ERROR).json(
  getJsonError({
    code: GENERIC_ERRORS.NOT_AUTHORIZED,
    message: 'Cannot get user',
  })),
);
};

```

A.11 Файл /src/entities/Auth/login.ts

```

/**
 * libs
 */
import bcrypt from 'bcryptjs';
import jwt from 'jsonwebtoken';
/**
 * configs
 */
import { SECRET } from '@config/index';
/**
 * constants
 */
import { EntityReturn } from '@constants/Common';
import { EMAIL_CREDENTIAL_ERRORS, GENERIC_ERRORS } from '@constants/Error';
/**
 * models
 */
import { UserRole } from '@models/User';
import { EmailCredential, User } from '@models/index';
/**
 * services
 */
import Log from '@services/Log';
import { Transaction, doInTransaction } from '@services/Db';

const log = Log.child({ entity: 'login' });

export default async function login({
  email,
  password,
  transaction: tx,
}: {
  email: string;
  password: string;
  transaction?: Transaction;
}): Promise<EntityReturn> {
  return doInTransaction(async (transaction) => {
    try {
      log.trace('Start');

      log.trace(`Find email credential by email '${email}'`);

      const emailCredential = await
EmailCredential.scope('withPassword').findOne({
        where: { email },
        include: [{ model: User, where: { role: UserRole.user }, required: true
}],
        transaction,
      });

      if (!emailCredential) {
        log.error(`Email credential '${email}' not exist`);

```

```

    await transaction.rollback();

    return {
      code: EMAIL_CREDENTIAL_ERRORS.NOT_FOUND,
      message: 'Incorrect email or password',
    };
  }

  log.trace(`Email credential '${email}' exist`);

  log.trace('Does the password match');

  const isMatch = bcrypt.compareSync(password, emailCredential.password);

  if (!isMatch) {
    log.error('Password not match');

    await transaction.rollback();

    return {
      code: EMAIL_CREDENTIAL_ERRORS.PASSWORD_NOT_MATCH,
      message: 'Password not match',
    };
  }

  log.trace('Password matched');

  log.trace('Finish');

  await transaction.commit();

  return {
    result: {
      accessToken: jwt.sign({ userId: emailCredential.userId }, SECRET.JWT),
    },
  };
} catch (error: any) {
  log.fatal('Something is wrong', error);

  await transaction.rollback();

  return {
    code: GENERIC_ERRORS.ERROR,
    message: 'Something went wrong',
  };
}
}, tx);
}

```

A.12 Файл /src/entities/Auth/loginAdmin.ts

```

/**
 * libs
 */
import bcrypt from 'bcryptjs';
import jwt from 'jsonwebtoken';
/**
 * configs
 */

```

```

import { SECRET } from '@config/index';
/**
 * constants
 */
import { EntityReturn } from '@constants/Common';
import { EMAIL_CREDENTIAL_ERRORS, GENERIC_ERRORS } from '@constants/Error';
/**
 * models
 */
import { UserRole } from '@models/User';
import { EmailCredential, User } from '@models/index';
/**
 * services
 */
import Log from '@services/Log';
import { Transaction, doInTransaction } from '@services/Db';

const log = Log.child({ entity: 'loginAdmin' });

export default async function loginAdmin({
  email,
  password,
  transaction: tx,
}: {
  email: string;
  password: string;
  transaction?: Transaction;
}): Promise<EntityReturn> {
  return doInTransaction(async (transaction) => {
    try {
      log.trace('Start');

      log.trace(`Find email credential by email '${email}'`);

      const emailCredential = await
EmailCredential.scope('withPassword').findOne({
        where: { email },
        include: [{ model: User, where: { role: UserRole.admin }, required: true
}],
        transaction,
      });

      if (!emailCredential) {
        log.error(`Email credential '${email}' not exist`);

        await transaction.rollback();

        return {
          code: EMAIL_CREDENTIAL_ERRORS.NOT_FOUND,
          message: 'Incorrect email or password',
        };
      }

      log.trace(`Email credential '${email}' exist`);

      log.trace('Does the password match');

      const isMatch = bcrypt.compareSync(password, emailCredential.password);

      if (!isMatch) {
        log.error('Password not match');

        await transaction.rollback();

```

```

        return {
            code: EMAIL_CREDENTIAL_ERRORS.PASSWORD_NOT_MATCH,
            message: 'Password not match',
        };
    }

    log.trace('Password matched');

    log.trace('Finish');

    await transaction.commit();

    return {
        result: {
            accessToken: jwt.sign({ userId: emailCredential.userId }, SECRET.JWT),
        },
    };
} catch (error: any) {
    log.fatal('Something is wrong', error);

    await transaction.rollback();

    return {
        code: GENERIC_ERRORS.ERROR,
        message: 'Something went wrong',
    };
}
}, tx);
}

```

A.13 Файл /src/entities/Group/createGroup.ts

```

/**
 * constants
 */
import { EntityReturn } from '@constants/Common';
import { GENERIC_ERRORS, GROUP_ERRORS } from '@constants/Error';
/**
 * models
 */
import { Group, GroupMaterial, Materials } from '@models/index';
/**
 * services
 */
import Log from '@services/Log';
import { Transaction, doInTransaction } from '@services/Db';

const log = Log.child({ entity: 'createGroup' });

export default async function createGroup({
    name,
    materialIds,
    transaction: tx,
}: {
    name: string;
    materialIds?: string[];
    transaction?: Transaction;
}): Promise<EntityReturn> {
    return doInTransaction(async (transaction) => {
        try {
            log.trace('Start');

```

```

log.trace(`Find group by name '${name}'`);
const existGroup = await Group.count({ where: { name }, transaction });

if (existGroup) {
  log.error(`Group named '${name}' already exists`);

  await transaction.rollback();

  return {
    code: GROUP_ERRORS.ALREADY_EXIST,
    message: 'Group already exist',
  };
}

log.trace(`Group named '${name}' does not exist`);

const group = await Group.create({ name }, { transaction });

if (materialIds) {
  const materialCount = await Materials.count({ where: { id: materialIds },
transaction });

  if (materialCount !== materialIds.length) {
    log.error(`Count of materials does not mach: materialIds.length =
${materialIds.length}, materialCount = ${materialCount}`);

    await transaction.rollback();

    return {
      code: GROUP_ERRORS.COUNT_OF_MATERIALS_DOES_NOT_MACH,
      message: 'Count of material does not mach'
    }
  }

  await GroupMaterial.bulkCreate(
    [
      ...materialIds.map((materialId) => ({
        materialId,
        groupId: group.id,
      })),
    ],
    { transaction },
  );
}

log.trace(`Group named '${name}' successfully created`);

const createGroup = await Group.findPk(group.id, {
  include: [{ model: GroupMaterial, include: [{ model: Materials }]}],
  transaction,
})

log.trace('Finish');

await transaction.commit();

return { result: createGroup };
} catch (error: any) {
  log.fatal('Something is wrong', error);

  await transaction.rollback();

  return {

```

```

        code: GENERIC_ERRORS.ERROR,
        message: 'Cannot create group',
    };
    }
}, tx);
}

```

A.14 Файл /src/entities/Group/deleteGroup.ts

```

/**
 * constants
 */
import { EntityReturn } from '@constants/Common';
import { GENERIC_ERRORS, GROUP_ERRORS } from '@constants/Error';
/**
 * models
 */
import { Group } from '@models/index';
/**
 * services
 */
import Log from '@services/Log';
import { Transaction, doInTransaction } from '@services/Db';

const log = Log.child({ entity: 'deleteGroup' });

export default async function deleteGroup({
  groupId,
  transaction: tx,
}: {
  groupId: string;
  transaction?: Transaction;
}): Promise<EntityReturn> {
  return doInTransaction(async (transaction) => {
    try {
      log.trace('Start');

      log.trace(`Find group by id = ${groupId}`);

      const existGroup = await Group.findByPk(groupId, { transaction });
      if (!existGroup) {
        log.error(`Group by id = ${groupId} not found`);

        await transaction.rollback();

        return {
          code: GROUP_ERRORS.NOT_FOUND,
          message: 'Group not found',
        };
      }

      log.trace(`Group by id = ${groupId} does exists`);

      log.trace(`Deleting group by id = ${groupId}`);

      await existGroup.destroy({ transaction });

      log.trace(`Group by id = ${groupId} successfully deleted`);

      log.trace('Finish');
    }
  });
}

```

```

    await transaction.commit();

    return {
      result: 'ok',
    };
  } catch (error: any) {
    log.fatal('Something is wrong', error);

    await transaction.rollback();

    return {
      code: GENERIC_ERRORS.ERROR,
      message: 'Cannot delete group',
    };
  }
}, tx);
}

```

A.15 Файл /src/entities/Group/getAllGroups.ts

```

/**
 * constants
 */
import { EntityReturn } from '@constants/Common';
import { GENERIC_ERRORS } from '@constants/Error';
/**
 * models
 */
import { Group, GroupMaterial, Materials } from '@models/index';
/**
 * services
 */
import Log from '@services/Log';
import { Transaction, doInTransaction } from '@services/Db';

const log = Log.child({ entity: 'getAllGroups' });

export default async function getAllGroups({ transaction: tx }: { transaction?: Transaction }): Promise<EntityReturn> {
  return doInTransaction(async (transaction) => {
    try {
      log.trace('Start');

      log.trace('Find all groups');

      const groups = await Group.findAll({ include: [{ model: GroupMaterial, include: [{ model: Materials }] }], transaction });

      log.trace('Finish');

      await transaction.commit();

      return {
        result: groups,
      };
    } catch (error: any) {
      log.fatal('Something is wrong', error);

      await transaction.rollback();
    }
  });
}

```

```

        return {
            code: GENERIC_ERRORS.ERROR,
            message: 'Cannot get groups list',
        };
    }
}, tx);
}

```

A.16 Файл /src/entities/Group/getGroupById.ts

```

/**
 * constants
 */
import { EntityReturn } from '@constants/Common';
import { GENERIC_ERRORS, GROUP_ERRORS } from '@constants/Error';
/**
 * models
 */
import { Group, GroupMaterial, Materials } from "@models/index";
/**
 * services
 */
import Log from '@services/Log';
import { Transaction, doInTransaction } from '@services/Db';

const log = Log.child({ entity: 'getGroupById' });

export default async function getGroupById({
    groupId,
    transaction: tx,
}: {
    groupId: string;
    transaction?: Transaction;
}): Promise<EntityReturn> {
    return doInTransaction(async (transaction) => {
        try {
            log.trace('Start');

            log.trace(`Find group by id = ${groupId}`);

            const group = await Group.findByPk(groupId, { include: [{ model:
GroupMaterial, include: [{ model: Materials }] }], transaction });
            if (!group) {
                log.error(`Group by id = ${groupId} not found`);

                await transaction.rollback();

                return {
                    code: GROUP_ERRORS.NOT_FOUND,
                    message: 'User not found',
                };
            }

            log.trace(`Group by id = ${groupId} exists`);

            log.trace('Finish');

            await transaction.commit();

            return { result: group };
        } catch (error: any) {

```

```

    log.fatal('Something is wrong', error);

    await transaction.rollback();

    return {
      code: GENERIC_ERRORS.ERROR,
      message: 'Cannot found user',
    };
  }
}, tx);
}

```

A.17 Файл /src/entities/Group/updateGroup.ts

```

/**
 * constants
 */
import { EntityReturn } from '@constants/Common';
import { GENERIC_ERRORS, GROUP_ERRORS } from '@constants/Error';
/**
 * models
 */
import { Group, GroupMaterial, Materials } from '@models/index';
/**
 * services
 */
import Log from '@services/Log';
import { Transaction, doInTransaction, Op } from '@services/Db';

const log = Log.child({ entity: 'updateGroup' });

export default async function updateGroup({
  groupId,
  name,
  transaction: tx,
}): {
  groupId: string;
  name: string;
  transaction?: Transaction;
}: Promise<EntityReturn> {
  return doInTransaction(async (transaction) => {
    try {
      log.trace('Start');

      log.trace(`Find group by id = ${groupId}`);

      const group = await Group.findByPk(groupId, { transaction });
      if (!group) {
        log.error(`Group by id = ${groupId} does not exists`);

        await transaction.rollback();

        return {
          code: GROUP_ERRORS.NOT_FOUND,
          message: 'Group not found',
        };
      }

      log.trace(`Group by id = ${groupId} does exist`);

      log.trace(`Find group by name '${name}'`);
    }
  });
}

```

```

    const existGroup = await Group.count({ where: { name, id: { [Op.not]:
groupId } }, transaction });
    if (existGroup) {
      log.error(`Group named '${name}' already exists`);

      await transaction.rollback();

      return {
        code: GROUP_ERRORS.ALREADY_EXIST,
        message: 'Group already exist',
      };
    }

    log.trace(`Group named '${name}' does not exist`);

    await group.update(
      { name },
      {
        where: { id: groupId },
        transaction,
      },
    );

    log.trace(`Group by id = ${groupId} successfully updated`);

    const updatedGroup = await Group.findByPk(group.id, {
      include: [{ model: GroupMaterial, include: [{ model: Materials }]}],
      transaction,
    });

    log.trace('Finish');

    await transaction.commit();

    return { result: updatedGroup };
  } catch (error: any) {
    log.fatal('Something is wrong', error);

    await transaction.rollback();

    return {
      code: GENERIC_ERRORS.ERROR,
      message: 'Cannot update group',
    };
  }
}, tx);
}

```

A.18 Файл /src/entities/Material/createMaterial.ts

```

/**
 * constants
 */
import { EntityReturn } from '@constants/Common';
import { GENERIC_ERRORS, MATERIAL_ERRORS } from '@constants/Error';
/**
 * models
 */
import { Group, GroupMaterial, Materials } from '@models/index';

```

```

/**
 * services
 */
import Log from '@services/Log';
import { Transaction, doInTransaction } from '@services/Db';

const log = Log.child({ entity: 'createMaterial' });

export default async function createMaterial({
  title,
  url,
  groupIds,
  transaction: tx,
}): {
  title: string;
  url: string;
  groupIds?: string[];
  transaction?: Transaction;
}): Promise<EntityReturn> {
  return doInTransaction(async (transaction) => {
    try {
      log.trace('Start');

      log.trace(`Find material by title '${title}'`);

      const existMaterial = await Materials.count({ where: { title }, transaction
    });
      if (existMaterial) {
        log.error(`Material '${title}' already exist`);

        await transaction.rollback();

        return {
          code: MATERIAL_ERRORS.ALREADY_EXIST,
          message: 'Material already exist',
        };
      }

      log.trace(`Material '${title}' does not exist`);

      const material = await Materials.create(
        {
          title,
          url,
          fileName: url.split('/')[url.split('/').length - 1],
        },
        { transaction },
      );

      if (groupIds) {
        const groupCount = await Group.count({ where: { id: groupIds },
transaction });

        if (groupCount !== groupIds.length) {
          log.error(`Count of groups does not mach: groupIds.length =
${groupIds.length}, groupCount = ${groupCount}`);

          await transaction.rollback();

          return {
            code: MATERIAL_ERRORS.COUNT_OF_GROUP_DOES_NOT_MACH,
            message: 'Count of group does not mach',
          };
        }
      }
    }
  });
}

```

```

    await GroupMaterial.bulkCreate(
      [
        ...groupIds.map((groupId) => ({
          groupId,
          materialId: material.id,
        })),
      ],
      { transaction },
    );
  }

  log.trace(`Material '${title}' successfully created`);

  const createdMaterial = await Materials.findPk(material.id, {
    include: [{ model: GroupMaterial, include: [{ model: Group }] }],
    transaction,
  })

  log.trace('Finish');

  await transaction.commit();

  return {
    result: createdMaterial
  };
} catch (error: any) {
  log.fatal('Something is wrong', error);

  await transaction.rollback();

  return {
    code: GENERIC_ERRORS.ERROR,
    message: 'Cannot create material',
  };
}
}, tx);
}

```

A.19 Файл /src/entities/Material/deleteMaterial.ts

```

/**
 * libs
 */
import fs from 'fs';
import path from 'path';
/**
 * constants
 */
import { EntityReturn } from '@constants/Common';
import { GENERIC_ERRORS, MATERIAL_ERRORS } from '@constants/Error';
/**
 *
 */
import { Materials } from '@models/index';
/**
 * services
 */
import Log from '@services/Log';
import { Transaction, doInTransaction } from '@services/Db';

```

```

const log = Log.child({ entity: 'deleteMaterial' });

export default async function deleteMaterial({
  materialId,
  transaction: tx,
}): {
  materialId: string;
  transaction?: Transaction;
}): Promise<EntityReturn> {
  return doInTransaction(async (transaction) => {
    try {
      log.trace('Start');

      log.trace(`Find material by id = ${materialId}`);

      const existMaterial = await Materials.findByPk(materialId, { transaction });
      if (!existMaterial) {
        log.error(`Material by id = ${materialId} not found`);

        await transaction.rollback();

        return {
          code: MATERIAL_ERRORS.NOT_FOUND,
          message: 'Material not found',
        };
      }

      log.trace(`Material by id = ${materialId} does exists`);

      log.trace(`Deleting material by id = ${materialId}`);

      await fs.unlinkSync(path.join(__dirname,
        `../../../../../static/${existMaterial.fileName}`));

      await existMaterial.destroy({ transaction });

      log.trace(`Material by id = ${materialId} successfully deleted`);

      log.trace('Finish');

      await transaction.commit();

      return {
        result: 'ok',
      };
    } catch (error: any) {
      log.fatal('Something is wrong', error);

      await transaction.rollback();

      return {
        code: GENERIC_ERRORS.ERROR,
        message: 'Cannot delete material',
      };
    }
  }, tx);
}

```

A.20 Файл /src/entities/Material/getAllMaterials.ts

```

/**
 * constants
 */
import { EntityReturn } from '@constants/Common';
import { GENERIC_ERRORS } from '@constants/Error';
/**
 * models
 */
import { Group, GroupMaterial, Materials } from '@models/index';
/**
 * services
 */
import Log from '@services/Log';
import { Transaction, doInTransaction } from '@services/Db';

const log = Log.child({ entity: 'getAllMaterial' });

export default async function getAllMaterial({
  transaction: tx,
}: {
  transaction?: Transaction;
}): Promise<EntityReturn> {
  return doInTransaction(async (transaction) => {
    try {
      log.trace('Start');

      log.trace('Find all materials');

      const materials = await Materials.findAll({
        include: [{model: GroupMaterial, include: [{ model: Group }]}],
        transaction,
      });

      log.trace('Finish');

      await transaction.commit();

      return { result: materials };
    } catch (error: any) {
      log.fatal('Something is wrong', error);

      await transaction.rollback();

      return {
        code: GENERIC_ERRORS.ERROR,
        message: 'Cannot get material list',
      };
    }
  }, tx);
}

```

A.21 Файл /src/entities/Material/getMaterialById.ts

```

/**
 * constants
 */
import { EntityReturn } from '@constants/Common';

```

```

import {GENERIC_ERRORS, MATERIAL_ERRORS} from "@constants/Error";
/**
 * models
 */
import {Group, GroupMaterial, Materials} from "@models/index";
/**
 * services
 */
import Log from '@services/Log';
import { doInTransaction, Transaction } from "@services/Db";

const log = Log.child({ entity: 'getMaterialById' });

export default async function getMaterialById({
  materialId,
  transaction: tx,
}: {
  materialId: string,
  transaction?: Transaction;
}): Promise<EntityReturn> {
  return doInTransaction(async transaction => {
    try {
      log.trace('Start');

      log.trace(`Find material by id = ${materialId}`);

      const material = await Materials.findByPk(materialId, {
        include: [{model: GroupMaterial, include: [{ model: Group }]}],
        transaction,
      })
      if (!material) {
        log.error(`Material by id = ${materialId} not found`);

        await transaction.rollback();

        return {
          code: MATERIAL_ERRORS.NOT_FOUND,
          message: 'Material not found',
        }
      }

      log.trace(`Material by id = ${materialId} exist`)

      log.trace('Finish');

      await transaction.commit();

      return { result: material };
    } catch (error: any) {
      log.fatal('Something is wrong', error);

      await transaction.rollback();

      return {
        code: GENERIC_ERRORS.ERROR,
        message: 'Cannot found material'
      }
    }
  }, tx)
}

```

A.22 Файл /src/entities/Material/getMaterialsByGroupId.ts

```

/**
 * constants
 */
import { EntityReturn } from '@constants/Common';
import { GENERIC_ERRORS } from '@constants/Error';
/**
 * models
 */
import { GroupMaterial, Materials } from '@models/index';
/**
 * services
 */
import Log from '@services/Log';
import { Transaction, doInTransaction } from '@services/Db';

const log = Log.child({ entity: 'getMaterialByGroupId' });

export default async function getMaterialsByGroupId({
  groupId,
  transaction: tx,
}: {
  groupId: string;
  transaction?: Transaction;
}): Promise<EntityReturn> {
  return doInTransaction(async (transaction) => {
    try {
      log.trace('Start');

      log.trace(`Find all materials by group id = ${groupId}`);

      const materials = await Materials.findAll({
        include: [
          {
            model: GroupMaterial,
            where: { groupId },
          },
        ],
        transaction,
      });

      log.trace('Finish');

      await transaction.commit();

      return { result: materials };
    } catch (error: any) {
      log.fatal('Something is wrong', error);

      await transaction.rollback();

      return {
        code: GENERIC_ERRORS.ERROR,
        message: 'Cannot get material by group id list',
      };
    }
  }, tx);
}

```

A.23 Файл /src/entities/Material/updateMaterial.ts

```

/**
 * constants
 */
import { EntityReturn } from '@constants/Common';
import { GENERIC_ERRORS, MATERIAL_ERRORS } from "@constants/Error";
/**
 * models
 */
import { Group, GroupMaterial, Materials } from "@models/index";
/**
 * services
 */
import Log from '@services/Log';
import { Op, Transaction, doInTransaction } from "@services/Db";

const log = Log.child({ entity: 'updateMaterial' });

export default async function updateMaterial({
  materialId,
  title,
  groupIds,
  transaction: tx,
}: {
  materialId: string
  title: string;
  groupIds?: string[];
  transaction?: Transaction;
}): Promise<EntityReturn> {
  return doInTransaction(async transaction => {
    try {
      log.trace('Start');

      log.trace(`Find material by title = ${title}`);

      const existMaterial = await Materials.count({ where: { title, id: {
[Op.not]: materialId } }, transaction })
      if (existMaterial) {
        log.error(`Material by title = ${title} already exists`)

        await transaction.rollback();

        return {
          code: MATERIAL_ERRORS.ALREADY_EXIST,
          message: 'Material already exist',
        }
      }

      log.trace(`Material by title = ${title} does exist`)

      log.trace(`Find material by id = ${materialId}`)

      const material = await Materials.findByPk(materialId, { transaction })
      if (!material) {
        log.error(`Material by id = ${materialId} not found`)

        await transaction.rollback();

        return {
          code: MATERIAL_ERRORS.NOT_FOUND,
          message: 'Material not found',
        }
      }
    }
  });
}

```

```

    }

    log.trace(`Material by id = ${materialId} founded`)

    const groupCount = await Group.count({ where: { id: groupIds }, transaction
});
    if (groupCount !== groupIds.length) {
        log.error(`Count of groups does not mach: groupIds.length =
${groupIds.length}, groupCount = ${groupCount}`);

        await transaction.rollback();

        return {
            code: MATERIAL_ERRORS.COUNT_OF_GROUP_DOES_NOT_MACH,
            message: 'Count of group does not mach',
        };
    }

    await GroupMaterial.destroy({ where: { materialId: material.id, },
transaction })

    await GroupMaterial.bulkCreate(
        [
            ...groupIds.map((groupId) => ({
                groupId,
                materialId: material.id,
            })),
        ],
        { transaction },
    );

    await material.update({ title }, { transaction })

    log.trace('Finish');

    await transaction.commit();

    return { result: material }
} catch (error: any) {
    log.fatal('Something is wrong', error);

    await transaction.rollback();

    return {
        code: GENERIC_ERRORS.ERROR,
        message: 'Cannot update material',
    };
}
}, tx)
}

```

A.24 Файл /src/entities/Test/createTest.ts

```

/**
 * constants
 */
import { EntityReturn } from '@constants/Common';
import { GENERIC_ERRORS, TEST_ERRORS } from '@constants/Error';
/**
 * models
 */

```

```

import { Test } from '@models/index';
import { Question } from '@models/Test';
/**
 * services
 */
import Log from '@services/Log';
import { Transaction, doInTransaction } from '@services/Db';

const log = Log.child({ entity: 'createTest' });

export default async function createTest({
  title,
  questions,
  transaction: tx,
}): {
  title: string;
  questions: Question;
  transaction?: Transaction;
}): Promise<EntityReturn> {
  return doInTransaction(async (transaction) => {
    try {
      log.trace('Start');

      log.trace(`Find test by title '${title}'`);
      const existTest = await Test.count({ where: { title }, transaction });

      if (existTest) {
        log.error(`Test '${title}' already exist`);

        await transaction.rollback();

        return {
          code: TEST_ERRORS.ALREADY_EXIST,
          message: 'Test already exist',
        };
      }

      log.trace(`Test '${title}' does not exist`);

      const test = await Test.create({ title, questions }, { transaction });

      log.trace(`Test '${title}' successfully created`);

      log.trace('Finish');

      await transaction.commit();

      return { result: test };
    } catch (error: any) {
      log.fatal('Something is wrong', error);

      await transaction.rollback();

      return {
        code: GENERIC_ERRORS.ERROR,
        message: 'Cannot create test',
      };
    }
  }, tx);
}

```

A.25 Файл /src/entities/User/createUser.ts

```

/**
 * libs
 */
import bcrypt from 'bcryptjs';
/**
 * constants
 */
import { EntityReturn } from '@constants/Common';
import { EMAIL_CREDENTIAL_ERRORS, GENERIC_ERRORS, GROUP_ERRORS } from
 '@constants/Error';
/**
 * models
 */
import { EmailCredential, Group, User, UserProfile } from '@models/index';
import { UserRole } from '@models/User';
/**
 * services
 */
import Log from '@services/Log';
import { Transaction, doInTransaction } from '@services/Db';

const log = Log.child({ entity: 'createUser' });

export default async function createUser({
  email,
  password,
  role,
  firstName,
  lastName,
  phone,
  groupId,
  transaction: tx,
}: {
  email: string;
  password: string;
  role: UserRole;
  firstName: string;
  lastName: string;
  phone: string;
  groupId: string;
  transaction?: Transaction;
}): Promise<EntityReturn> {
  return doInTransaction(async (transaction) => {
    try {
      log.trace('Start');

      log.trace(`Find email credential by email '${email}'`);
      const existEmailCredential = await EmailCredential.count({ where: { email },
transaction });

      if (existEmailCredential) {
        log.error(`Email credential '${email}' already exist`);

        await transaction.rollback();

        return {
          code: EMAIL_CREDENTIAL_ERRORS.ALREADY_EXIST,
          message: 'Email already exist',
        };
      }
    }
  });
}

```

```

log.trace(`Email credential '${email}' does not exist`);

log.trace(`Find group by id '${groupId}'`);

const existGroup = await Group.count({ where: { id: groupId }, transaction
});

if (!existGroup) {
  log.error(`Group by id '${groupId}' not found`);

  await transaction.rollback();

  return {
    code: GROUP_ERRORS.NOT_FOUND,
    message: 'Group not found',
  };
}

log.trace(`Group by id '${groupId}' does exists`);

const user = await User.create({ role }, { transaction });
const emailCredential = await EmailCredential.create(
  {
    userId: user.id,
    email,
    password: bcrypt.hashSync(password, 12),
  },
  { transaction },
);
const userProfile = await UserProfile.create(
  {
    userId: user.id,
    groupId,
    firstName,
    lastName,
    phone,
  },
  { transaction },
);

log.trace(`User with email '${email}' successfully created`);

log.trace('Finish');

await transaction.commit();

return {
  result: {
    user,
    emailCredential,
    userProfile,
  },
};
} catch (error: any) {
  log.fatal('Something is wrong', error);

  await transaction.rollback();

  return {
    code: GENERIC_ERRORS.ERROR,
    message: 'Cannot create user',
  };
}
}, tx);

```

```
}
```

A.26 Файл /src/entities/User/deleteUser.ts

```
/**
 * constants
 */
import { EntityReturn } from '@constants/Common';
import { GENERIC_ERRORS, USER_ERRORS } from '@constants/Error';
/**
 * models
 */
import { User } from '@models/index';
/**
 * services
 */
import Log from '@services/Log';
import { Transaction, doInTransaction } from '@services/Db';

const log = Log.child({ entity: 'deleteUser' });

export default async function deleteUser({
  userId,
  transaction: tx,
}: {
  userId: string;
  transaction?: Transaction;
}): Promise<EntityReturn> {
  return doInTransaction(async (transaction) => {
    try {
      log.trace('Start');

      log.trace(`Find user by id=${userId}`);

      const existUser = await User.count({ where: { id: userId }, transaction });
      if (!existUser) {
        log.error(`User by id = ${userId} not found`);

        await transaction.rollback();

        return {
          code: USER_ERRORS.NOT_FOUND,
          message: 'User not found',
        };
      }

      log.trace(`User by id = ${userId} does exists`);

      log.trace(`Deleting user by id = ${userId}`);

      await User.destroy({ where: { id: userId }, transaction });

      log.trace(`User by id = ${userId} successfully deleted`);

      log.trace('Finish');

      await transaction.commit();

      return {
        result: 'ok',
      };
    }
  });
}
```

```

    } catch (error: any) {
      log.fatal('Something is wrong', error);

      await transaction.rollback();

      return {
        code: GENERIC_ERRORS.ERROR,
        message: 'Cannot delete user',
      };
    }
  }, tx);
}

```

A.27 Файл /src/entities/User/getAllUsers.ts

```

/**
 * constants
 */
import { EntityReturn } from '@constants/Common';
import { GENERIC_ERRORS } from '@constants/Error';
/**
 * models
 */
import { EmailCredential, Group, User, UserProfile } from '@models/index';
/**
 * services
 */
import Log from '@services/Log';
import { Transaction, doInTransaction } from '@services/Db';

const log = Log.child({ entity: 'getAllUsers' });

export default async function getAllUsers({ transaction: tx }: { transaction?: Transaction }): Promise<EntityReturn> {
  return doInTransaction(async (transaction) => {
    try {
      log.trace('Start');

      log.trace('Find all users');

      const users = await User.findAll({
        include: [
          { model: EmailCredential },
          {
            model: UserProfile,
            include: [{ model: Group }],
          },
        ],
        transaction,
      });

      log.trace('Finish');

      await transaction.commit();

      return {
        result: users,
      };
    } catch (error: any) {
      log.fatal('Something is wrong', error);
    }
  });
}

```

```

    await transaction.rollback();

    return {
      code: GENERIC_ERRORS.ERROR,
      message: 'Cannot get users list',
    };
  }
}, tx);
}

```

A.28 Файл /src/entities/User/getUserById.ts

```

/**
 * constants
 */
import { EntityReturn } from '@constants/Common';
import { GENERIC_ERRORS, USER_ERRORS } from '@constants/Error';
/**
 * models
 */
import { EmailCredential, Group, User, UserProfile } from '@models/index';
/**
 * services
 */
import Log from '@services/Log';
import { Transaction, doInTransaction } from '@services/Db';

const log = Log.child({ entity: 'getUserById' });

export default async function getUserById({
  userId,
  transaction: tx,
}): {
  userId: string;
  transaction?: Transaction;
}): Promise<EntityReturn> {
  return doInTransaction(async (transaction) => {
    try {
      log.trace('Start');

      log.trace(`Find user by id = ${userId}`);

      const user = await User.findByPk(userId, {
        include: [
          { model: EmailCredential },
          {
            model: UserProfile,
            include: [{ model: Group }],
          },
        ],
        transaction,
      });

      if (!user) {
        log.error(`User by id=${userId} not found`);

        await transaction.rollback();

        return {
          code: USER_ERRORS.NOT_FOUND,

```

```

        message: 'User not found',
    };
}

log.trace(`User by id=${userId} exists`);

log.trace('Finish');

await transaction.commit();

return { result: user };
} catch (error: any) {
    log.fatal('Something is wrong', error);

    await transaction.rollback();

    return {
        code: GENERIC_ERRORS.ERROR,
        message: 'Cannot found user',
    };
}
}, tx);
}

```

A.29 Файл /src/entities/User/updateUser.ts

```

/**
 * constants
 */
import { EntityReturn } from '@constants/Common';
import { EMAIL_CREDENTIAL_ERRORS, GENERIC_ERRORS, GROUP_ERRORS, USER_ERRORS } from '@constants/Error';
/**
 * models
 */
import { EmailCredential, Group, User, UserProfile } from '@models/index';
import { UserRole } from '@models/User';
/**
 * services
 */
import Log from '@services/Log';
import { Op, Transaction, doInTransaction } from '@services/Db';

const log = Log.child({ entity: 'updateUser' });

export default async function updateUser({
    userId,
    email,
    role,
    firstName,
    lastName,
    phone,
    groupId,
    transaction: tx,
}): {
    userId: string;
    email: string;
    role: UserRole;
    firstName: string;
    lastName: string;
    phone: string;

```

```

    groupId: string;
    transaction?: Transaction;
  }): Promise<EntityReturn> {
    return doInTransaction(async (transaction) => {
      try {
        log.trace('Start');

        log.trace(`Find user by id = ${userId}`);

        const existUser = await User.count({ where: { id: userId }, transaction });
        if (!existUser) {
          log.error(`User by id = ${userId} not found`);

          await transaction.rollback();

          return {
            code: USER_ERRORS.NOT_FOUND,
            message: 'User not found',
          };
        }

        log.trace(`User by id = ${userId} does exists`);

        log.trace(`Find email credential '${email}'`);

        const existEmailCredential = await EmailCredential.count({
          where: {
            email,
            userId: { [Op.not]: userId },
          },
          transaction,
        });
        if (existEmailCredential) {
          log.error(`Email credential '${email}' already exist`);

          await transaction.rollback();

          return {
            code: EMAIL_CREDENTIAL_ERRORS.ALREADY_EXIST,
            message: 'Email already exist',
          };
        }

        log.trace(`Email credential '${email}' does not exist`);

        log.trace(`Find group by id '${groupId}'`);

        const existGroup = await Group.count({ where: { id: groupId }, transaction
      });
      if (!existGroup) {
        log.error(`Group by id '${groupId}' not found`);

        await transaction.rollback();

        return {
          code: GROUP_ERRORS.NOT_FOUND,
          message: 'Group not found',
        };
      }

      log.trace(`Group by id '${groupId}' does exists`);

      await User.update({ role }, { where: { id: userId }, transaction });
      await EmailCredential.update({ email }, { where: { userId }, transaction });
    });
  }

```

```

    await UserProfile.update(
      {
        firstName,
        lastName,
        phone,
        groupId,
      },
      {
        where: { userId },
        transaction,
      },
    );

    log.trace(`User by id = ${userId} successfully updated`);

    const updatedUser = await User.findByPk(userId, {
      include: [
        { model: EmailCredential },
        {
          model: UserProfile,
          include: [{ model: Group }],
        },
      ],
      transaction,
    });

    log.trace('Finish');

    await transaction.commit();

    return {
      result: updatedUser,
    };
  } catch (error: any) {
    log.fatal('Something is wrong', error);

    await transaction.rollback();

    return {
      code: GENERIC_ERRORS.ERROR,
      message: 'Cannot update user',
    };
  }
}, tx);
}

```

A.30 Файл /src/middlewares/checkRole.ts

```

import { NextFunction, Request, Response } from 'express';
import StatusType from '@constants/StatusType';
import { GENERIC_ERRORS } from '@constants/Error';
import { getJsonError } from '@constants/ResponseType';
import { UserRole } from '@models/User';

export default function checkRole(role: UserRole) {
  return function (req: Request, res: Response, next: NextFunction) {
    try {
      if (req.body.user.role !== role) {
        return res.status(StatusType.FORBIDDEN).json(
          getJsonError({
            code: GENERIC_ERRORS.ACCESS_DENIED,

```

```

        message: 'Access denied',
      })),
    );
  }

  next();
} catch (error: any) {
  return res.status(StatusType.FORBIDDEN).json(
    getJsonError({
      code: GENERIC_ERRORS.ACCESS_DENIED,
      message: 'Access denied',
    })),
  );
}
};
}

```

A.31 Файл /src/middlewares/verifyToken.ts

```

import jwt from 'jsonwebtoken';
import { NextFunction, Request, Response } from 'express';
import { SECRET } from '@config/index';
import StatusType from '@constants/StatusType';
import { getJsonError } from '@constants/ResponseType';
import { GENERIC_ERRORS } from '@constants/Error';
import { EmailCredential, Group, User, UserProfile } from '@models/index';

export default async function verifyToken(req: Request, res: Response, next:
NextFunction) {
  try {
    const token = req.headers.authorization?.split(' ')[1];

    const decoded: any = jwt.verify(token, SECRET.JWT);

    const user = await User.findByPk(decoded.userId, {
      include: [{ model: UserProfile, include: [{ model: Group }] }, { model:
EmailCredential }],
    });

    if (!user) {
      return res.status(StatusType.UNAUTHORIZED).json(
        getJsonError({
          code: GENERIC_ERRORS.NOT_AUTHORIZED,
          message: 'Not authorized',
        })),
      );
    }

    req.body.user = user;

    next();
  } catch (error: any) {
    return res.status(StatusType.UNAUTHORIZED).json(
      getJsonError({
        code: GENERIC_ERRORS.NOT_AUTHORIZED,
        message: 'Not authorized',
      })),
    );
  }
}

```

Додаток Б
Слайди презентації

Національний університет «Запорізька політехніка»
Кафедра програмних засобів

ПРОЕКТУВАННЯ ТА РОЗРОБЛЕННЯ АРХІТЕКТУРИ СИСТЕМИ ЗБЕРІГАННЯ ТА ОБРОБЛЕННЯ ДАНИХ ДЛЯ ОНЛАЙН ШКІЛ НА ОСНОВІ МІКРОСЕРВІСІВ

ВИКОНАВ
СТ.ГР.КНТ-121М

Соколовський Д.В.

КЕРІВНИК

Табунщик Г.В.

Рисунок Б.1 – Слайд 1

Мета, об'єкт та предмет дослідження

Мета - розробка сервісів для підтримки клієнтських застосувань для онлайн шкіл.

Об'єкт дослідження - сервіси для онлайн навчання.

Предмет дослідження - програмні засоби реалізації сервісів для підтримки клієнтських застосувань для онлайн навчання.

Рисунок Б.2 – Слайд 2

Аналіз аналогів

Функції	Система	
	Ruzuku	Master Test
Створення курсів	так	так
Завантаження файлів різних форматів	так	ні
Перездача тесту	так	ні
Змішаний порядок питань/відповідей	так	ні
Зміна кількості правильних відповідей	так	так
Наявність розкладу занять для учнів	ні	ні
Пошук за назвою на сторінках	ні	ні
Завантаження учнем домашніх завдань (файли різних форматів)	ні	ні

Рисунок Б.3 – Слайд 3

Application Programming Interface

Це інтерфейс взаємодії між сайтом і сторонніми програмами і серверами. Програміст може скористатися API для отримання доступу до функціоналу сторонньої програми.

Використання API скорочує необхідність створювати самостійно складні програми або додатки.

Рисунок Б.4 – Слайд 4

Вибір між TDD та BDD

Test Driven Development	Behavior Driven Development
Визначення	
TDD — це техніка розробки, яка зосереджена на роботі функцій.	BDD — це техніка розробки, спрямована на поведінку системи.
Контриб'ютор	
Розробники	Розробники, QA, клієнти
Фокус на	
Юніт тестуванні	Розумінні вимог

Рисунок Б.5 – Слайд 5

Test Driven Development

- Це техніка розробки програмного забезпечення, яка пропонує повторення коротких циклів розробки: спочатку пишеться тест, що покриває бажану зміну, потім пишеться код, який дозволить пройти тест, і потім проводиться рефакторинг нового коду відповідно до стандартів.



Рисунок Б.6 – Слайд 6

Порівняння мікросервісного та монолітного підходу

Критерії	Мікросервісний додаток	Монолітний додаток
Діапазон технологій	Кожний сервіс може бути написаним на своїй мові програмування, мати власний спосіб зберігання даних, необхідні бібліотеки.	Код монолітного додатку є єдиним цілим, тому вся команда розробників повинна дотримуватись вже обраних інструментів та методів.
Швидкість релізів	Для запуску нових функцій або оновлень вже існуючих, достатньо лише замінити один модуль програми.	Для того, щоб протестувати реліз та підготувати до нього додаток, необхідно оновити повністю всю систему.
Оптимізація додатку	Управління функціональністю, ресурсами та інфраструктурою програми можна довірити різним сервісам та оптимізувати кожен із них окремо.	Необхідно не забувати про постійне враховування внутрішніх зв'язків між його модулями, бо оновлення хоча б одного модуля призведе до зміни системи в цілому.
Процес розробки	Можна вести гнучку розробку та за необхідності швидко змінити склад команди або вимоги до продукту.	Більш вимогливий до порога входження, бо кожному новому розробнику в команді проекту доведеться повністю вивчити код програми та її функціональність.
Відмовостійкість	При наявності проблем всередині одного сервісу, робота системи загалом не порушується, нові помилки не з'являються	Всі елементи безпосередньо або опосередковано пов'язані один з одним, тому збій всередині одного модуля може призвести до повної відмови системи
Масштабованість	При збільшенні навантаження на ресурси мікросервісів, вони автоматично масштабуються, а гнучкий процес розробки дозволяє посилювати команду додатковими працівниками.	При зміні хоча б одного блоку, буде необхідно масштабувати весь додаток.

Рисунок Б.7 – Слайд 7

Порівняння мов програмування

Критерії	Мова					
	C#	Go	Java	JavaScript	PHP	Python
Імперативна мова	+	+	+	+	+	+
Об'єктно-орієнтована мова	+	+/-	+	+	+	+
Функціональна мова	+/-	+/-	+/-	+/-	+/-	+
Статистична типізація	+	+	+	-	-	-
Динамічна типізація	+	-	-	+	+	+
Явна типізація	+	+	+	-	+/-	+/-
Неявна типізація	+/-	-	-	+	+	+
First class functions	+	-	-	+	+	+
Анонімні функції	+	-	+	+	+	+
Лексичні замикання	+	-	+	+	+	+
Часткове застосування	-	-	-	+	-	+
Карирування	+	-	-	+	+	+
Пітримка Unicode в ідентифікаторах	+	+	+	+	+/-	+
Значення параметрів за замовчуванням	+	+	-	+	+	+
Наявність бібліотек для роботи з графікою і мультимедіа	+	-	+	+	+/-	+

Рисунок Б.8 – Слайд 8

Порівняння фреймворків

Критерії	Фреймворк		
	Express	Fastify	Koa
Можливості	Простий, мінімалістичний, гнучкий і масштабований. Швидка розробка програми. Повністю настраюється. Легка інтеграція сторонніх служб.	Блискавча швидкість порівняно з іншими популярними фреймворками Node. Забезпечує повну інкапсуляцію для плагінів, автоматично аналізує JSON із відносно швидшим відтворенням і забезпечує швидку маршрутизацію.	Важить дуже мало (близько 600 рядків коду), що робить його дуже легким. Koa не використовує зворотні виклики, отже полегшує вхідний і низхідний потік елементів керування. Рефакторинг більшої бази коду в Koa легший завдяки модульності.
Документація	Велика кількість документації	Поки що не так багато документації	Середній рівень об'єму документації
Підтримка	Високий рівень підтримки спільноти	Слабка підтримка спільноти для дебагінгу	Низький рівень підтримки спільноти

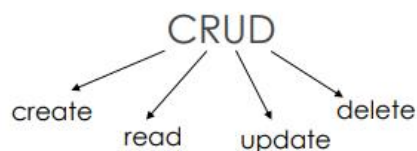
Рисунок Б.9 – Слайд 9

Основні розроблені функції

- ▶ вхід в адміністративну панель з попередньою валідацією (перевірка існування користувача та перевірка хеш-пароля);
- ▶ відправка повідомлень на електронну пошту користувача при реєстрації та при відновленні пароля;
- ▶ маніпуляції даними за допомогою адміністративної панелі;
- ▶ авторизація користувача з валідацією, відновлення пароля;
- ▶ пошук та фільтрація майже в кожному ендпоінті;
- ▶ пагінація даних;
- ▶ ендпоінти для отримання наступних списків: домашні завдання, навчальні файли, розклади та інше.

Рисунок Б.10 – Слайд 10

CRUD



Переваги:

- він спрощує та полегшує проектування додатків, роблячи їх більш масштабованими;
- полегшує контроль безпеки, задовольняючи різні вимоги доступу;
- краща продуктивність.

Рисунок Б.11 – Слайд 11

Таблиці БД

Homeworks

1	Назва поля	Тип поля	Опис поля
2	ID	UUID	Ключове поле таблиці
3	name	Varchar	Назва домашнього завдання
4	group_id	UUID	FK Groups
5	test_ids	[UUID]	Масив FK Tests
6	material_ids	[UUID]	Масив FK Materials
7	video_lesson_id	UUID	FK Video Lessons
8	created_at	Date	Дата створення
9	updated_at	Date	Дата оновлення

Done Homeworks

1	Назва поля	Тип поля	Опис поля
2	ID	UUID	Ключове поле таблиці
3	user_id	UUID	FK Users
4	homework_id	Varchar	FK Homeworks
5	done_homework_file_url	Varchar	Посилання на файл виконаного домашнього завдання
6	done_homework_file_name	Varchar	Назва файлу з виконаним домашнім завданням
7	tests_results	JSONB	Результати пройдених тестів
8	status	Enum: created, inProgress, needReview, reviewed, checked	Статус виконаного домашнього завдання
9	created_at	Date	Дата створення
10	updated_at	Date	Дата оновлення

Рисунок Б.12 – Слайд 12

Діаграма БД

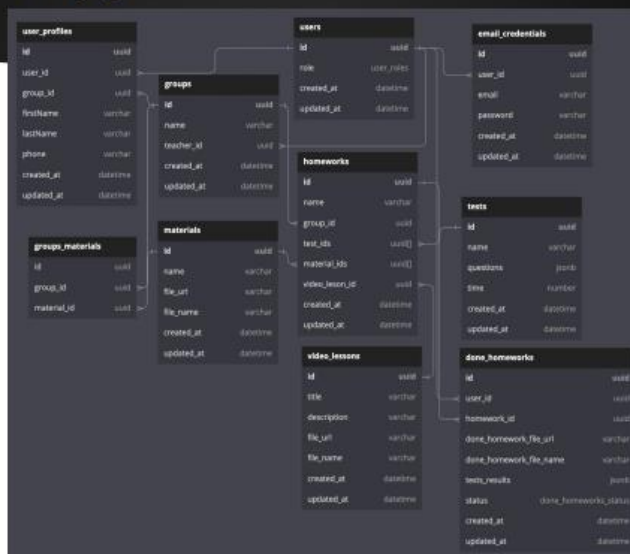


Рисунок Б.13 – Слайд 13

Деякі розроблені алгоритми

- ▶ Відправка повідомлень на пошту: Sendgrid, API key

- ▶ Захищення даних

Процес хешування виконується за допомогою бібліотеки bcryptjs.

```

    bcrypt.hash = function(s, salt, callback, progressCallback) {
        function _async(callback) {
            if (typeof s === 'string' && typeof salt === 'number') {
                bcrypt.genSalt(salt, function(err, salt) {
                    _hash(s, salt, callback, progressCallback);
                });
            } else if (typeof s === 'string' && typeof salt === 'string') {
                _hash(s, salt, callback, progressCallback);
            } else {
                nextTick(callback.bind(this, Error('Illegal arguments: '+(typeof s)+ ' - '+(typeof salt))));
            }
        }

        if (callback) {
            if (typeof callback !== 'function') {
                throw Error('Illegal callback: '+(typeof callback));
            }
            _async(callback);
        } else {
            return new Promise(function(resolve, reject) {
                _async(function(err, res) {
                    if (err) {
                        reject(err);
                    } else {
                        resolve(res);
                    }
                });
            });
        }
    };
  
```

Рисунок Б.14 – Слайд 14

15

Виняткові ситуації

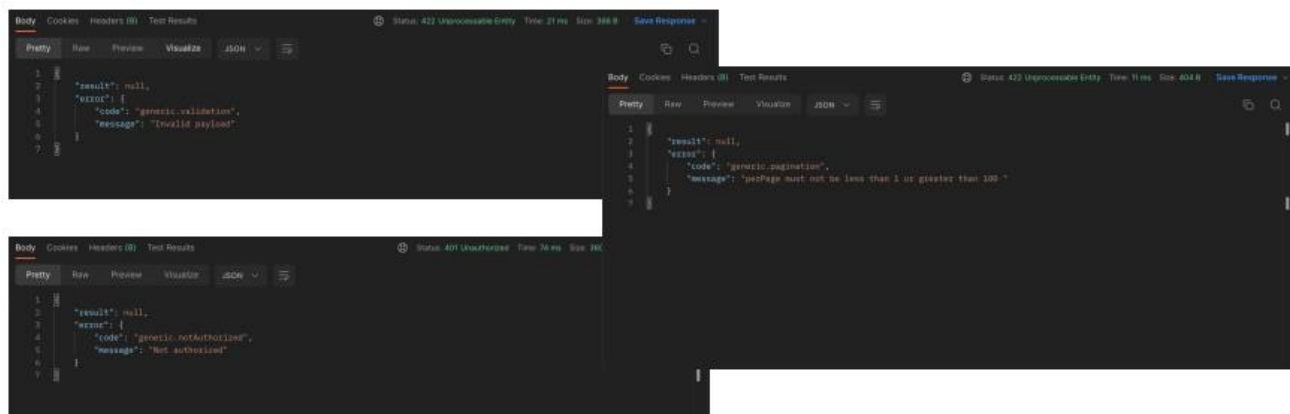


Рисунок Б.15 – Слайд 15

16

Тестування навантаження



Рисунок Б.16 – Слайд 16

Висновки

В ході виконання кваліфікаційної роботи магістра, було проведено аналіз існуючих систем для онлайн навчання та викладання. Проведений аналіз показав, що подібні існуючі системи мають не весь функціонал, який зазвичай бажають мати онлайн школи. Тому було розроблено сервіси, які усувають головні недоліки існуючих систем, використовуючи сучасні інструменти.

Рисунок Б.17 – Слайд 17