НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:
В. о. завідувача кафедри
_____ Михайло НОВОТАРСЬКИЙ
(підпис)
“__” _____2025 р.

Дипломний проєкт

на здобуття ступеня бакалавра
за освітньо-професійною програмою “Інженерія програмного забезпечення комп’ютерних систем”
спеціальності 121 “Інженерія програмного забезпечення”

на тему: Серверна частина веб-застосунку для організації роботи фітнес-клубів.

Виконав : студент __4__ курсу, групи _ІМ-13_
(шифр групи)

Шерстюк Денис Михайлович
(прізвище, ім’я, по батькові) (підпис)

Керівник доцент, к.т.н. Волокита А. М.
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант (нормоконтроль) асистент Пономаренко А. М, _____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2025 р.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)

Освітньо-професійна програма

“Інженерія програмного забезпечення комп’ютерних систем”

спеціальності 121 “Інженерія програмного забезпечення”

ЗАТВЕРДЖУЮ
В. о. завідувача кафедри
_____ Михайло НОВОТАРСЬКИЙ
(підпис)

“__” _____2025 р.

ЗАВДАННЯ
на бакалаврський дипломний проєкт студента

Шерстюка Дениса Михайловича

1. Тема проєкту *Серверна частина веб-застосунку для організації роботи фітнес-клубів.*
керівник проєкту Волокита Артем Миколайович, доцент, к.т.н.,
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від 23.05.2025 №1705-с

2. Термін здачі студентом закінченого проєкту *30 травня 2025 р.*

3. Вихідні дані до проєкту *технічна документація, теоретичні дані.*

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

Розділ 1. Огляд функціоналу та бізнес-логіки існуючих застосунків у сфері управління фітнес-клубами

Розділ 2. Огляд та вибір технологій для розробки уніфікованої платформи.

Розділ 3. Проектування та реалізація платформи.

Розділ 4. Висновки щодо реалізованої платформи.

5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень) структурна схема системи, функціональна схема (діаграма класів), алгоритм дій програмного забезпечення.

6. Консультанта проєкту, з вказівкою розділів проєкту, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Пономернко А. М.		

7. Дата видачі завдання «12» січня 2025 р.

Календарний план

№ п/п	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту	Примітки
1.	*Затвердження теми проєкту*	01.02.2025 – 10.02.2025	
2.	*Вивчення та аналіз завдання*	11.02.2025 – 28.02.2025	
3.	*Розробка архітектури та загальної структури системи*	01.03.2025 – 31.03.2025	
4.	*Розробка структур окремих Підсистем*	01.04.2025 – 30.04.2025	
5.	*Програмна реалізація системи*	01.05.2025 – 15.05.2025	
6.	*Оформлення пояснювальної записки*	20.05.2025 – 28.05.2025	
7.	*Захист програмного продукту*	29.05.2025	
8.	*Передзахист*	30.05.2025	
9.	*Захист*	17.06.2025	

Студент-дипломник _____ Денис ШЕРСТЮК
(підпис)

Керівник проєкту _____ Артем ВОЛОКИТА
(підпис)

АНОТАЦІЯ

У даній роботі проводиться аналіз існуючих застосунків для фітнес-клубів. Визначаються їх переваги та недоліки. На основі дослідження буде обрано ключові функції, що стануть основою для розробки власного рішення. В результаті буде розроблено серверну частину застосунку для фітнес-клубів. Реалізоване рішення буде протестоване за допомогою юніт-тестів та інтеграційного тестування для підтвердження його стабільності та готовності до використання в реальних умовах.

Ключові слова: Серверна частина застосунку, застосунок для фітнес-клубів, TypeScript, NestJS, REST API.

ANNOTATION

In this project for Bachelor’s Degree existing applications for fitness clubs are analized. Their advantages and disadvantages are identified. Based on the study, the key functions will be selected, which will become the basis for the development of own solution. As a result, the server side of the application for fitness clubs will be developed. The implemented solution will be tested using unit tests and integration testing to confirm its stability and readiness for use in real conditions.

Keywords: Server side of the application, application for fitness clubs, TypeScript, NestJS, REST API.

справки	Формат	*Значення*	*Найменування*	Кіл. листів	№ екземпля	*Додаток*
			Документація загальна			
			Знову розроблена			
	А4	***ІАЛЦ.467200.002 ТЗ***	Серверна частина веб-застосунку	3		
			для організації роботи			
			фітнес-клубів			
			Технічне завдання			
	А4	***ІАЛЦ.467200.003 ПЗ***	Серверна частина веб-застосунку	103		
			для організації роботи			
			фітнес-клубів			
			Пояснювальна записка			
	А4	***ІАЛЦ.467200.004 Д1***	Серверна частина веб-застосунку	1		
			для організації роботи			
			фітнес-клубів			
			Структурна схема			
			системи			
	А4	***ІАЛЦ.467200.005 Д2***	Серверна частина веб-застосунку	1		
			для організації роботи			
			фітнес-клубів			
			ER-діаграма таблиць бази даних			
			(функціональна схема)			
	А4	***ІАЛЦ.467200.006 Д3***	Серверна частина веб-застосунку	1		
			для організації роботи			
			фітнес-клубів			
			Алгоритм запису на тренування			
	А4	***ІАЛЦ.467200.007 Д4***	Серверна частина веб-застосунку	50		
			для організації роботи			
			фітнес-клубів			
			Текст програмного коду			

					ІАЛЦ.467200.001 ОА	
Зм	***Лист***	***№ докум.***	***Підп***	***Дата***		
Розроб		Шерстюк Д. М			*Серверна частина веб-застосунку для організації роботи фітнес-клубів.* Опис альбому	Літ. / Аркуш: 1 / Аркушів: 1
Перев		Волокита А.М.				***КПІ ФІОТ ІМ-13***

ТЕХНІЧНЕ ЗАВДАННЯ
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Серверна частина веб-застосунку для організації роботи фітнес-клубів.*»

Київ – 2025

ЗМІСТ

					ІАЛЦ.467200.002 ТЗ		
		№ докум.	Підпис	Дата			
Розробив		Шерстюк Д. М.			Серверна частина веб-застосунку для організації роботи фітнес-клубів Технічне завдання	Літ.	Аркуш 1 / Аркушів 3
Перевірив		Волокита А. М.					
Н. Контр.		Пономаренко А.М.				КПІ ім. Ігоря Сікорського, ФІОТ,ІМ-13	
Затвердив							

1 НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання поширюється на розробку серверної частина веб-застосунку для організації роботи фітнес-клубів.

Областю застосування цієї системи є використання її в фітнес-клубах, студіях для персонального тренування, спортивних секціях або інших закладах, що ведуть облік взаємодії клієнтів з відповідним бізнесом.

2 ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки даної системи є завдання для виконання роботи кваліфікаційно-освітнього рівня «бакалавр інженерії програмного забезпечення», який був затверджений факультетом “Інформатики та обчислювальної техніки” кафедрою обчислювальної техніки Національного технічного Університету України «Київський Політехнічний інститут ім. Ігоря Сікорського».

3 МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Мета розробки полягає в стандартизації та спрощені створення застосунків для організації роботи фітнес-клубів, які забезпечують зручне зберігання та доступ до інформації щодо взаємодії клієнтів з клубом. Призначенням розробки є впровадження більш універсального рішення для реалізації програмних продуктів на будь-яких платформах за допомогою уніфікованої серверної частини відповідних інформаційних систем.

4 ДЖЕРЕЛА РОЗРОБКИ

Джерелом розробки даного дипломного проекту є офіційні документації, публікації та статті в мережі Інтернет на дану тему, науково-технічна література.

5 ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розробленого продукту

Розроблена система має виконувати такі вимоги:

- Легка інтеграція з більшістю можливих платформ, на базі яких може бути створений користувацький інтерфейс
- Підтримка маштабування та розшення функціоналу без докорінних змін архітектури
- Висока можливість до перевикористання коду при розробці інтерфейсу кінцевого користувача на базі платформи

5.2. Вимоги до програмного забезпечення

- ОС Windows, Mac чи Linux.
- IDE WebStorm 2020.1 або новіше.
- Node.js версії 20.x або вище.

5.3. Вимоги до апаратної частини

- ЦП не менше ніж Intel® Core ™ i3-10100.
- RAM не менше ніж 6 ГБ.

6 ЕТАПИ РОЗРОБКИ

Назва етапів виконання	Термін виконання
Затвердження теми роботи	01.02.2025 – 10.02.2025
Вивчення та аналіз завдання	11.02.2025 – 28.02.2025
Розробка архітектури та загальної структури системи	01.03.2025 – 31.03.2025
Розробка структур окремих частин системи	01.04.2025 – 30.04.2025
Програмна реалізація системи	01.05.2025 – 15.05.2025
Виправлення помилок	16.05.2025 – 19.05.2025
Оформлення пояснювальної записки	20.05.2025 – 28.05.2025

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «Серверна частина веб-застосунку для організації роботи фітнес-клубів.»

Київ – 2025

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

API (Application Programming Interface) Інтерфейс програмування додатків

REST (Representational State Transfer) Передача стану представлення

SQL (Structured Query Language) Мова структурованих запитів

NoSQL (Not Only SQL) Не тільки SQL

JSON (JavaScript Object Notation) Нотація об’єктів JavaScript

BSON (Binary JSON) Бінарний JSON

ACID (Atomicity, Consistency, Isolation, Durability) Атомарність, консистентність, ізоляція, довговічність

HTTP (HyperText Transfer Protocol) Протокол передачі гіпертексту

IDE (Integrated Development Environment) Інтегроване середовище розробки

ОС (Operating System) Операційна система

ЦП (Central Processing Unit) Центральний процесор

ROM (Read-Only Memory) Пам’ять лише для читання

RAM (Random Access Memory) Оперативний запам’ятовуючий пристрій

JWT (JSON Web Token) JSON Веб-токен

SMS (Short Message Service) Служба коротких повідомлень

TCP (Transmission Control Protocol) Протокол керування передачею

TTL (Time To Live) Час життя

UI (User Interface) Користувацький інтерфейс

YAML (YAML Ain’t Markup Language) Ще одна мова розмітки

СУБД (Database Management System) Система управління базами даних

ВСТУП

Прискорюються темпи цифрової трансформації сучасного світу. Вона зачіпає всі існуючі індустрії. Однією з них є сфера спортивних послуг. Майже кожна людина прагне порядку та бажає планувати своє життя і хоче щоб навіть похід у фітнес-клуб був прогнозованим.

Базовими необхідності відвідувачів фітнес-клубів являють собою можливість придбати абонемент онлайн, сформувати персональний розклад враховуючи тренування, отримати повідомлення про наближення тренування. Для адміністрації критичною постає повна прозорість операцій, гнучке управління персоналом, прогнозування завантаженості клубів і підтримка високого рівня лояльності клієнтів.

З огляду на це виникає потреба у централізованому, технологічно нейтральному «ядрі», яке пропонує уніфікацію взаємодії, узгоджену модель даних. Такий підхід дозволяє одночасно знизити бар'єр входу для стартапів, що не можуть витрачати багато коштів на розробку ПЗ, та спростити цифрову еволюцію мережевих операторів, які прагнуть швидко додавати нові сервіси без докорінних змін архітектури застосунку. Відповіддю на цей виклик є концепція, викладена в данній дипломній роботі, що зосереджується на створенні спільного технологічного ядра для різноплатформних клієнтів.

Метою дослідження є розробка архітектурної моделі, що забезпечить стандартизацію бізнес-процесів бронювання, відвідування, платежів і взаємодії з персоналом. Вона має реалізовувати єдину точку авторизації та автентифікації з чітким розмежуванням ролей «клієнт», «тренер», «адміністратор». Підтримка горизнтального масштабування є критичною для уніфікованої платформи. Однією з ключових переваг має бути підвищення ефективності базової архітектури програми. Дана платформа дозволятиме розробникам створити

унікальний дизайн для свого користувацького інтерфейсу. Це дозволить кожному фітнес-клубу підкреслити свою унікальність.

Практичне значення полягає у скороченні часу й витрат на створення нових цифрових продуктів та підвищенні надійності завдяки централізації критичних сервісів — платежів, журналювання, статистики відвідувань.

З вище викладеного слідує, що уніфікація серверної частини додатків для фітнес-клубів є актуальною і важливою.

РОЗДІЛ 1
ОГЛЯД ФУНКЦІОНАЛУ ТА БІЗНЕС-ЛОГІКИ ІСНУЮЧИХ ЗАСТОСУНКІВ У СФЕРІ УПРАВЛІННЯ ФІТНЕС-КЛУБАМИ

1.1 Сучасні вимоги до цифрових платформ для фітнес-клубів

1.1.1 Ключові функції для управління фітнес-клубом

Доцільним є розпочати з питання, що є ключовими функціями, котрі формують фундамент сучасної цифрової платформи для фінтес-клубу? З одного боку, можна подумати, що достатньо мінімально життєздатного набору функцій, а з іншого — кожна прогалина може відгукнутися незадоволенням клієнтів або незручностями для персоналу або навіть новими витратами для власників бізнесу. Далі буде наведено осмислений перелік ключових функцій без надмірного занурення в них, проте з акцентом на те, що саме повинна вміти платформа, аби клуб залишався конкурентим.

Однією з ключових можливостей є змога управління клієнтськими профілями та контактами. Сервер має зберігати інформацію про кожного клієнта, їх контактні дані. Також корисним було б зберігати інформацію щодо медичних застережень або іншої корисної інформації про користувача.

За допомогою платформи планування і бронювання тренувань повинно бути легким. Це забезпечується за допомогою зберігання графіку занять у тренерів. Також плюсом цього є те, що таким чином можливо уникнути ситуацій, коли на один час у тренера має бути два різних клієнти. Фіксування тренувань клієнтів допоможе уникнути проблем з переповненнями.

Кожна послуга, що надається в фітнес-клубі є платною. Через це інформацію щодо оплати наданих послуг варто зберігати теж. Сучасні фітнес-клуби вже давно пропонують набагато більше ніж послуги тренера. Наприклад вони можуть продавати спортпіт. Доцільним є створення модуля який буде оброблювати оплату як мінімум абонемента на відвідування.

З покращенням якості наданий послуг допоможе аналітичний модуль. Сучасний тренер має працювати не тільки з самим клієнтом та «залізом» в залі. Він також має працювати з даними. Платформа повинна дозволяти створювати й редагувати шаблони для тренувань та формувати прості статистичні звіти щодо прогресу тренуючогося.

Для зручності клієнтів і змешення вірогідності того, що клієнт забуде про бронювання тренувань потрібно реалізувати нагадування про запис. Прості SMS вже давно відійшли на другий план, зараз зазвичай використовують push-нотіфікації. Тому така функція має бути реалізована в уніфікованій серверній платформі.

Будь яка платформа має включати в себе систему керування доступом та базові заходи щодо безпеки. Аутентифікація здійсюється за допомогою JWT токену – цей метод є одним з найпопулярніших для сучасних додатків. Для мінімізації ризику використовуються токени, що мають невеликий термін життя.

1.1.2 Типові бізнес-процеси та архітектурні патерни у фітнес-ПЗ

На перший погляд більшість фітнес-клубів працює за схожим сценарієм: продаж абонементів, бронювання індивідуальних або групових занять, контроль відвідуваності. Але при детільнішому аналізі виявляється, що мережа процесів, які щоденно відбуваються в залі є ширшою. Однією з ключових осіб в життєвому циклі клубу є клієнт. Типовий цикл клієнта виглядає так:

1. Клієнт приходить до клубу та реєструється в додатку за допомогою адміністратора ресепшину, обирає один з можливих абонементів та купує його. Ці події формують перший запис з базовою інформацією про особу в застоснку.
2. Клієнт згідно своїх потреб та уподобань регулярно бронює тренування або тренується сам. Він отримує повідомлення про наближення часу тренування. Інформація про його відвідування зберігається в системи для створення статистики.
3. Коли абонемент підходить до завершення, клієнт або продовжує його купуючи новий або завершує відвідування клубу. Інформація про користувача буде зберігатись деякий час після завершення дії абонемента.

Ці етапи життя охоплюють майже всю екосистему фінтес-клубу, але їх ще потрібно доповнити процесами керування персоналом. Створенням графіку проведення занять тренером, аналітики щодо кількості відвідувачів та їх прогресу. Ключовим моментом в цьому є те, що більшість дій є ініційованими саме клієнтом

Вибір архітектури впливає на масштабованість, швидкість розробки й, зрештою, економіку продукту. Зазвичай використовується один з наступних шаблонів:

Таблиця 1.1 – Шаблони архітектур, де вони застосовуються та їх плюс і потенційні обмеження

Шаблон	Де застосовується	Плюси	Потенційні обмеження
Клієнт–сервер	Мобільні й веб-додатки клубів	Чітке розділення відповідальностей, просте масштабування клієнтів	Єдиний точковий ризик на стороні шлюзу

Кінець таблиці 1.1

Шаблон	Де застосовується	Плюси	Потенційні обмеження
Мікросервіси	Платформи з багатьма філіями, франчайзингові мережі	Незалежні команди, гетерогенний стек	Складна інфраструктура
Event-driven (EDA)	Системи з реального часу: turnstile, пуш-повідомлення	Асинхронність, гнучке масштабування окремих черг	Складна розробка через проблеми в пошуці помилок

Найчастішим вибором серед шаблонів є клієнт-серверна архітектура, її структуру зображено на рис. 1.1. На першу думку може виникнути запитання: якщо мікросервіси забезпечують широку маштабованість а EDA – реактивність, чому ж більшість фітнес-платформ опираються на доволі класичний підхід клієнт–сервер? У цього є декілька причин.

Рисунок 1.1 – Структура архітектурного шаблону клієнт-сервер

Шаблон взаємодії «клієнт–сервер» забезпечує прозорість шляху данних. Більшість функцій подібних застосунків мають короткий життєвий цикл і повинні бути гарантованно виконаними. Завдяки данній моделі процес знаходження помилок, їх уникнення є найпростішим серед вище перелічених.

Такий сервіс буде легко маштабувати. Коли у клуба з’являється велика кількість нових користувачів все, що треба зробити, щоб забезпечити їх доступом до застосунку це додати ще один серверіс який буде забезпечувати

реалізацію їх потреб. У мікросервісній архітектурі в такому випадку довелось би перебудовувати цілу мережу сервісів.

Безпосередній контроль доступу також є перевагою. За допомогою зв'язку з API можна реалізувати вхід в зал навіть коли там немає нікого з адміністрації, наприклад в нічний час. Це дасть змогу збільшити потенціальну клієнтну базу. Клієнт-серверна модель дає чітку точку автентифікації, де можна об'єднати OAuth2 для мобільного додатка та апаратні токени для контролю доступу.

Цікаво зазначити, що й сам клієнт-сервер еволюціонував. Сучасна серверна частина фітнес-ПЗ — це вже не моноліт у чистому вигляді, а радше «композитний моноліт» із чітко виокремленими модулями. Вони спілкуються між собою всередині, а зовнішньому світу презентують себе як один шлюз. У такий спосіб поєднуються переваги спрощеної інфраструктури та гнучкості доменно-орієнтованих модулів.

Варто звернути увагу на те, що такий шаблон взаємодії має і свої мінуси. З одного боку, клієнт-сервер і справді знижує поріг входу в розробку. З іншого — моноліт може вирости до рівня, де кожна зміна зачіпає половину кодової бази. Для того щоб втримати баланс гарною практикою є створення чіткого контрактного шару, дотримання приципу єдиної відповідальності допоможе в цьому.

З вище викладеного слідує, що більшість застосунків для фітнес-клубів використовують шаблон «клієнт-сервер» через наявність великої кількості переваг при його виборі. До цих переваг відносяться швидка розробка, нескладна інфраструктура для забезпечення працездатності, маштабованість та прозорість шляху який проходить інформація. Саме тому відповідна архітектура не є мертвим раритетом – це продуманий компроміс між інженерною зрілістю та бізнес-економікою. Вона наочно проявляє свою силу там, де вимоги до консистентності переважають переважають мікросервісну маштабованість, а зручність для команди стає не менш вагомою ніж

витонченність технологій. Серверна частина в такому застосунку виконує роль «ядра», яке уніфікує бізнес-процеси клубу, надає спільний шлюх для клієнтських додатків і тримає цілісність даних. Саме це є вагомим аргументом для її вибори в сфері фітнес-індустрії, що швидко зростає та постійно експериментує

1.2 Огляд існуючих рішень

Огляд існуючих рішень – це не формальная традиція, а дієвий інструмент, що дозволяє підвищити якість нової розробки. Систематизація інформації щодо підходів допомагає з’ясувати яким чином були реалізовані функції, що потрібні для створення уніфікованої серверної платформи для фітнес-клубів. Аналіз сильних та слабких сторін розроблених застосунків зменшує ризик повторення вже кимось допущених помилок, та водночас допомагає відокремити та використати вдалі конструкції, що вже довели свою життєздатність. Обґрунтована аргументація, підкріплена фактами та результатами інших команд формує впевненість щодо обраного технічного курсу. Далі буде оглянуто два застосунки, що використовуються в реальних умовах.

1.2.1 Mindbody

Mindbody починала як спеціалізований календар для студій йоги але зараз вже виросла в масштабну програму-послугу, що допомагає понад сорока тисячам фітне-клубів по всьому світу. Розробники наголошують, що лише за минулий рік за допомогою їх застосунку, зображеного на рис. 1.2, було зафіксовано близько сімсот дванадцять мільйонів бронювань, а завантажено його було в середньому 6 разів на хвилину. Це підтверджує маштаб ринкового охоплення й стабільність застосунку [1].

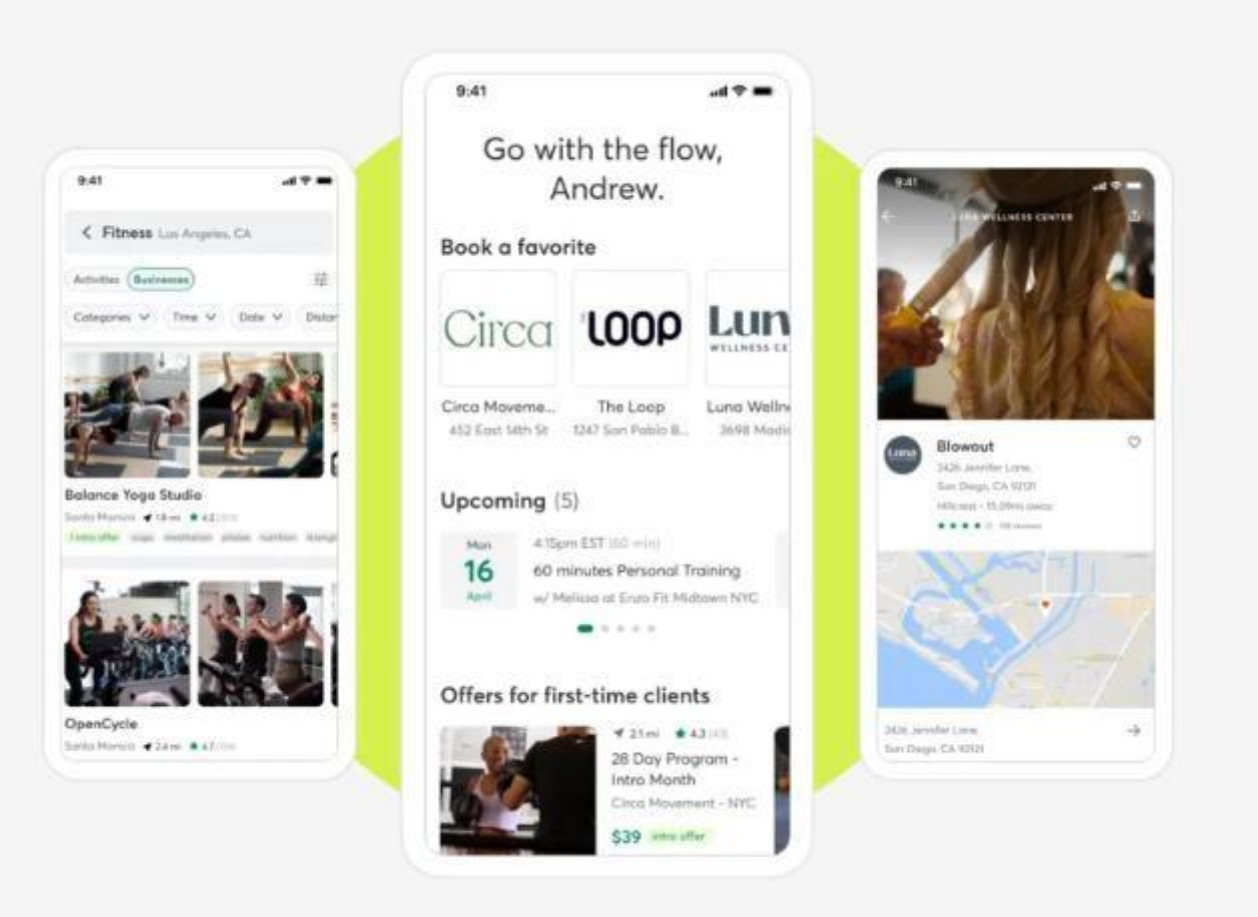


Рисунок 1.2 - Застосунок Mindbody [2]

Сценарій першого контакту користувача з застосунком починається з завантаження. Потім користувач в ньому реєструється та обирає адресу найближчого клубу й направляється туди на пробне заняття. В ході данного процесу в базі данних створюється профіль користувача з інформацією про нього.

Планування дана платформа реалізує через єдину панель керування, де адміністратор може змінювати місткість залу чи додавати онлайн-сесії. Розділ створення броней відображається на сайті. Система підтримує навіть накладання подій на невеликий проміжок часу задля того щоб уникнути конфліктів ресурсів. У підсумку витрати клуба на адміністрацію скорочуються, а користувач бачить реальну картину навантаженості клубу і отримує нагадування про свої бронювання [3].

Фінансові операції також є інтегрованими. Розділ оплат гарантує захищену обробку всіх транзакцій. Також в ньому реалізовано можливість автоматичного продовження абонементів для користувача. Таким чином це зменшує навантаження на менеджерів та майже прибирає необхідність у сторонніх терміналах. Таким чином Mindbody забезпечує майже безперервний грошовий потік для малих клубів та великих мережевий фітнес-операторів.

Аналітика в застосунку побудована за допомогою дашбордів. На них відслідковуються коефіцієнти утримання, середній чек, завантаженість залів. Сайт не розкриває внутрішньої реалізації даних аналітичних зведень.

Архітектура Mindbody є прикладом класичної хмарної моделі. На офіційному сайті наголошується, що платформа об’єднує двадцятирічний досвід і розвинену споживчу мережу, що дозволяє їй одночасно масштабуватися для глобальних брендів і залишатися доступною локальним студіям [1]. Серверна частина складається з набору сервісів, до яких звертаються фірмові застосунки, серед них є відкритий REST або GraphQL шлюзи. Таким чином забезпечується гнучкий вибір платіжнних постачальників, що актуально для такої інтернаціональної компанії як Mindbody.

З усього вищесказаного можна зробити висновок, що цей застосунок створює єдине «ядро» якє включає в себе всі сервіси потрібні для фітнес-клубів. Для українського ринку подібний підхід виглядає корисним орієнтиром. Поєдання маштабованої архітектури і нативних фінансових інструментів доводить, що уніфікована серверна частина може мінімізувати технологічну нерівність між маленькими клубами та гігантами індустрії таким чином підвищуючи лояльність клієнтів.

1.2.2 Trainerize

Про застосунок ABC Trainerize можна часто почути, що він мобільнеп продовження фітнес-бізнесу. Ця характеристика базується скоріше не на маркетингових гаслах а на конкретних цифрах, які компанія публікує у вільному доступі на їхньому сайті. На головній сторінці даної платформи зазначено, що через цю платформу щорічно проходить понад 186 мільйонів доларів. Застосунок, що зображено на рис. 1.3, же використовується як і окремими тренерами так і мережеві оператори клубів [4].

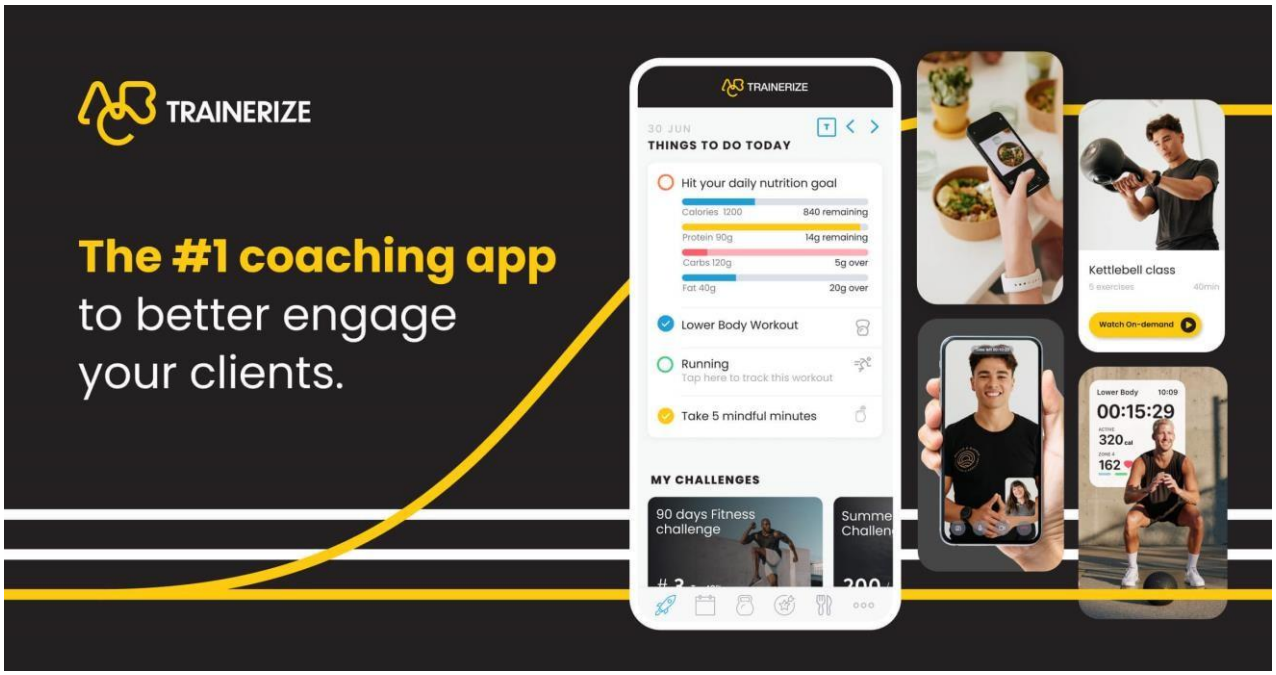


Рисунок 1.3 – Застосунок Trainerize [4]

Згідно з вище зазначеною інформацією можна вважати, шо система вже подолала критичну «масу користувачів» і перейшла в фазу, коли мережевий ефект генерує додану цінність навіть без активного розвитку. Клієнти все частіше рекомендують програму іншим власникам клубів й вони стають новими клієнтами.

Якщо спробувати відстежити життєвий цикл користувача, то його початок буде всередині мобільного застосунку. Потенційний клієнт завантажує

його і створює там свій профіль. В ньому ж в нього відображається його план тренувань. Trainerize одразу поєднує функції контент-платформи та платіжного шлюзу. Сайт сервісу підкреслює наявність прямої інтгерації зі Stripe. Він забезпечує можливість придбати послугу одним дотиком пальця, а тренер відразу отримає повідомлення про успішну оплату.

В застосунку існує моділ щоденного супроводу, він відповідальний за надсилання нагадувань, демонструє вправи котрі користувач має виконати та інше. В ньому також існує розділ в якому знаходиться інформація щодо оновлень плану тренувань в реальному часі. Також є інтеграція з розумними речами такими як годинники.

Фінансова логіка платформи не обмежується Stripe. У довіднику користувача зазначена можливість підключати й інші платіжні системи, наприклад PayPal [5]. Таки вид інтеграції означає відсутність потреби отримувати окремі рахунки й мінімізує вірогідність людської помилки при звірці підписок.

Архітектурно команда, що розробляла застосунок, відкрито декларує наявність REST та GraphQL шлюзів для партнерської екосистеми [6]. Завдяки такій реалізації досить легким є підключення сторонніх сервісів. Це є підставою вважати що розробники приділили велику кількість уваги швидкому маштабуванню, делегуючи спеціалізацію сценаріїв власникам клубів або спільноті розробників.

Типовий бізнес-процес в Trainerize починається з продажу програми. Після оплати тренер формує план тренувань та надсилає його клієнту. Весь життєвий цикл відбувається всередині одного застосунку, що для користувача є зручним безшовним рішенням, а для тренера коридором стабільного зв’язку з клієнтом. Стастистика не є щоденним елементом в даному процесі, проте компанія наголошує, що вона прислуховується до зворотнього зв’язку спільноти [7].

Через все вищесказане Trainerize можна розгла як приклад гнучкої та комплексної системи де менеджмент та аналітика об’єднані. Цей кейс є цінним прикладом з відкритим API, широким вибором платіжних провайдерів. Цей додаток являє собою цілісний додаток-сервіс який враховуючи все вищесказане не жертвує безпекою даних та функціональною цілісністю.

1.3 Порівняльний аналіз застосунків

Mindbody і ABC Trainerize являють собою дві компліментарні, але різні картини цифрової трансформації бізнесу. Перша платформа об’єднує понад сорок тисяч студій, генеруючи сотні мільйонів бронювань на рік і поступово перетворюючи власний календар на глобальний агрегатор попиту. Trainerize же виріс із мобільного щоденника тренера тож його логіка підштовхує до створення середовища, де смартфон клієнта є центром планування, оплати та зворотнього зв’язку.

Якщо порівняти шлях клієнта, відразу помітно, що Mindbody концентрується на тому, що клієнт має прийти на заняття. А Trainerize забезпечує програму тренувань відразу.

Фінансові механіки обох платформ є досить схожими. Mindbody окрім класичного онлайн процесингу пропонує власний бізнес ресурс Mindbody Capital, Trainerize же не створює власного банку а використовує вже існуючі рішення типу PayPal. Стратегія цього застосунку зменшує вхідний поріг для фрілансерів, яким не потрібная громіздка фінансова інфраструктура.

Архітектурно обидва продукти виконують хмарний підхід, проте ступінь відкритості відрізнається. Mindbody пропонує API, але основні бізнес-функції упаковані у вертикально сегментовані модулі. Trainerize відкрито рекламує свій відкритий API шлюз і спонукає користувачів самостійно збирати потрібний їм стек. Ці розбіжності вказують на різні пріоритети маштабування. Mindbody

робить ставку на екосистему, де нові курси клієнтів надходять із власного маркетплейсу. Trainerize збільшує дохід тренера через постійне залучення кінцевого користувача, адже чим частіше клієнт відкриває додаток, тим вище ймовірність прибутку.

Таблиця 1.2 – Критерії та порівняння додатків

Критерій	Mindbody	ABC Trainerize
Головний напрям розвитку	Централізована автоматизація операцій клубу	Мобільний персоналізований коучинг
Фінансова модель	Власний процесинг плюс Mindbody Capital	Інтеграції Stripe/PayPal
Масштаб користувачів	40 000+ студій, 712 млн бронювань	45 000 бізнесів, 400 000 тренерів
Можливі інтеграції	Власна «шина» + обмежена кастомізація	Zapier-конектор
Ризик-фактор	Менша швидкість UX-експериментів	Залежність від зовнішніх шлюзів
Аналітика	Аналітика в реальному часі	Мікро-аналітика клієнта

Порівняльна таблиця демонструє дві парадигми відповідних додатків. Mindbody має на меті забезпечити стабільну заповненість залів та прогнозований дохід, чому сприяє власний платіжний контур. Аналітика зосереджена на індикаторах в реальному часі. API шлюз є відкритим але тісно прив’язаним до інших модулів.

Trainerize виходить зі логіки де в центрі стоїть мобільний застосунок: смартфон клієнта стає щоденним інструментом, що допомагає в тренуваннях. Платежі інтегруються через існуючі рішення типу PayPal.

Такий підхід додає гнучкості але робить застосунок залежним від сторонніх шлюзів. Обидва підходи мають набір практик, які можна використати для створення гібридної уніфікованої платформи для локального ринку.

ВИСНОВОК ДО РОЗДІЛУ 1

Проведений огляд цифрових рішень для управління фітнес-клубами переконує, що навіть найуспішніші платформи концентруються навколо обмеженого, але логічно зв’язаного набору сервісів. Порівняння Mindbody та ABC Trainerize показало: ядро системи формується не переліком додаткових функцій, а здатністю стабільно обслуговувати фундаментальні сценарії взаємодії між клієнтом, тренером і адміністрацією. На цій підставі логічним припущенням є, що для реалізації уніфікованої серверної частини доцільно сфокусуватися лише на тих функціях, які прямо впливають на якість сервісу та фінансову стійкість клубу.

Отже, з усього спектра бізнес-процесів було свідомо обрано чотири ключові напрями. По-перше, створення записів про тренування, причому і про самостійні сесії клієнта, і про заняття з тренером. Такий крок дозволяє, з одного боку, фіксувати фактичну активність користувача, а з іншого — надавати тренерам дані для персоналізації майбутніх програм. По-друге, сервіс нагадувань, що автоматично інформує клієнта про наближення тренування. Практика існуючих систем доводить: своєчасне повідомлення помітно знижує кількість пропусків і підвищує лояльність. По-третє, інтегрований модуль обробки платежів, який забезпечує безпечний прийом коштів. Саме цей компонент робить платформу не просто обліковою, а фінансово завершеною. Нарешті, сервіс простих статистичних звітів, що агрегує базові показники відвідуваності. Можна припустити, що для більшості малих і середніх клубів саме швидка аналітика. Вона дає найбільшу додану вартість, бо допомагає ухвалювати рішення в режимі реального часу.

Всі вказані функції природним чином укладаються в класичну модель «клієнт – сервер». Клієнтські застосунки будь-якого типу звертаються до єдиного API-шлюзу, отримуючи узгоджену поведінку й спільну модель даних.

Такий підхід мінімізує дублювання логіки, спрощує масштабування й спирається на перевірені практики безпеки. Сервер концентрує всі критичні механізми і при цьому клієнтські частини залишаються тонкими та легко адаптуються до дизайнерських вимог. Саме тому надмірно фрагментована мікросервісна архітектура на ранніх етапах розвитку платформи стала б передчасним ускладненням, тоді як модель «клієнт – сервер» пропонує збалансований компроміс між швидким запуском і подальшою еволюцією.

Обрана конфігурація може здатися мінімалістичною, але вона охоплює весь життєвий цикл взаємодії клієнта з клубом. Таке звуження фокуса дозволить сконцентрувати ресурси розробки на поліпшенні надійності та продуктивності без розпорошення сил на другорядний функціонал. Втім, слід наголосити: модель залишається відкритою до розширення завдяки чітким API-контрактам, а отже в майбутньому може безболісно прийняти розширену аналітику чи підтримку зовнішніх маркетингових сервісів.

Підсумовуючи сказане, можна стверджувати, що результати огляду підтвердили доцільність вибору саме цих чотирьох сервісів як базових. Вони утворюють взаємодоповнювальний ланцюг, де кожен елемент підсилює інші, а обраний клієнт-серверний шаблон гарантує прозору інтеграцію й передбачувану експлуатацію. Саме така комбінація формує надійну платформу, здатну служити стартовим майданчиком для подальшого розвитку універсальної екосистеми цифрових послуг у фітнес-індустрії.

РОЗДІЛ 2
ОГЛЯД ТА ВИБІР ТЕХНОЛОГІЙ ДЛЯ РОЗРОБКИ УНІФІКОВАНОЇ ПЛАТФОРМИ

2.1 Вибір мови розробки

Вибір мови програмування є одним з найважливіших рішень при розробці будь-якого ПЗ. Він має принципове значення, що визначає майбутню ефективністть, гнучкість, швидкість розробки і в кінці-кінців успішність всього проекту. Саме мова програмування задає базові межі та умови, в рамках яких вже і відбувається подальший розвиток застосунку. У данному контексті варто зазначити, що немає універсального рішення, котре буде найефективніших при вірешнні всіх можливих задач.

Цей вибір впливає на швидкість і зручність реалізації функціоналу. Також важливим фактором є надійність та безпека. Застосунки, що зберігають у собі персональну інформацію своїх користувачів, вимагають підвищеної уваги, щодо питань кібербезпеки. Через це доцільним є розглядання мов з суворою типізацією. Такі мови забезпечують додатковий рівень безпеки за допомогою чіткої структури та ефективних інструментів тестування. Вони дозволяють уникнути багатьох потенційних помилок ще на етапі компіляції.

Не останню роль також відіграє спільнота розробників. Чим більше є розробників, що використовують певну мову, тим легше буде створити дійсно досвідчену команду, котра буде виконувати завдання швидко та не припускатиметься помилок при розробці.

Вибір мови також має враховувати й аспекти інтеграції з іншими системами. Багато застосунків в різних сферах потребують взаємодії з іншими зовнішними сервісами, платіжними системами, соціальними мережами тощо.

Підсумовуючи все вище сказане, вибір мови є відповідальною задачею, що не має єдиного правильного рішення а залежить від баагатьох факторів, що було перелічено.

Нижче було оглянуто мови, що можна було б використати для розробки.

2.1.1 Python

Python активно використовується у багатьох галузях – від веб-розробки й аналізу даних до штучного інтелекту й автоматизації бізнес-процесів.

Серед головних переваг даної мови варто виділити простоту й читабельність, синтаксис мови зображено на рис. 2.1. Це значно зменшує час навчання та підвищує ефективність розробників, знижуючи поріг входу для нових членів команди. Серверна архітектура застосунку для організації роботи фітнес-клубів чіткої та маштабованої структури, використання Python в даному випадку забезпечить комфортну і зрозумілу основу для горизонтального розширення застосунку. Згідно офіційної документації синтаксис мови був розроблений з акцентом на простоту й зрозумілість, що має мінімізувати можливість допущення помилок у процесі розробки [8].

```python
# Simple to-do list in Python

class Task:
    def __init__(self, title): self.title = title; self.done = False
    def __str__(self): return f"[{'✓' if self.done else ' '}] {self.title}"

class ToDoList:
    def __init__(self): self.tasks = []
    def add(self, title): self.tasks.append(Task(title))
    def show(self):
        if not self.tasks: print("No tasks.")
        for i, t in enumerate(self.tasks, 1): print(f"{i}. {t}")
    def complete(self, i):
        try: self.tasks[i - 1].done = True
        except IndexError: print("Invalid number.")

def main():
    todo = ToDoList()
    while True:
        cmd = input("\n1=add 2=show 3=done 4=exit: ")
        if cmd == "1":
            todo.add(input("Title: "))
        elif cmd == "2":
            todo.show()
        elif cmd == "3":
            try: todo.complete(int(input("Number: ")))
            except ValueError: print("Enter a number.")
        elif cmd == "4":
            break
        else:
            print("Unknown command.")

if __name__ == "__main__"
    main()
```

Рисунок 2.1 – Простий список справ на мові Python

Другою перевагою Python є ровинена екосистема бібліотек і фреймворків. Наприклад Django і FastAPI підходять для розробки серверної логіки веб-застосунків. Django забезпечує потужну архітектуру з вбудованими інструментами для авторизації, роботи з базами даних та адміністрування.[9] FastAPI ж, згідно документації, створений спеціально для високопродуктивних веб-сервісів, які потребують асинхронного виконання задач [10]. Використання цих фреймворків може суттєво знизити витрати часу й ресурсів на розробку та

підтримку проєкту. Python також відомий високою швидкістю розробки. Зважаючи на широкий вибір стандартних бібліотек і широку доступність готових рішень, розробник може зосередитись на розробці прикладної логіки.

Однак ця мова є далеко не ідеальною й має низку недоліків. Один з цих недоліків – відносно низька швидкодія в порівнянні з компільованими мовами програмування. Це означає що при виконанні задач, потребуючих значні обчислювальні ресурси, застосунки на Python можуть поступатисяза продуктивністю. Втім для більшості завдань, що пов'язані з функціями фітнес-клубів, ця проблема є майже непомітною.

Ще одним аспектом, який може викликати сумніве, є динамічна типізація. Вона надає гнучкості при розробці, але при цьому може призводити до труднощів або проблем з безпекою.

2.1.2 JavaScript

JavaScript є однією з найпопулярніших мов програмування. Це, певно, спричинено насамперед його унікальною здатністю забезпечувати повноцінну інтеграцію клієнтської та серверної частин у межах єдиної технологічної екосистеми.

Однією з головних переваг JavaScript є його універсальність і гнучкість. За допомогою цієї мови програмування можна створити повноцінний застосунок. Такий підхід помітно скорочує час розробки, спрощує комунікацію всередині команди й полегшує підтримку. Для розробки платформи для фітнес-клубів використання JavaScript забезпечить швидку адаптацію під різні платформи й інтеграцію з зовнішніми сервісами. Node.js забезпечує асинхронну модель обробки запитів, приклад зображено на рис. 2.2, яка дозволяє серверам ефективно витримувати великі навантаження без суттєвих затримок, що особливо актуально для фітнес-клубів у години пікового навантаження при реєстрації на тренування чи здійсненні онлайн-платежів [11].

```javascript
// Simulated request queue
const requestQueue = [];

// Simulate adding requests
function addRequest(id) {
  requestQueue.push({ id, timestamp: Date.now() });
}

// Simulate processing a request with delay
async function processRequest(request) {
  console.log(`Processing request ${request.id}...`);
  return new Promise(resolve => {
    setTimeout(() => {
      console.log(`Done with request ${request.id}`);
      resolve();
    }, 500);
  });
}

// Asynchronous request handler
async function handleRequests() {
  while (requestQueue.length > 0) {
    const request = requestQueue.shift();
    await processRequest(request);
  }


// Simulate incoming requests
addRequest(1);
addRequest(2);
addRequest(3);

// Start processing
handleRequests();
```

Рисунок 2.2 – Симуляція асинхронної обробки запитів на мові JavaScript

Ще однією вагомою перевагою використання даної мови є велика кількість різних фреймворків і бібліотек, що значно прискорюють швидкість

розробки. Окремо варто відзначити також високу швидкість JavaScript-застосунків завдяки асинхронності та неблокуючому вводу-виводу, що забезпечує оперативну обробку великої кількості одночасних підключень. У контексті застосунку до якого будуть надсилати багато запитів одночасно наявність такої системи є вагомою перевагою. Крім того наявність великої спільноти розробників сприяє швидкому знаходженню вирішень на велику кількість помилок, що можуть виникнути під час роботи.

Втім, слід пам’ятати й про суттєві недоліки цієї мови. JavaScript є динамічно типізованою мовою, тому й має всі відповідні мінуси згадані при розгляді мови Python. Інший аспект, який потрібно врахувати, це те, що без чіткої архітектури та контролю якості, кодова база має тендецію ставати менш читабельною та складною для підтримки. Також JavaScript показує схожі резаультати з мовою Python і в задачах, де потрібна висока інтенсивність обчислень. Для реалізації даного застосунку цей фактор не критичним.

2.1.3 TypeScript

TypeScript є надбудовою над JavaScript, яка додає статичну типізацію і низку корисних конструкцій, що суттєво підвищує стабільність та підтримуваність проєктів. Популярність TypeScript значною мірою зумовлена потребою подолати деякі недоліки, притаманні JavaScript.

Однією з ключових переваг цієї мови є статична типизація, приклад коду зображено на рис. 2.3, яка дозволяє виявити помилки до початку виконання коду. В застосунках зі складною логікою використання TypeScript є надійним рішенням. Це дозволяє підтримувати високий рівень якості коду, просто підтримки та розшинення та чіткість структури [12].

```typescript
interface User {
  id: number;
  name: string;
  email: string;
}

const mockUsers: User[] = [
  { id: 1, name: "Alice", email: "alice@example.com" },
  { id: 2, name: "Bob", email: "bob@example.com" },
];

function fetchUserById(id: number): Promise<User> {
  return new Promise((resolve, reject) => {
    const user = mockUsers.find(u => u.id === id);
    setTimeout(() => {
      user ? resolve(user) : reject(new Error("User not found"));
    }, 500);
  });
}

async function showUser(id: number): Promise<void> {
  try {
    const user: User = await fetchUserById(id);
    console.log(`Name: ${user.name}, Email: ${user.email}`);
  } catch (error) {
    if (error instanceof Error) {
      console.error("Error:", error.message);
    }
  }
}

showUser(1);
```

Рисунок 2.3 – Імітація запиту на отримання користувача на мові TypeScript

Використовуючи цю мову програмування розробник отримує можливість значно швидше орієнтуватись в кодовій базі за допомогою чітко визначених інтерфейсів та типів. Вони полегшують процес розуміння того як взаємодіють різні частини програми. Це стає особливо важливим для застосунків, що постійно розширюються та змінюють свій вигляд. Використовуючи TypeScript, команда розробників може суттєво зменшити час, необхідний для

рефакторингу та розширення коду, що забезпечує більш гнучкий і швидкий розвиток проєкту.

Ще однією з переваг цієї мови є те, що вона повністю сумісна з всіма інструментами що використовує JavaScript. Інакше кажучи, це означає що для розробки можна використовувати будь-яку бібліотеку та фреймворк для JavaScript і при цьому отримати перевагу у вигляді суворої типізації.

Незважаючи на очевидні переваги, варто згадати що мова має також слабкі сторони. TypeScript потребує транспіляції в свого динамічно типізованого попередника. Таку особливість не можна назвати критичною, але вона все ж створить незначну, але все ж таки помітну, затримку на етапі розробки. Завдяки сучасним інструментам, таким як Vite, це процес опитмізується і не займає велику кількість часу.

Щодо ефективності застосунку, TypeScript не має суттєвих переваг над JavaScript, оскільки компілюється в останній перед виконанням. Проте чітка типізація допомагає уникати деяких помилок і позитивно впливає на швикдодію при цьому роблячи код ефективнішим.

2.1.4 Підсумок, щодо вибору мови

Враховуючи особливості й вимоги до розробки уніфікованої серверної частини застосунку для фітнес-клубів, можна зробити аргументований висновок, що найбільш доцільним вибором у цьому контексті є TypeScript. Вибір саме цієї мови зумовлений низкою вагомих переваг, які роблять її оптимальним рішенням у порівнянні з такими альтернативами як Python і JavaScript.

Одним з важливих факторів на користь вибору даної мови є її сувора типізація. Якщо порівнювати з альтернативами розглянутими вище дана мова забезпечує суттєво вищий рівень надійності та стабільності програмного продукту. В умовах складних, розгалужених проєктів, до яких і належить

розробка застосунку для фітнес-клубів, подібна перевага є досить вагомою адже вона дозволяє зменшити витрати на тестування й підтримку продукту в майбутньому.

TypeScript суттєво покращує підтримуваність і масштабованість коду за допомогою використання інтерфейсів, структурованих моделей і механізмів впровадження залежностей. Такий підхід гарантує вищу читабельність та зрозумілість програмного коду, що є вкрай важливим в даній швидкозмінній сфері. В порівнянні з JavaScript, що без чіткої архітектури може швидко перетворитися на важкий у підтримці код, TypeScript пропонує значно більшу прозорість та керованість процесу розробки.

Варто також згадати, що код на обраній мові бездоганно інтегрується в екосистему JavaScript. Завдяки повній сумісності використання всіх сучасних бібліотек та фреймворків, що є частиною зазначеної вище екосистеми, не виникає додаткових складнощів при створенні застосунку. TypeScript забезпечує баланс між гнучкістю JavaScript і суворою архітектурою, характерною для мов, орієнтованих на великі корпоративні рішення.

Крім того, використання обраної мови полегшує взаємодію серверної частини з клієнтською, оскільки вона може бути використана для розробки обох частин застосунку. Розглянутий вище Python вимагає знань і навичок у різних мовах.

TypeScript має велику та активну спільноту і активно розвивається, це забезпечує стабільну підтримку та регулярні оновлення. Хоча Python і JavaScript теж мають потужні спільноти, TypeScript останнім часом демонструє більш виразну динаміку розвитку, особливо в контексті корпоративних і бізнес-орієнтованих рішень.

Варто зазначити, що процес початкового освоєння обраної мови може бути довшим ніж у аналогів, але ця інвестиція компенсується більш високою якість програмного коду і спрощенням довгострокової підтримки.

Підсумовуючи все вище викладене, можна стверджувати, що використання TypeScript є оптимальним для створення уніфікованої пдатформи для розробки застосунків для організації роботи фітнес-клубів. Він дозволяє забезпечити високу якість, стабільність, зручність підтримки коду, швидкість розробки та ефективність взаємодії між розробниками.

2.2 Вибір СУБД

Вибір системи управління базами даних є не менш важливим ніж вибір мови розробки, цей вибір теж суттєво впливає на ефективність та надійність всього програмного рішення. Саме база даних являє собою центральну складову, що забезпечує зберігання, обробку та ефективний доступ до всієї критично важливої інформації — починаючи з обліку клієнтів і завершуючи розкладом тренувань.

Слід зауважити, що сучасні фітнес-клуби можуть працювати з достатньо великим обсягом даних, які мають структурований характер. Відповідно до даної вимоги СУБД має бути достатньо гнучкою, здатною працювати із різними типами даних без суттєвих компромісів у швидкості та надійності.

З одного боку існують традиційні реляційні СУБД, такі як PostgreSQL, вони демонстують високу надійність у роботі з чітко визначеними структурами. Їх особливість це сувора відповідність схемам, що дозволяє мінімізувати ризик виникнення помилок у роботі з критичною інформацією. Практичним прикладом може бути PostgreSQL, яка завдяки широким можливостям підтримки складних запитів і транзакційної цілісності є однією з найпопулярніших для веб-застосунків у сфері послуг.

Але не варто забувати й про сучасні NoSQL-рішення (не реляційні БД), як MongoDB. Їх основною перевагою є гнучкість для роботи з

неструктурованими даними. Також схожі застосунки можуть досить швидко адаптуватися до змін у бізнес-логіці.

Одним з головних критеріїв при виборі СУБД для застосунку, котрий буде використовуватися у фітнес-клубах, є можилвість до маштабування та продуктивність. Адже клієнтна база може дуже швидко зростати, особливо враховуючи популяризацію здорового способу. Тому вибрана СУБД має забезпечувати горизонтальне або вертикальне масштабування, зберігаючи стабільність та доступність системи. Наприклад, PostgreSQL, хоч і має чудові характеристики продуктивності, зазвичай масштабують вертикально, в той час як MongoDB легко масштабується горизонтально, що може бути значною перевагою на довгостроковому етапі експлуатації системи.

Не існує єдиного підходу до вибору СУБД, який був би оптимальним для всіх випадків. Вибір, звісно, залежить від вимог проєкту. Саме тому перед приняттям рішення варто проаналізувати можливі варіанти. Такий підхід гарантуватиме стабільність, продуктивність і врешті-решт задоволення кінцевих користувачів. Нижче буде розглянуто одну з самих популярних реляційних та не реляційних СУБД.

2.2.1 MongoDB

MongoDB являє собою перспективне рішення, здатне значно спростити процес створення та подальшого супроводу інформаційних систем. Ця база даних зазвичай використовується у роботі з різноманітними даними, що не завжди можливо забезпечити класичними реляційними базами.

Дана СУБД належить до класу документно-орієнтованих нереляційних систем зберігання інформації, її схематична структура зображена на рис. 2.4. Центральним елементом зберігання є документ формату BSON, що аналогічний звичному JSON, але містить додаткові типи для підвищення ефективності. Важливо зазначити, що, згідно з документацією MongoDB, BSON дозволяє

значно покращити ефективність зберігання і обробки даних порівняно зі звичайним JSON [13].

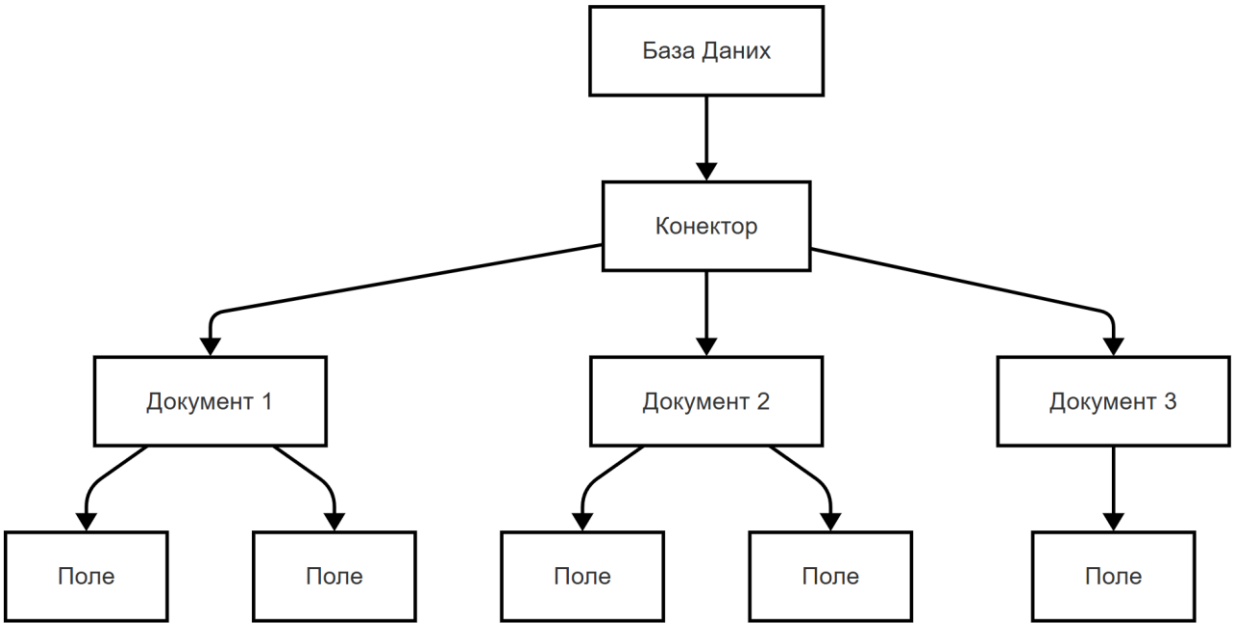


Рисунок 2.4 – Спрощений вигляд структури MongoDB

Однією з переваг MongoDB є відсутність жорсткої структурної схеми даних, що дозволяє динамічно змінювати її відповідно до бізнес-вимог. Така побудова суттєво прискорює внесення змін до структури бази без зупинки або перезавантаження системи. MongoDB підтримує концепцію «гнучких схем» [14].

Додатковою перевагою MongoDB є високий рівень горизонтального масштабування завдяки механізму шардінгу (sharding). При збільшенні навантаження на базу, наприклад, за умов активного росту кількості клієнтів або різних маркетингових кампаній, MongoDB може розподіляти дані між багатьма серверами, забезпечуючи тим самим стабільну роботу без втрати продуктивності [15].

Ще одним важливим моментом є висока швидкість роботи з даними. Вона досягається за допомогою гнучкої системи індексації. MongoDB підтримує створення простих і складних індексів, включаючи багаторівневі комбіновані

індекси, які можуть суттєво підвищити швидкість виконання запитів та загальну продуктивність роботи системи.

Попри зазначені переваги, є й певні обмеження. Зокрема, MongoDB поступається класичним реляційним базам даних за рівнем транзакційної підтримки, хоча останні версії вже дозволяють виконувати складні транзакції між декількома колекціями. Також слід враховувати, що при роботі з дуже складними та взаємозалежними даними MongoDB може поступатися в продуктивності реляційним СУБД.

2.2.2 PostgreSQL

PostgreSQL виступає одним з найнадійніших і функціонально багатих рішень на сучасному ринку СУБД. Це пояснюється здатністю системи ефективно вирішувати складні завдання, пов'язані зі зберіганням, обробкою та управлінням даними різного рівня складності та структури. PostgreSQL є потужною реляційною СУБД з відкритим кодом, яка розвивається понад 30 років і постійно покращується завдяки великій спільноті розробників. СУБД має повну підтримку стандарту SQL, а також безліч розширень, що робить її універсальною для вирішення різних бізнес-задач. Однією з найбільших переваг PostgreSQL є її високий рівень транзакційної підтримки, яка відповідає стандартам ACID. Це означає, що будь-які операції над даними гарантовано виконуються повністю або взагалі не виконуються, що є критично важливим при роботі з фінансовими транзакціями, бронюванням послуг, продажем абонементів тощо [16].

Система запитів PostgreSQL дозволяє виконувати надзвичайно складні запити, які містять багаторівневі JOIN-операції, аналітичні функції та різноманітні методи агрегації. Завдяки цьому вона являє собою чудовий вибір для застосунків, які потребують регулярного формування звітів.

PostgreSQL пропонує широкі можливості роботи з індексами, зокрема багаторівневими, частковими та функціональними індексами. Це суттєво прискорює процес вибірки даних та оптимізує продуктивність роботи бази навіть за великих обсягів інформації [17].

Щодо масштабованості, PostgreSQL ефективно масштабується вертикально, тобто покращення продуктивності досягається шляхом додавання потужніших апаратних ресурсів сервера, спрощений вигляд якого зображено на рис. 2.5. Проте, горизонтальне масштабування може бути складнішим завданням, хоча сучасні рішення, такі як реплікація, шардінг за допомогою сторонніх інструментів значною мірою вирішують цю проблему.

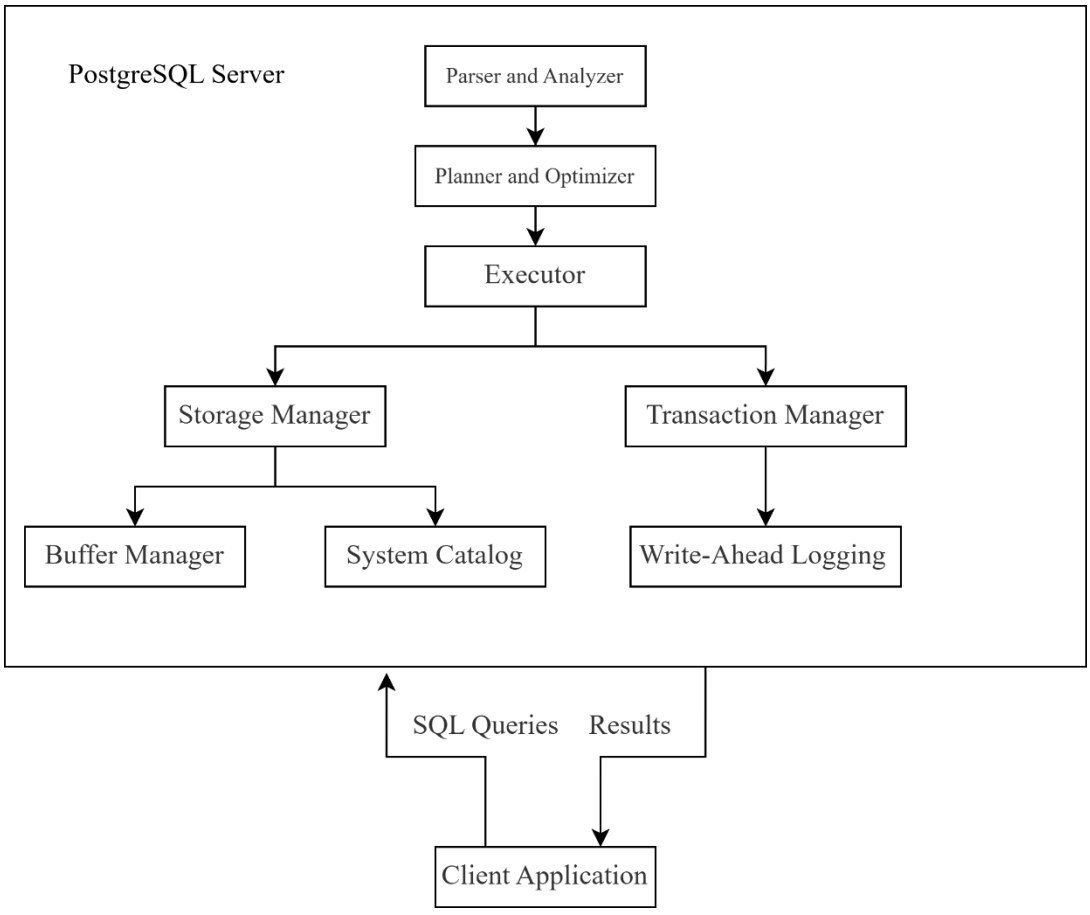


Рисунок 2.5 – Спрощений вигляд внутрішньої структури PostgreSQL сервера

Зручність інтеграції PostgreSQL з сучасними фреймворками є також важливою перевагою. Втім, варто зазначити й певні обмеження, що властиві PostgreSQL. Одним із них є сувора відповідність структурі бази даних, що може вимагати ретельного планування міграцій у разі змін бізнес-логіки або структури даних. Це може створювати певні труднощі в гнучких проєктах з високою частотою змін.

2.2.3 Підсумок, щодо вибору СУБД

Враховуючи всі фактори подані вище, мною була обрана СУБД PostgreSQL. Вибір ґрунтується на потребі досягти максимальної надійності, узгодженості та гнучкості водночас. З одного боку, реляційна модель даних забезпечує чітку структуру й зрозумілу організацію інформації про клієнтів, абонементи, тренування, а з іншого — розширена підтримка ACID-транзакцій гарантує безпечне виконання всіх критичних операцій, що є незамінним у випадку обробки платежів та бронювання послуг.

Ключовими факторами вибору було саме поєднання потужності SQL-аналітики та стабільності виконання складних операцій. Адже адміністратори та менеджери фітнес-клубів потребують детальної статистики завантаженості залів, ефективності роботи тренерів.

Питання маштабування також є важливим. З одного боку, PostgreSQL більш природно масштабується вертикально за допомогою розширення апаратних ресурсів, що підходить для середніх і великих інфраструктур, а з іншого — горизонтальні стратегії, реалізовані через реплікацію, кластеризацію чи сторонні інструменти, забезпечують додаткові шляхи для досягнення високої доступності та відмовостійкості.

Незважаючи на гнучкість NoSQL-рішень, простоту зміни схеми даних, яку пропонують MongoDB чи інші документно-орієнтовані бази, було надано первагу гарантуванню цілісності транзакційних операцій. В даному випадку

пріоритетом було забезпечити максимальну надійність, зручність інтеграції з іншими компонентами системи та можливість виконувати складні аналітичні запити без втрати продуктивності. Саме тому PostgreSQL видається найбільш оптимальним вибором: вона поєднує перевірену десятиліттями стійкість, потужність реляційної моделі та набір розширень для сучасних задач, що робить її ключовим елементом успішної реалізації уніфікованої серверної частини застосунку для фітнес-клубів.

2.3 MikroORM

Інтеграція MikroORM у серверну частину значно підвищує продуктивність розробки та полегшує підтримку коду. По-перше, цей інструмент забезпечує єдиний інтерфейс для роботи з різними базами даних, що відповідає головній меті проєкту — уніфікації та спрощення створення рішення для фітнес-клубів. Використання репозиторіїв і сутностей суттєво скорочує обсяг шаблонного коду оскільки розробник фокусується на бізнес-логіці, а не на низькорівневих SQL-запитах.

MikroORM реалізує патерн Unit of Work, який узагальнює транзакції і гарантує цілісність даних. З одного боку, це забезпечує оптимальне групування змін до бази даних в єдину транзакцію, з іншого — знижує ризик виникнення конкуренції під час паралельної обробки запитів. Наприклад, коли користувач оновлює свій персональний план тренувань і водночас адміністратор вносить правки у розклад клубу, Unit of Work автоматично формує коректний набір SQL-команд та виконує їх у межах однієї транзакції. Таке рішення дозволяє уникнути помилок запису й забезпечити консистентність стану програми. Завдяки цьому реалізація складних сценаріїв роботи з даними стає більш передбачуваною і надійною.

Також MikroORM може створювати міграції, що значно скорочують час налаштування та оновлення схеми бази даних. З одного боку, вони гарантують синхронізацію між моделями сутностей у коді та таблицями в СУБД, з іншого — спрощують відкат змін у разі помилок.

Також слід звернути увагу на інтеграцію з TypeScript. MikroORM забезпечує сувору типізацію сутностей та полів, що відповідає вимогам сучасних проєктів з високими стандартами надійності. Наприклад, при створенні сутності у коді можна явно вказати типи полів, на рис. 2.6 продемонстровано приклад описання сутності MikroORM з використанням раніше обраної мови TypeScript. Це дає змогу виявляти помилки ще на етапі компіляції, а не під час виконання на сервері.

```typescript
import { Entity, PrimaryKey, Property, MikroORM } from '@mikro-orm/core';

@Entity()
class User {
  @PrimaryKey()
  id!: number;

  @Property()
  name!: string;

  @Property()
  email!: string;

  @Property({ onCreate: () => new Date() })
  createdAt = new Date();
}
```

Рисунок 2.6 – Приклад MikroORM сутності описаної на TypeScript

Екосистема MikroORM включає кешування, розширювані плагіни та потужний QueryBuilder. З одного боку, кешування друкує повторні запити до диску, з іншого — підвищує швидкодію найчастіше використовуваних даних.

Користь від застосування MikroORM полягає в уніфікації доступу до даних та збереженні чистої архітектури серверної частини. Стабільний

інтерфейс для роботи з сутностями, можливість легкої заміни СУБД, автоматичні міграції, статична типізація й кешування створюють комплексне рішення, яке відповідає потребам як малого стартапу, так і великої мережі фітнес-клубів. Вибір цього інструмента обумовлений не лише зручністю розробки, а й довгостроковою підтримкою та масштабованістю системи. Наявність всіх цих переваг робить MikroORM вагомим елементом у серверній архітектурі проєкту.

2.4 Вибір бібліотеки для розробки

Вибір бібліотеки для розробки має фундаментальне значення і не зводиться лише до можливості швидко написання кодової бази. Від рішення залежить не лише продуктивність системи а й її подальша еволюція. З одного боку обрана бібліотека визначає набір інструментів і патернів які будуть доступні розробнику з іншого — вона накладає певні обмеження на архітектурні рішення.

Доцільно розглянути три ключові сфери впливу бібліотеки. Перш за все це підтримка продуктивності та масштабованості. В години пік коли одночасно обробляються велика кількість запитів від користувачів важливо мати швидкий механізм кешування та оптимізовану роботу з базою даних. Бібліотека має бути гнучкою та дозволяти за потреби виходити за рамки високорівневого API.

Вибір бібліотеки впливає також на зручність командної роботи та швидкість залучення в проєкт нових розробників. Якщо використовується досить популярний пакет з активною спільнотою то новий інженер швидше розбереться з документацією прикладами і готовими рішеннями.

З одного боку відкрите ліцензування бібліотеки полегшує інтеграцію та модифікацію коду з іншого — воно може поставити під сумнів безпеку

використання у корпоративному середовищі. Це пов’язано з недостатньою увагою до перевірки

Втім вибір бібліотеки визначає і якість підтримки та оновлень у довгостроковій перспективі. Є сенс поставити запитання наскільки часто випускаються патчі чи фікси безпеки чи як швидко вирішуються питання у репозиторії. У контексті нашого проєкту коли фітнес-клуби покладаються на стабільність системи в години пік невчасні оновлення можуть призвести до простою і втрати довіри клієнтів.

Вибір бібліотеки — не формальність а стратегічне рішення яке впливає на продуктивність підтримку масштабованість і безпеку системи. Уникаючи шаблонного підходу до цієї задачі розробник отримує можливість створити довговічну архітектуру що відповідатиме вимогам локального ринку і буде гнучкою до майбутніх змін. Далі буде розглянуто дві популярні фреймворки, що сумісні з мовою TypeScript та мають підтримку СУБД PostgreSQL і MikroORM.

2.4.1 NestJS

NestJS виник як відповідь на потребу в більш структурованому підході до побудови серверних додатків у середовищі Node.js і TypeScript. Автори надихалися архітектурою Angular і перенесли її принципи на бекенд. NestJS базується на модульній архітектурі, приклад якої зображено на рис. 2.7. Вона дозволяє організовувати код у вигляді окремих контекстів зі своїми контролерами і провайдерами [18]. Така модель спрощує підтримку масштабних проєктів, з іншого допомагає уникнути хаосу в структурі коду.

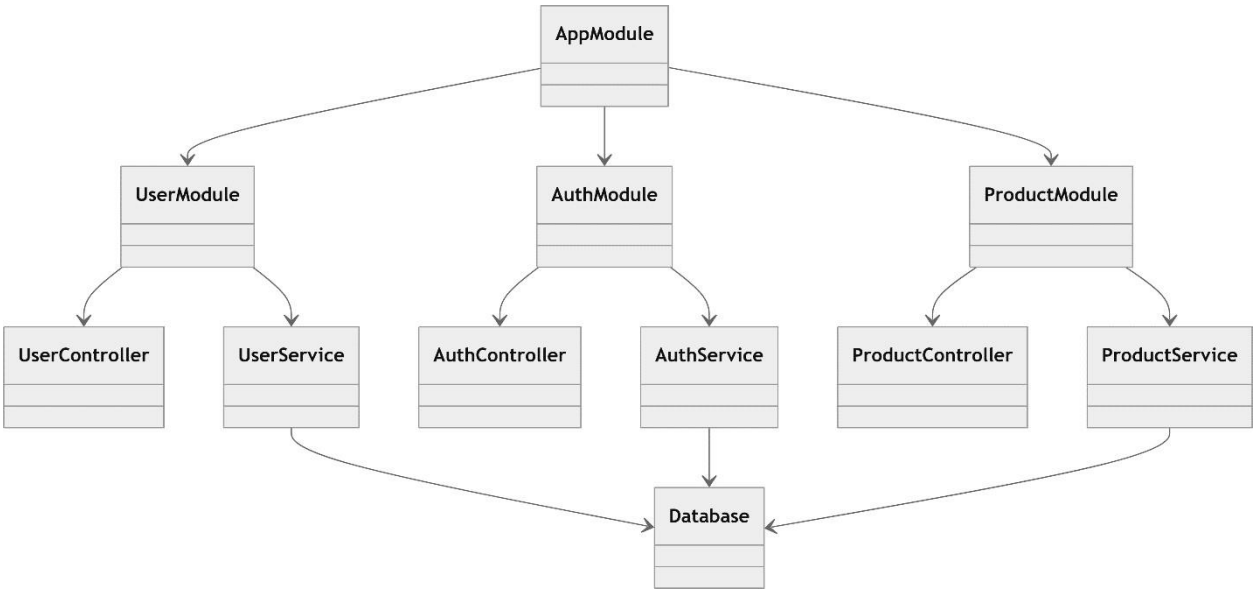


Рисунок 2.7 – Приклад архітектури веб-застосунку на NestJS

Варто розглянути систему впровадження залежностей яка є ключовою в NestJS і сприяє слабкому зв’язуванню модулів. Система інверсії керування реалізована через декоратори що полегшує тестування й повторне використання сервісів [19]. Можливо це пов’язано з тим що такий підхід сприятливий для великих команд розробників де треба чітко визначити контракти між частинами системи.

Контролери в NestJS відіграють роль шлюзу для обробки HTTP-запитів і не містять бізнес-логіки. Сервіси беруть на себе завдання обробки даних і взаємодії з базою [20]. Таке розділення сприяє чистоті коду та спрощує написання юніт-тестів.

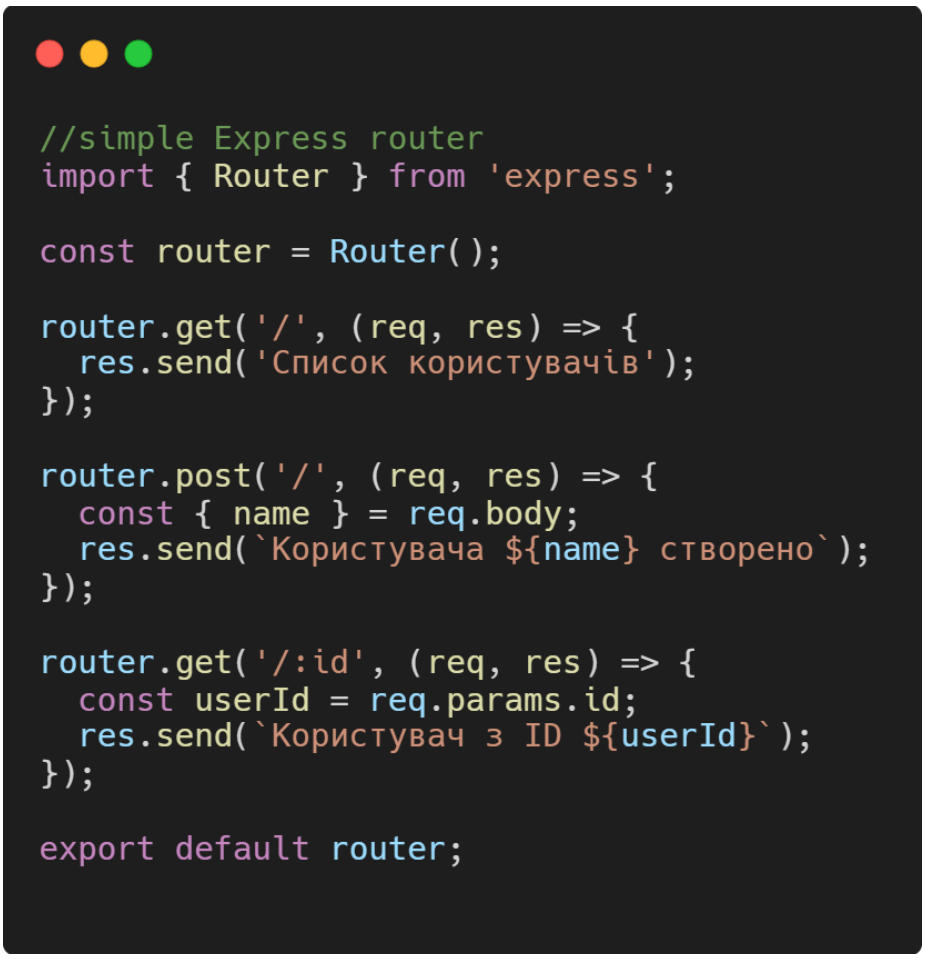
NestJS використовує можливості TypeScript на повну силу включно з інтерфейсами, що підвищує надійність коду, статична типізація допомагає уникнути багатьох помилок ще на етапі компіляції.

Втім у цьому середовищі можуть виникнути певні обмеження через необхідність вивчення специфіки framework-орієнтованої розробки. Автори оцінили цінність чіткого поділу шарів архітектури і реалізували це в NestJS за допомогою модулів.

NestJS підходить для середніх і великих проєктів де важливі стандарти і є бажання масштабувати систему без значних витрат часу на рефакторинг. Фреймворк вимагає певного оволодіння архітектурними концепціями, але значно підвищує продуктивність командної розробки і якість кінцевого продукту. Він займає гідне місце серед сучасних server-side рішень завдяки поєднанню найкращих практик TypeScript і перевірених підходів модульної архітектури.

2.4.2 Express.js

Express.js виник як відповідь на потребу в максимально легковагому HTTP-фреймворку для Node.js із мінімальними абстракціями та гнучкою структурою коду. Автори прагнули надати розробникам свободу вибору архітектурних підходів без нав’язування жорстких схем та патернів. Фреймворк пропонує систему маршрутизації основану на послідовному описі шляхів і методів обробки запитів що робить розробку читабельнішою та інтуїтивно зрозумілою [21]. Приклад програмного коду машрутизатора зображено на рис. 2.8.



```
//simple Express router
import { Router } from 'express';

const router = Router();

router.get('/', (req, res) => {
  res.send('Список користувачів');
});

router.post('/', (req, res) => {
  const { name } = req.body;
  res.send(`Користувача ${name} створено`);
});

router.get('/:id', (req, res) => {
  const userId = req.params.id;
  res.send(`Користувач з ID ${userId}`);
});

export default router;
```

Рисунок 2.8 – Приклад простого маршрутизатора Express.js

Варто звернути увагу на концепцію посередників , що представляють собою функції здатні обробляти запити на різних етапах конвеєра викликів [22]. Така модель дозволяє впроваджувати кросс-функціональні рішення наприклад логування авторизацію обробку помилок у єдиному потоці коду. Таким чином автори надали розробникам можливість створювати власні плагіни без необхідності втручатися в ядро фреймворку. Використання посередників спрощує введення нових функціональних блоків без зміни базових частин програми.

Express.js зберігає мінімалізм інтерфейсу і не містить вбудованих ORM систем кешування. Це пов’язано з прагненням уникнути надмірної ваги фреймворку і дозволити інтегрувати лише ті бібліотеки що дійсно потрібні для

конкретного проєкту. Фреймворк надає вбудовану підтримку обробки помилок через спеціальних посередників, що приймають чотири параметри і виконуються лише коли виникає помилка [3]. Це підвищує надійність коду і допомагає централізувати логіку обробки виключень.

Втім у цьому середовищі можуть виникнути певні обмеження пов’язані з реалізацією таких функцій як стиснення відповіді кешування на стороні сервера або підтримка вебсокетів. Express.js підходить для проєктів де цінують простоту створення сервісу швидкий час виходу на ринок і легку крива навчання нових учасників команди. Фреймворк охоплює широкий спектр прикладів від базової конфігурації до складної маршрутизації що значно полегшує старт нових розробників. Робота з Express.js не вимагає глибоких знань про внутрішні механізми Node.js хоча іноді доводиться вдаватися до дослідження кодової бази в пошуках оптимальних рішень для масштабування або безпеки.

2.4.3 Підсумок, щодо вибору бібліотеки

Вибір фреймворку для розробки потребує зваженого підходу, адже від архітектурної основи залежить не лише швидкість впровадження нових функцій, а й подальша масштабованість проєкту. Було розглянуто два можливих рішення – легкому Express.js та структурованому NestJS. З одного боку Express.js приваблює мінімалізмом і свободою вибору компонентів, з іншого – така гнучкість може ускладнювати підтримку кодової бази в довгостроковій перспективі. NestJS спирається на модульність і чіткі патерни, що може спочатку здатися зайвою вагою, втім у великих командах та проєктах саме така дисципліна запобігає хаосу й прискорює адаптацію нових учасників.

NestJS пропонує розбиття на модулі та провайдери, яке забезпечує чітке розмежування відповідальностей і спрощує орієнтування в коді. Контролери в даному фреймворку відповідають лише за обробку запитів, а сервіси беруть на

себе всю бізнес-логіку. При наявності десятків сервісів та контролерів така архітектура знижує ризик перетину обов’язків і виникнення побічних ефектів.

Система інверсії керування у NestJS значно спрощує тестування та повторне використання компонентів, а застосування декораторів для ін’єкції залежностей зменшує кількість «жорстко прописаних» посилань у коді.

NestJS має вбудовану підтримку багатьох архітектур та різних транспортних шарів, серед яких HTTP і WebSockets. Саме через це продукт легко адаптується до гібридних сценаріїв, коли частина логіки обробляється синхронно, а частина – асинхронно. Така універсальність корисна для застосунків, де необхідно оперативно надсилати сповіщення про майбутні події та одночасно обробляти велику кількість запитів до бази даних.

Активне використання TypeScript у NestJS сприяє підвищенню надійності та зрозумілості коду. Це пов’язано з тим, що статична типізація дозволяє виявити помилки ще на етапі компіляції. NestJS може здатися складним для невеликих проєктів через необхідність розуміти принципи модульності. З іншого боку для середніх і великих команд саме така структура стає перевагою, оскільки уніфікує підходи до організації коду.

Express.js залишатиметься популярним у невеликих сервісах, де критичним є час виходу на ринок і мінімальні залежності, з іншого – NestJS демонструє переваги у проєктах, яким необхідно забезпечити стабільність, масштабованість і гнучку інтеграцію з різними частинами системи.

Враховуючи все вище сказане, наявність модульної архітектури, переваги TypeScript, вибір на користь NestJS виглядає обґрунтованим. Він дозволяє поєднати стандарти корпоративної розробки із швидким темпом впровадження нових функцій. NestJS стане надійним «ядром» для уніфікованої серверної частини фітнес-застосунку, гарантуючи баланс між гнучкістю та структурованістю проєкту.

2.5 Допоміжні технології

2.5.1 RedisIO

RedisIO це низькорівневий інтерфейс взаємодії з Redis через стандартні команди зчитування та запису даних. Головною мотивацією використання RedisIO є прагнення до максимально швидкого доступу до часто використовуваних об’єктів у пам’яті, оскільки Redis зберігає всі дані в оперативній пам’яті й надає інтерфейс для миттєвого зчитування та запису через прості текстові протокольні повідомлення.

Втім, сам процес кешування за допомогою RedisIO базується на комбінації команд GET і SET із встановленням часу життя ключів через EXPIRE або вбудованими параметрами команд. RedisIO також підтримує конвеєри за допомогою яких кілька команд відправляються в одному пакеті, а відповідь отримується теж одним блоком. Це зменшує витрати часу і оптимізує продуктивність при серійних операціях запису чи зчитування.

Окремо варто звернути увагу на можливість атомарних операцій, які запускаються командою EVAL. Таким чином можна реалізувати складні патерни кешування, наприклад, одночасне перевірка наявності ключа й оновлення його значення, не порушуючи консистентності.

Втім, не можна не врахувати проблему «холодного старту», коли після очищення TTL одночасні запити можуть створити пікове навантаження на базу даних. Щоб налаштувати RedisIO у ролі кешу потрібно підібрати оптимальні параметри TTL, стратегію видалення та обсяг даних у кеші.

Підсумовуючи, можна ствердити, що RedisIO забезпечує універсальну низьколатентну платформу для кешування, що спрощує архітектуру серверної частини і підвищує швидкодію застосунків.

2.5.2 Websocket

WebSocket втілює двосторонній канал зв’язку поверх одного TCP-з’єднання, що дає змогу клієнту та серверу обмінюватися повідомленнями в режимі реального часу без необхідності постійно відкривати нові HTTP-запити. Ця здатність підтримувати довготривалі з’єднання робить WebSocket привабливим для застосунків, де важлива миттєва передача даних.

У практичній реалізації найчастіше виокремлюють два ключові аспекти роботи з WebSocket: встановлення з’єднання і обмін даними. Під час рукопотискання клієнт відправляє серверу HTTP-запит із заголовком Upgrade: websocket і спеціальним ключем, після чого сервер, підтвердивши можливість обміну WebSocket-повідомленнями, відправляє відповідь із кодом 101 Switching Protocols та власним ключем відповіді.

Після встановлення з’єднання WebSocket використовує власний формат фреймів, що дозволяє передавати текстові та бінарні повідомлення, здійснювати контрольні запити та коректно закривати з’єднання через обмін фреймами закриття. Така підтримка контрольних фреймів запобігає випадковим розривам зв’язку — клієнт чи сервер може періодично надсилати фрейм типу ping, а інша сторона відповідає pong, підтверджуючи живе з’єднання.

Інтеграція WebSocket на стороні Node.js зазвичай здійснюється через бібліотеку WebSockets, яка підтримує ті самі події та методи, що й браузерний API, приклад надсилання повідомлення за його допомогою зображено на рис. 2.9.

```
// server.js
const WebSocket = require('ws');

const wss = new WebSocket.Server({ port: 8080 });

wss.on('connection', function connection(ws) {
  console.log('Client connected');

  ws.on('message', function incoming(message) {
    console.log('Received:', message);
    ws.send(`You said: ${message}`);
  });

  ws.send('Welcome to the WebSocket server!');
});

// Connect to the server
const socket = new WebSocket('ws://localhost:8080');

socket.onopen = () => {
  console.log('Connection established');
  socket.send('Hello server!');
};

socket.onmessage = (event) => {
  console.log('Server responded:', event.data);
};
```

Рисунок 2.9 – Приклад використання WebSocket в Node.js

В експлуатації WebSocket треба враховувати низку обмежень. Оскільки з’єднання тримається відкритим, кожен клієнт споживає ресурси сервера, отже реальна кількість одночасних з’єднань обмежена апаратними можливостями та налаштуваннями ОС. Для великих проєктів доцільно застосовувати балансування навантаження на рівні TCP та механізми горизонтального масштабування серверів, щоб уникнути перевантаження одного екземпляру. WebSocket забезпечує надійну, двосторонню та низьколатентну комунікацію між клієнтом і сервером.

2.5.3 Swagger

Swagger є потужним інструментом для опису та документування REST-API за допомогою специфікації OpenAPI. Його популярність зумовлена прагненням розробників до уніфікації формату опису інтерфейсів і автоматичного генерування документації, приклад зображено на рис. 2.10, що значно спрощує підтримку та розвиток сервісів. Інструментарій Swagger ґрунтується на JSON або YAML форматі для опису маршрутів, параметрів запитів і відповідей, а також авторизації й прикладів даних згідно зі специфікацією OpenAPI версії 3 1.

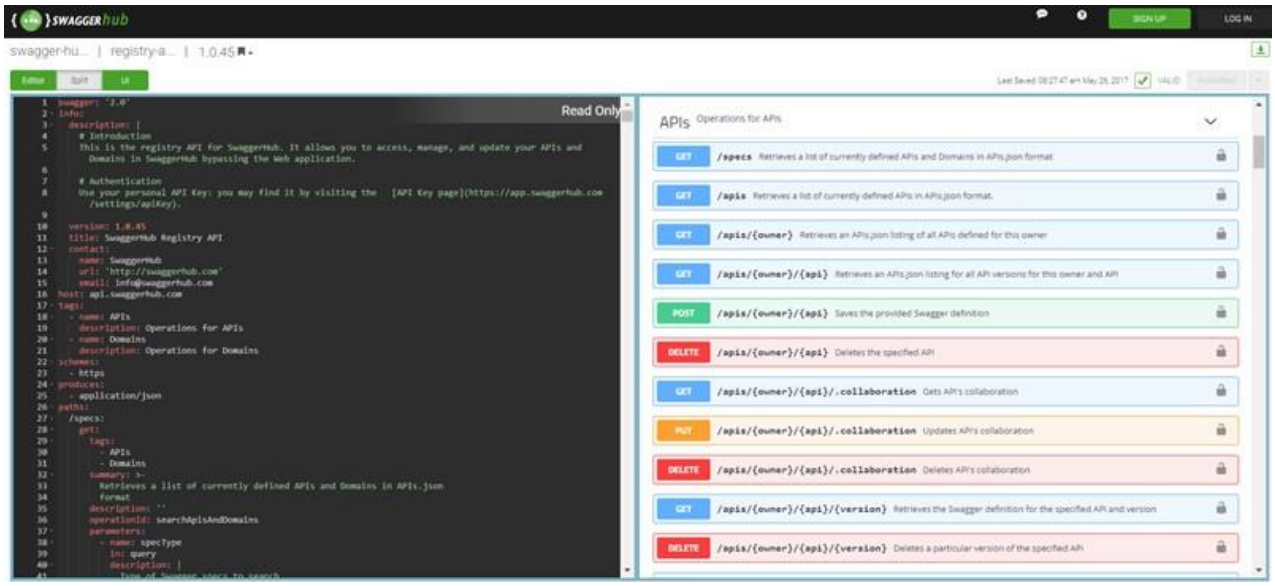


Рисунок 2.10 – Зовнішій вигляд документації в Swagger [24]

Ядро екосистеми Swagger складається з трьох основних компонентів. Перший компонент виконують бібліотеки для генерування клієнтського і серверного коду зі специфікацій, що дозволяє швидко запустити роботу над новим мікросервісом. Другий компонент забезпечує інтерактивний інтерфейс для перегляду й тестування API у браузері через Swagger UI, який підтягує файл з описом API та відтворює дружній до користувача інтерфейс. Третій компонент представлений Swagger Editor, де можна редагувати YAML або

JSON опис у реальному часі та відразу бачити результати вбудованого перегляду документації.

Впровадження Swagger в існуючий проект починається зі створення файлу openapi.yaml або openapi.json у корені репозиторію. У ньому необхідно задати загальні метадані про сервіс такі як назва, версія та контактні дані розробника. Далі слід описати шляхи доступу до ресурсів, вказавши HTTP-метод, параметри запитів, типи даних у тілі запиту та можливі відповіді зі статусами. У практичному застосуванні можуть виникати питання щодо сумісності описів зі специфічними фреймворками.

Одним із вагомих плюсів використання Swagger є прозорість і єдиність підходу до опису API незалежно від використовуваних технологій. Документація створюється одночасно з розробкою функціоналу і не вимагає розробки окремих конвертерів або ручного оновлення сторінок API. Крім того, завдяки підтримці інструментів для валідатора OpenAPI можна автоматично перевіряти коректність опису на відповідність схемам при кожному коміті. Ця інтеграція сприяє ранньому виявленню розбіжностей між реалізацією й описом інтерфейсу.

Swagger широко застосовується у корпоративному IT-середовищі, де важливо дотримуватися стандартів і підтримувати єдину практику розробки сервісів. З часом команді розробників стає зрозумілою структура точок входу і вони витрачають менше часу на уточнення запитів до документації, оскільки всі параметри, формати і помилки вже описані у файлі специфікації.

Swagger виступає надійним засобом уніфікації опису REST-API та забезпечує прозорий обмін інформацією між усіма учасниками розробки. Застосування Swagger допоможе підтримувати єдину логіку маршрутів, зменшить ризик помилок.

ВИСНОВОК ДО РОЗДІЛУ 2

Проведений аналіз довів, що справжня продуктивність народжується з узгодженого технологічного ансамблю, а не з випадкової колекції бібліотек. З огляду на це ядро майбутньої платформи сформовано довкола мови TypeScript, фреймворку NestJS, реляційної PostgreSQL та MikroORM, які разом забезпечують чіткий поділ відповідальностей і просту еволюцію бізнес-логіки.

Статична типізація TypeScript значно знижує ризик прихованих помилок у фінансово чутливих сценаріях, тоді як сумісність із JavaScript-екосистемою зберігає доступ до тисяч перевірених пакетів. NestJS, спираючись на інверсію залежностей і модульну структуру, уникає хаотичного «склеювання» залежностей, що характерне для «чистого» Express, та полегшує тестування через вбудовані механізми моків і пайпів.

PostgreSQL обрана через транзакційну надійність, розширювану систему типів і активну спільноту. Саме реляційна модель гарантує цілісність, коли йдеться про проведення платежів. Варто зазначити, що JSONB і матеріалізовані уявлення дозволяють поєднати структуровані та напівструктуровані дані без додаткових шарів перетворення.

MikroORM виступає тонким, але потужним мостом між об’єктами предметної області та базою даних. Його гібридний підхід поєднує переваги Active Record та Data Mapper, залишаючи за розробником повний контроль над запитами. Таким чином знижується кількість шаблонного коду, і при цьому не страждає продуктивність, адже SQL усе ще доступний там, де потрібна оптимізація.

Допоміжні технології логічно підсилюють основу. Redis працює як кеш, WebSockets забезпечують надсилання нагадувань. Swagger утворює живу документацію, зменшуючи часові витрати на інтеграцію мобільних клієнтів. Jest слугує каркасом для модульних та інтеграційних тестів.

Також важливо підкреслити, що всі компоненти підпорядковані парадигмі клієнт-сервер. Сервер концентрує автентифікацію, платежі, нагадування та аналітику; клієнтські застосунки залишаються тонкими, адаптивними та легко брендуються. З одного боку, мікросервісне розділення могло б покращити масштабування відокремлених доменів, з іншого — додаткова складність на ранньому етапі не виправдана. Тому зважений моноліт із чітко відокремленими модулями наразі видається найкращим компромісом між швидким запуском і подальшою еволюцією.

Підсумовуючи, можна стверджувати, що обраний стек утворює цілісну, притомну й водночас гнучку конфігурацію. Вона дозволяє сконцентрувати ресурси команди на розвитку справді цінного функціоналу замість боротьби з несумісними інструментами. Таке поєднання забезпечить баланс швидкості розробки, надійності та майбутнього масштабування, створюючи міцний фундамент.

РОЗДІЛ 3
ДЕТАЛІ РОЗРОБКИ СИСТЕМИ

3.1 Інфраструктура розробленого застосунку

Серверна частина розробленого застосунку для організації роботи фітнес-клубів поєднує низку взаємопов’язаних шарів, кожен із яких відповідає за певну сферу функціональності та забезпечує масштабованість, безпеку й гнучкість системи. Ядром цієї екосистеми є фреймворк NestJS, котрий своєю сприяє чіткому розподілу відповідальностей між компонентами застосунку. Саме така архітектурна модель дозволила уникнути хаотичного розростання коду в умовах частих змін вимог. Високий рівень абстракції MikroORM забезпечує просте описування сутностей через класи мовою TypeScript. Завдяки цьому оновлення структури таблиць перетворюється на передбачувану процедуру, адже всі зміни проходять через механізми міграцій, сформовані відповідно до конфігурації у файлі mikro-orm.config.ts, зображеного на рис. 3.1. Це полегшує роботу розробників, та гарантує підтримку консистентності даних у використаній реляційній базі даних.

```
import { defineConfig } from '@mikro-orm/core';
import { PostgreSqlDriver } from '@mikro-orm/postgresql';
import { config as envConfig } from 'dotenv';

envConfig();

export default defineConfig({
  entities: ['dist/**/*.entity.js'],
  entitiesTs: ['src/**/*.entity.ts'],
  dbName: process.env.DB_NAME,
  user: process.env.DB_USER,
  password: process.env.DB_PASSWORD,
  host: process.env.DB_HOST,
  port: Number(process.env.DB_PORT),
  driver: PostgreSqlDriver,
  allowGlobalContext: true,
  migrations: {
    tableName: 'orm_migrations',
    path: 'src/database/migrations',
    transactional: true,
    disableForeignKeys: true,
    allOrNothing: true,
    dropTables: true,
    safe: false,
    emit: 'ts',
  },
});
```

Рисунок 3.1 – Конфігураційний файл MikroORM

У контексті перемикань між середовищами розробки та експлуатації ваги набуває можливість читання параметрів підключення до бази без зміни коду. Зазначена практика дає змогу використати налаштування змінних оточення у окремому файлі і легко переключатися між локальними, тестовими та робочими середовищами.

На рівень вище працює шар кешування, реалізований як окремий модуль із використанням Redis. Саме кешування стало критичним елементом для тих сценаріїв, коли статистичні звіти або підсумки тренувальних циклів

обчислюються багаторазово. Як тільки подібний результат утворюється вперше, підсистема кешування записує його в Redis із певним часом життя.

Згодом, коли повторний запит потребує тієї самої інформації, дані дістаються з кешу без звернення до бази, що значно скорочує час відповіді й знижує навантаження на основне сховище. Структура кешу організована так, аби уникнути занадто довгого зберігання результатів, адже в умовах пікових навантажень застарілі дані можуть призвести до нерелевантних звітів. Утім, ґрунтуючись на налаштуваннях, можна регулювати час життя ключів, щоб забезпечити баланс між продуктивністю та актуальністю інформації.

У питанні безпеки застосунку спостерігається поєднання двох рівнів перевірок. Перший рівень базується на механізмі видачі підписаного JWT. Другий — це механізм охоронців, котрі перевіряють кожен вхідний запит на відповідність правам користувача щодо певного контролера чи сервісу. Наприклад, AuthGuard ініціює попередню перевірку наявності та валідності токена, а RoleGuard порівнює ролі з політиками доступу, визначеними декларативно. Така двохрівнева перевірка дозволяє розробникам додавати нові точки доступу, просто дописуючи їхню політику доступу, без дублювання логіки авторизації.

Окрему увагу приділено інтеграції з платіжною системою Монобанк. Було реалізовано сервіс-адаптер, який створює рахунки для оплати клубного абонемента, надсилає запити на зовнішній інтерфейс та отримує вебхуки про статус платежу. З іншого боку, обробка вебхуків виконується через унікальний MonobankWebhookGuard, котрий верифікує підписи від Монобанку. Це забезпечує високий рівень довіри до платіжних подій, адже навіть якщо вебхук від Монобанку опубліковано, перевіряється справжність підпису й лише потім передає дані далі. Підсистему платіжних операцій відокремлено від бізнес-логіки абонементів. Завдяки цьому підсистема клубного членства працює з полями, знайомими лише йому, а логіка оплати залишається ізольованою. Якщо

колись виникне необхідність підключити альтернативний банк, можна буде реалізувати ще один адаптер, не втручаючись у загальний сервісний код.

Ключовим елементом для зручності розробники і тестування стала автоматична генерація документації інтерфейсу за допомогою Swagger. Під час запуску застосунок аналізує всі модулі та контролери, формує відкритий опис ендпоінтів і розгортає інтерактивний інтерфейс. В ньому можна одразу перевірити параметри запиту, переглянути зразок відповіді та навіть авторизуватися, щоб протестувати захищені точки доступу. Створена графічна оболонка робить процес тестування значно легшим, адже розробнику не потрібно писати окремі клієнтські запити: достатньо відкрити сторінку Swagger UI, виконати запит і побачити результат.

Поділ на окремі підсистеми дозволяє розробникам працювати паралельно, не заважаючи одне одному. Наприклад, розробка механізму нагадувань про тренування не впливає на код, що відповідає за збір статистики, адже вони взаємодіють через базовий транзакційний сервіс, який реалізує атомарні операції над сутностями. Наявність цього єдиного «нервового центра» гарантує цілісність даних навіть тоді, коли кілька модулів одночасно змінюють стан системи. Менеджер сутностей, що бере участь у транзакціях, забезпечує збереження змін у базовому сховищі, а також керує кешуванням для прискорення повторних запитів.

Отже, інфраструктура побудована навколо зрозумілих і перевірених практик: суворе розмежування відповідальностей між шаром підсистем, інтерфейсів та сховищ, використання ORM для абстракції SQL-операцій, Redis для кешування. Таким чином система демонструє високий рівень модульності, можливість швидкого масштабування під піки навантажень і простоту інтеграції нових функцій без втручання в готовий програмний код.

3.2 Підсистема безпеки

Підсистему аутентифікації та авторизації в цьому застосунку було спроектовано як самостійну логічну підсистему що охоплює всі рівні роботи застосунку. Він не є просто побіжним доповненням до бізнес-логіки а навпаки пронизує всі інші підсистеми та компоненти, що відповідають за виконання операцій. Це пов’язано з прагненням уникнути фрагментарного підходу коли перевірка прав доступу існує лише на окремих етапах і в результаті виникають прогалини в безпеці.

Крім даних про себе, користувач вказує його електронну адресу та пароль. Вважається необхідним щоб кожна адреса використовувалась тільки одним обліковим записом щоб уникнути дублювання. Пароль під час реєстрації зазнає хешування з додаванням унікальної солі для кожного користувача. Використання надійного алгоритму хешування водночас із генерацією унікальної солі перетворює навіть однакові паролі на різні хеші що значно знижує можливість їх успішного підбору. Уявімо що зловмисник отримав доступ до бази даних та побачив хеші в поєднанні з унікальними солями динамічно згенерованими для кожної обліковки тоді навіть застосування апаратно оптимізованих методів підбору потребуватиме надзвичайних обчислювальних ресурсів.

З вище сказаного зрозуміло, що через те, що пошта має бути унікальною потребує відповідної перевірки. Якщо знайдено запис що відповідає вказаній адресі тоді система формує відповідь що не розкриває інформацію про наявність користувача у базі даних а просто вказує на помилковість реєстраційних даних. Завдяки такому підходу потенційний недоброзичливець не отримує жодного натяку який із кроків уведення даних виявився невірним.

У разі успішного створення нового облікового запису або під час входу користувача здійснюється генерація унікального підписаного JWT токену, приклад якого зображено на рис. 3.2.

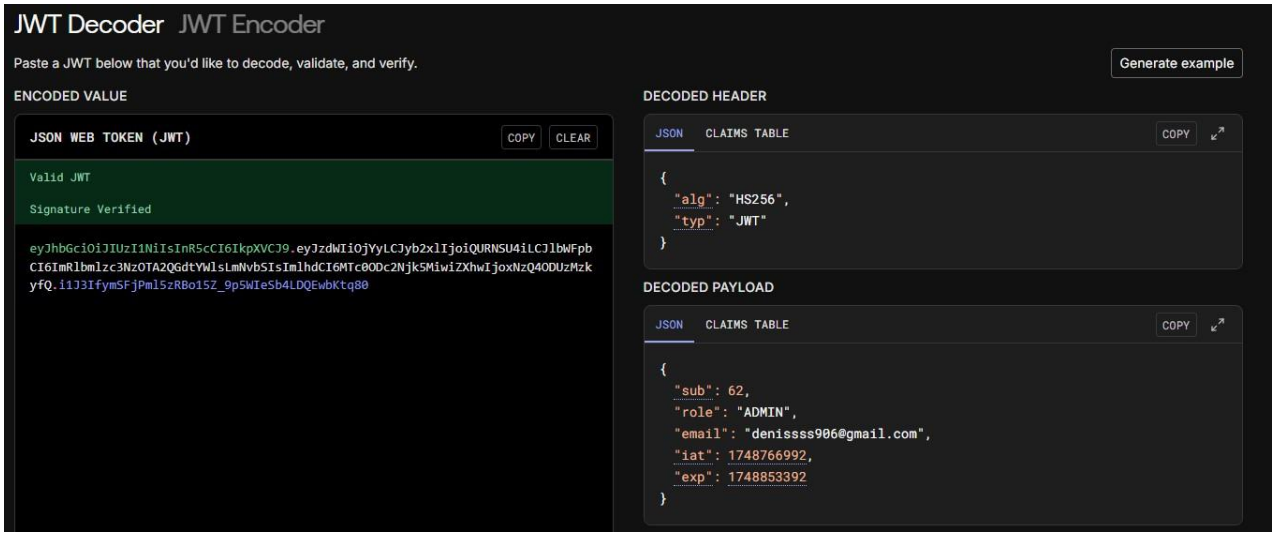


Рисунок 3.2 – Приклад JWT токену

Він формується так щоб містити мінімум необхідних даних що включають унікальний ідентифікатор облікового запису саму роль користувача й час дії маркера. Секретна фраза яку використовують для підпису не знаходиться безпосередньо в коді а підтягується із змінних середовища. Це дозволяє легко змінювати її без перекомпіляції і зменшує ризики компрометації в разі витоку вихідних файлів. Термін життя токена також задається конфігураційно і може бути налаштований відповідно до бізнес-вимог наприклад становити добу чи менше.

Під час перевірки аутентифікаційних запитів система очікує маркер у заголовку із певним префіксом. Як тільки маркер отримано відбувається його верифікація яка включає одразу кілька кроків. По–перше система переконується що маркер не прострочений що жодна бічна зміна в ньому не відбулася і що алгоритм підпису відповідає очікуваному. По–друге загальна цілісність маркера підтверджується шляхом перевірки за допомогою секретної фрази. Втім якщо на будь–якому з цих кроків виникає невідповідність

подальший рух запиту припиняється і клієнт отримує повідомлення про несанкціонований доступ. Це підвищує довіру до системи бо навіть фальшива спроба позмінити час дії або алгоритм підпису миттєво призведе до відмови.

У випадку якщо маркер є валідним, перед виконанням операції здійснюється додаткова перевірка ролі. Пункт доступу до певної функціональності супроводжується декларативним описом необхідних прав. Під час ініціалізації запиту перевіряючий компонент зіставляє роль закодовану в маркері із роллю що зазначена поряд із самим маршрутом. Якщо користувач не має достатніх прав відбувається відмова у доступі з відповідною помилкою. Така декларативна модель дозволяє розробникам одразу бачити які ролі потрібні для виконання операції а також уникнути дублювання коду перевірки прав у багатьох місцях. Завдяки цьому зберегти код чистим і зрозумілим навіть у разі активного розвитку функціоналу стає набагато простіше.

У блоці обробки платіжних повідомлень що надходять від зовнішньої платіжної платформи реалізовано окрему перевірку на коректність цифрового підпису. Спершу аналізується заголовок або тіло повідомлення де містяться дані про платіж та відповідні криптографічні атрибути. Після цього спеціальний алгоритм верифікує підпис використовуючи публічний ключ платіжної системи що також надається через одне із середовищ оточення. Якщо підпис не відповідає очікуваним значенням запит відразу відхиляється. Водночас сама бізнес–логіка заробітку абонемента отримує лише ті поля що пройшли процес нормалізації і вже не тягнуть за собою надмірного навантаження на основний код. Це означає що навіть якщо зловмисник спробує підмінити повідомлення швидше за все він не матиме валідного цифрового підпису а відповідно не зможе змінити стан внутрішніх сутностей.

Окрему увагу було приділено тому щоб у відповіді клієнту не відправляти конфіденційну інформацію з бази. Отже перед надсиланням даних видаляються або явно приховуються поля що містять хеші паролів унікальні солі й інші приватні атрибути. Ця поведінка реалізована на рівні перетворення об’єктів

перед серіалізацією таким чином що відповідь завжди містить лише безпечний набір даних. Якщо не приділити належної уваги такому аспекту можна випадково розкрити чутливу інформацію що грубо порушує принцип розділення обов'язків.

Паралельно з цим у підсистемі застосовано перевірку структури вхідних даних для захисту від некоректних запитів. Щоразу коли користувач передає дані відбувається їх валідація відповідно до заданих правил, наприклад перевірка формату адреси електронної пошти мінімальної довжини пароля чи обов'язковості певних полів. Якщо хоча б одне правило не дотримано клієнт отримує докладний опис помилки що значно спрощує налагодження запиту особливо під час тестування через веб-інтерфейс. Використання такого підходу допомагає усунути уразливості що виникають через неправильне форматування даних або спробу виконати операцію з неповними параметрами.

Також важливим є також те що всі налаштування безпеки а отже і секретні ключі для підпису токенів цифровий ключ для перевірки підписів платіжної платформи а також інші чутливі параметри розташовані поза межами відібраного коду. У результаті перенести застосунок між різними середовищами розробки тестування чи виробництва можна лише шляхом зміни значень у середовищі оточення. Це відповідає сучасним принципам розробки які передбачають чітке розмежування конфігурацій і вихідного коду, що дозволяє уникнути непередбачуваних наслідків під час розгортання.

Процес тестування аутентифікації та авторизації став надзвичайно зручним завдяки інтеграції з інструментом автоматичної генерації документації. Під час старту сервера система динамічно аналізує структуру точок доступу і формує так званий інтерфейс що дозволяє легко відправляти запити без потреби писати клієнтський код. Достатньо вказати отриманий раніше токен і виконувати операції через графічний інтерфейс. За рахунок цього розробники можуть одразу тестувати як процедуру входу так і подальшу взаємодію із захищеними маршрутами не залишаючи середовище програми. Це, у свою чергу, позитивно

впливає на швидкість виявлення можливих помилок або упущень у логіці перевірок.

Всі описані вище компоненти створюють цілісний ланцюжок, що починається з криптостійкого хешування паролів, потім переходить до реєстрації користувача. Він також включає токена в ході входу і забезпечує двоступеневу перевірку запитів: спочатку перевірка валідності токена, потім перевірка ролі. Додаючи до цього захист вебхуків платіжної платформи та приховане зберігання чутливих даних у відповіді, формується модель безпеки, що дозволяє розширювати функціональність без боязні залишити дірки що будуть використані для некоректних дій. Саме такий підхід гарантує гнучкість й надійність системи в процесі подальшого розвитку функцій фітнес-клубу.

3.3 Комплексне керування тренувальним процесом

Підсистема комплексного керування тренувальним процесом у розробленому застосунку створена як своєрідний нервовий центр, який акумулює не лише вхідні запити клієнтів, але й увесь ланцюг логіки від планування занять до постобробки результатів. Таку архітектуру важко уявити без чіткої багатошарової моделі, але її гнучкість полягає в тому, що кожен логічний шар реалізовано як окремий модуль, котрий відповідає за власну сферу відповідальності. Вони не взаємодіють напряму, а покладаються на спеціальний інтерфейс, що гарантує чітку взаємодію та мінімізує ризик «переплутати» бізнес-логіку з механізмом збереження даних чи кешування.

Планування тренувального процесу реалізовано наступним чином: запит користувача на створення заняття потрапляє у відповідний інтерфейс, який передає інформацію до сервісу планування. Далі відбувається низка послідовних перевірок: чи є тренер у базі, чи відповідає обраний тип тренування (індивідуальне чи групове) обраному формату, а також чи не виникає конфлікт

часу з вже запланованими заходами. Якщо клієнт обирає готовий набір вправ, система автоматично долучає до створюваного запису посилання на відповідний комплекс із каталогу. Такий механізм допомагає швидко присвоїти заняттю набір вправ, тоді як у випадку, якщо тренер бажає скласти авторський сценарій, застосунок дозволяє комбінувати базові елементи з різних наборів, створюючи нову програму. Після підтвердження усіх умов, дані про нове заняття фіксуються в основному сховищі даних, а одночасно дублікуються в кеш-шарі. За рахунок кешування можна організувати майже миттєвий пошук найближчих подій для конкретного користувача, адже кеш служить буфером для найчастіше запитуваних операцій.

Із наближенням часу початку тренування запускається окремий фоновий процес, котрий виконує роль планувальника сповіщень. Він періодично обходить таблицю запланованих тренувань і обчислює часовий відрізок від поточного моменту до початку кожного із занять. У разі, якщо йдеться про групове тренування, сповіщення розсилається всім учасникам такої групи, а якщо індивідуальне — лише одному клієнту. Цей механізм був обраний тому, що в умовах кількох одночасних групових секцій важливо вчасно попередити всіх учасників, і саме за рахунок фонових завдань, що використовують таймери та черги повідомлень, вдається організувати таку розсилку без значного навантаження на основний цикл обробки HTTP-запитів.

Коли тренування починається, тренер має можливість зафіксувати фактичний час старту та завершення заняття. Якщо клієнт відчуває втому або виникають інші обставини, тренер може оперативно скоригувати заплановану програму вправ. Всі закінчені тренування фіксуються в БД задля їх використання в подальших статистичних звітах, без їхньої фіксації ефективне відстеження прогресу було б неможливим. Крім того, реалізовано можливість позначити користувача як відсутнього чи присутнього, що поєднується з механізмом обробки статусу заняття: після завершення всієї сесії статус тренування змінюється з «запланованого» на «завершений» або «скасований».

Підсистема управління комплексними тренуваннями вирізняється тим, що компоненти не перетинаються прямими викликами; вони взаємодіють виключно через чіткі контракти. Це видається надзвичайно важливим, адже коли виникне потреба реалізувати підтримку нового більш специфічного функціоналу, досить буде створити новий сервіс і під'єднати його до існуючого ланцюжка взаємодій, не втручаючись у вже написану кодову базу. Фактичний тягар підрахунків перенесено з операційної бази: кеш-шар зберігає «гарячі» вибірки, не блокуючи транзакцій. Саме кешування дозволяє швидко отримувати найбільш актуальні дані, наприклад, список найближчих занять для користувача чи оновлений розклад тренера.

Нарешті, весь контроль доступу централізовано сконцентровано в окремому сервісі ролей. Коли з'являється потреба в додатковій ролі, достатньо лише оновити перелік дозволених ролей, без необхідності змінювати логіку методів, що обслуговують тренування. Це демонструє, що декларативний підхід до опису політики доступу працює доволі ефективно, оскільки будь-яка операція супроводжується вказівкою необхідної ролі, і в разі невідповідності запит не виконується, повертаючи клієнту відповідну помилку. Саме така сувора модель керованості і дозволяє системі залишатися прозорою для майбутньої підтримки й розвитку, не жертвуючи продуктивністю.

Збережені в базі дані про тренування аналізуються сервісом статистики, формуючи звіти про відвідуваність, прогрес кожного клієнта та результати тренера. Наприклад, за допомогою агрегованих запитів можна отримати кількість проведених тренувань певним тренером за обраний період, кількість унікальних учасників, а також розподіл між індивідуальними та груповими сесіями. Завдяки такому підходу адміністрація клубу може швидко оцінювати ефективність роботи тренерів та оптимізувати розклад, виходячи з реальних показників.

Описана підсистема комплексного керування тренувальним процесом поєднує в собі планування, сповіщення, моніторинг виконання тренувань і

детальну аналітику, реалізовані за допомогою локалізованих модулів. Це не лише сприяє технічній чистоті коду, але й забезпечує високий рівень підтримки та можливість швидкого розширення функціональності у майбутньому. Така структурована модель дозволяє системі зберігати гнучкість і керованість навіть за умов зростання навантажень і нарощування бізнес-вимог, водночас гарантуючи прозорість логіки для розробників і адміністраторів.

3.4 Каталог вправ і конструктор планів тренувань

Підсистема каталогу вправ і конструктора планів тренувань має ключове значення для всього життя фітнес-застосунку. Це місце, де акумулюється методична база тренувального процесу, кожна вправа описується детально з низкою атрибутів. Саме через конструктор планів тренувань формується послідовність вправ, яку надалі «прикріплюють» до конкретної сесії, тож від надійності й гнучкості цього блоку напряму залежить якість обслуговування клієнтів. Всі дані про набори вправ зберігаються в реляційній базі разом із чіткою версіонністю, а найбільш популярні вибірки одразу дублюються у кеш-шарі. Подібний підхід допомагає скоротити середній час відповіді й убезпечити систему від надмірних запитів до бази в пікові періоди.

На самому початку кожна вправа потрапляє до єдиного каталогу, який слугує єдиною довідковою базою рухів. Для кожного запису зберігаються такі атрибути, як можливі варіації техніки виконання у залежності від фізичних особливостей клієнта, а також стандартні значення для рекомендованої кількості підходів і повторень. Наприклад, для базових вправ зі штангою система може вказувати діапазон рекомендованих ваг і крок прогресії, що дає змогу тренеру становити програму поступового нарощування навантаження. Цікаво зазначити, що ще на етапі введення нового запису передбачено перевірку унікальності комбінації «назва + обладнання», аби уникнути

дублювання в каталозі й одночасно забезпечити чистоту даних. Видається логічним припущенням, що такий механізм дозволяє запобігти неоднозначностям, коли існують дуже схожі вправи, але з різними наборами інструментів.

Після того, як базовий опис вправи внесено до сховища, його обов'язково дублюють у кеш-шарі, якщо він належить до категорії популярних або найчастіше запитуваних. Кешування обраних вибірок (наприклад, усіх базових вправ зі штангою чи всіх кардіо-вправ середнього рівня складності) значно скорочує час обробки запитів, оскільки не потрібно додатково звертатися до основної бази даних. Втім, коли атрибут, що не є статичним для вправи, змінюється, система автоматично інвалідовує відповідні ключі в кеші, щоб уникнути ситуацій, коли тренер бачить застарілу інформацію. Таким чином підтримується цілісність даних.

У центрі підсистеми розташовано конструктор планів тренувань, що фактично уособлює собою інструмент для збирання з каталогу окремих вправ у єдину програму. Тренер, працюючи з цією системою, має можливість у вигляді поєднувати вибрані вправи у правильному порядку, задаючи для кожної конкретні параметри: кількість підходів, інтервали відпочинку, темп виконання та рекомендовані вагові показники. Завдяки такому підходу зберігається не лише цілісність статистики, а й прозорість еволюції програм, що є критичним при аналізі прогресу клієнтів чи оцінці ефективності методик. Якщо тренер пізніше модифікує вже існуючий набір, змінені параметри не впливатимуть на попередні дані результатів — за рахунок ревізій і збережених копій зрозуміло, які вправи й у яких умовах виконувалися клієнтом минулого тижня чи місяця.

Коли настав момент формування розкладу, конструктор планів інтегрується з підсистемою запланованих тренувань. У цей момент повний опис набору, включно з послідовністю вправ та їхніми атрибутами, «вбудовується» безпосередньо в сутність тренування. Завдяки цьому під час відображення деталей майбутньої сесії в користувацькому інтерфейсі не виникає необхідності

робити додаткові запити до каталогу чи конструктора: застосунок може одразу показати кожному клієнту докладний план. Якщо ж під час заняття тренер вирішує замінити окрему вправу, система створює нову тимчасову копію оригінального набору з відміткою про походження. Завдяки цьому зберігається достовірність ретроспективного аналізу: ніхто не змінює історичні дані, але водночас у поточній сесії клієнт отримує оновлену програму. Завдяки цьому уникається розмивання статистичних даних і адміністрація фітнес-клубу отримує змогу відслідковувати, як зміни в програмі впливали на результати відвідувачів.

Соціально-технічний аспект цієї підсистеми також не варто недооцінювати. Коли новий тренер починає працювати в клубі, достатньо надати йому доступ до конструктора — і він може одразу використовувати існуючі набори вправ, а також створювати власні, зберігаючи їх окремо. Це спрощує процес адаптації до внутрішніх процедур і дає змогу швидше вводити нові методики без довгих узгоджень. Саме завдяки такій організації вдалося досягти високої якості обслуговування клієнтів: тренери не витрачають зайвого часу на «тонке налаштування» інтерфейсу, а мають всю інформацію, тоді як клієнти отримують програму, яку можна адаптувати під індивідуальні особливості та навіть моментально змінити в разі потреби.

Підсистема каталогу вправ і конструктора планів тренувань є наочним прикладом того, як у розробленому застосунку поєднуються академічна строгость і практична гнучкість. З одного боку, чітка структура даних, ревізійна система та налаштоване кешування забезпечують високу продуктивність і точність статистики. З іншого — модульна архітектура й можливість легко розширити функціонал дозволяють адаптувати систему до нових викликів, наприклад, впровадити підтримку зовнішніх аналітичних сервісів чи навчальних матеріалів.

3.5 Платіжний функціонал

3.5.1 Клубне членство

Клубне членство у цьому застосунку спроектоване як окремий, незалежний функціональний модуль, призначений для комплексного управління фінансовими операціями та їх синхронізації з тренувальними процесами. Такий підхід обґрунтовано практичною необхідністю чітко розмежовувати відповідальності й не допускати потенційних збоїв, що можуть виникати через змішування різних завдань. Завдяки цьому можна досягти гнучкості в розвитку системи, підтримуючи легке розширення функціоналу та швидке реагування на зміни бізнес-процесів клубу.

Всі дані необхідні для управління підписками атрибути, такі як дати початку й завершення періоду підписки, унікальний ідентифікатор рахунку, статус членства та інформацію про здійснену оплату містяться в базі даних. Кожне запис має чітко визначений життєвий цикл, який проходить через стадії створення, активації, закінчення терміну дії та можливого скасування. Це дозволяє максимально чітко відстежувати поточний стан абонемента і легко реагувати на різні події в системі.

Під час першої оплати клієнтом, після отримання підтвердження від банківського інтерфейсу, система автоматично створює запис абонемента та надає йому статус «активний». Одразу ж визначаються дата початку дії та дата закінчення терміну абонемента, що дозволяє клієнту відразу використовувати всі доступні сервіси, наприклад, здійснювати бронювання тренувань. Усі подальші фінансові операції з цим абонементом чітко фіксуються в базі даних, завдяки чому адміністратор має змогу бачити історію оплат без необхідності ручних перевірок. Це дозволяє уникнути значних витрат часу на з’ясування фінансових питань та суттєво знижує ймовірність помилок.

Підсистема клубного членства ретельно спроектована з точки зору безпеки. Всі дії з членствами захищені кількома рівнями контролю доступу. По-перше, будь-яка дія починається з перевірки автентифікації користувача через JWT токен. Наступним кроком є перевірка відповідності прав, які вимагаються для здійснення конкретної операції, реальним правам користувача. Так, звичайний користувач має право лише на перегляд інформації про свої підписки, тоді як адміністратор може управляти абонементами інших користувачів — створювати, продовжувати чи припиняти їхню дію. Така централізація політики доступу дозволяє значно спростити підтримку та масштабування бізнес-логіки застосунку.

Було передбачено можливість швидкого та безпечного розширення тарифної матриці. Це означає, що коли клуб вводить новий тариф або спеціальну акцію, адміністратору достатньо просто додати новий тип членства у відповідний довідник і встановити необхідні параметри, після чого система автоматично враховуватиме нові правила без потреби в додаткових технічних діях. Така гнучкість є критично важливою для динамічних ринкових умов, оскільки клуб отримує змогу оперативно тестувати нові формати співпраці з клієнтами, залучаючи нові категорії відвідувачів та корпоративних клієнтів без ризику втручання у стабільність наявних бізнес-процесів.

Технічна реалізація цього модулю забезпечується завдяки чіткій розподіленості відповідальностей між ключовими компонентами. Завдяки цьому модуль клубного членства є максимально ізольованим та зручним для підтримки й розвитку.

3.5.2 Інтеграція з Monobank API

Інтеграція з Monobank реалізована як окрема підсистема, що стоїть поміж підсистемою платежів і зовнішнім банківським шлюзом. Його основна місія — зробити взаємодію з API банку прозорою для решти системи, сховавши під усі

нюанси підпису запитів, обробки вебхук-подій і повторних спроб у разі тимчасових збоїв мережі. Під’єднання до банку відбувається на рівні сервісу-адаптера. Він використовує параметрами з файлу серидовища. Таким чином жоден бізнес-клас не зберігає ключів безпосередньо в коді, а перемикання на тестовий контур зводиться до зміни двох-трьох змінних середовища.

Коли клієнт намагається сплатити абонемент, підсистема платежів звертається до сервісу-адаптера з потрібними параметрами. Адаптер формує запит до банківського API. Відповідь містить короткочасне посилання на сторінку оплати. Перейшовши по посиланню клієнт взаємодіє вже безпосередньо з Monobank, а сервер очікує асинхронний зворотний виклик. Щойно платіж завершено, банк надсилає запит на захищений маршрут застосунку. Перш ніж зберегти подію, перевіряється підпис: він порівнює передану контрольну суму із обчисленою локально, використовуючи раніше зчитаний таємний ключ. Невідповідність спричиняє негайну відмову, тож підробити повідомлення практично неможливо.

Після успішної перевірки зчитується тип події (успішна оплата, повернення коштів, відхилення), суму, валюту та ідентифікатор замовлення. Щоб уникнути повторної обробки, подія проходить через схему ідемпотентності: у таблиці зберігається унікальний хеш — якщо той уже є, обробка зупиняється. Далі керування переходить до підсистеми членства. Для випадку «успішно сплачено» він встановлює статус активний і фіксує новий діапазон дат.

Описаний алгоритм, зображена на рис. 3.3, працює не лише швидко, а й гнучко: якщо клуб вирішить додати ще один банк, знадобиться створити новий адаптер із власним підписом і веб-хуком, залишаючи незмінними інші підсистеми.

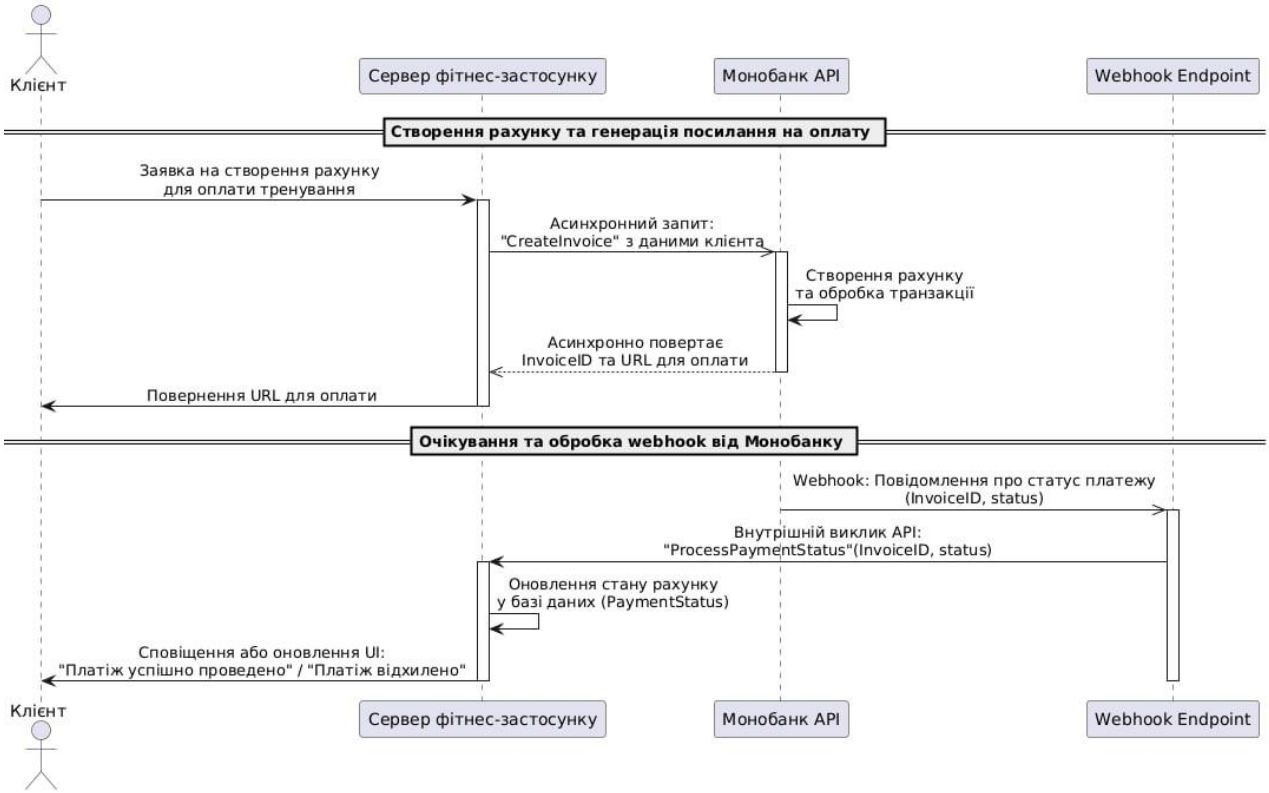


Рисунок 3.3 – Алгоритм обробки запитів пов’язаних з інтерфейсом Монобанк

Саме така сегментація, підкріплена додатковими перевірками та ідемпотентним шаром, мінімізує імовірність фінансових розбіжностей і робить платіжний контур надійним навіть у пікові години трафіку.

3.6 Аналітика та формування статистичних звітів

Підсистема аналітики у розробленому серверному застосунку постає самодостатньою, що тісно співпрацює з іншими підсистемами, проте лишається автономною в частині власних обчислень і політик зберігання. Її ядром є сервіс агрегування: він отримує лише минулі події а вже всередині виконує всю важку роботу зі зведення, ранжування та нормалізації даних.

Відвідуваність клієнтів, завантаженість тренерів, кількість відвідувачів у клубі – це приклади статистичних звітів. Для кожного напрямку підготуавні агрегати різного рівня: щоденні, тижневі, місячні. Тонке налаштування періодів є однією з переваг. Аби не блокувати транзакції в основній схемі, підсистема

використовує окремий менеджер запитів. Після завершення обробки сервіс «генерує» нові зведені дані.

Для повсякденого використання статистика має бути актуальною, але часто не надто детальної статистики: приміром, скільки тренувань відвідав користувач за минулий тиждень. Задля обробки подібних запитів підсистема отримує актуальну інформацію з БД, серіалізує відповідь у JSON і, перш ніж віддати, кешує на короткий проміжок. Отже, повторні звернення за однаковою вибіркою десятки разів швидші й не конфліктують із іншими обчисленнями.

Важливим елементом безпеки лишається перевірка ролей: показники персонального прогресу тренувань бачить лише власник та, за потреби, його тренер, статистику по клубу - адміністратор. Фактична перевірка ролей відбувається ще на рівні маршруту контролера, тож підсистема отримує тільки валідний, очищений від неавторизованого трафіку запит і може зосередитися на логіці формування відповіді. А щоб уникнути витоку персональних даних у системні журнали, усі ідентифікатори користувачів маскуються.

З архітектурного погляду описана підсистема лишає простір для подальшого зростання. Якщо знадобиться машинне навчання для прогнозу відтоку, достатньо додати новий сервіс-процес, який забиратиме зведені дані, будуватиме модель і повертатиме ймовірнісні оцінки. За такої потреби дана підсистема не змінюється: до існуючого API просто додається ще одне поле, що містить прогноз, а перевірка ролей, забезпечить те, що чутлива аналітика потраплятиме лише в потрібні руки.

Саме така багатошарова конструкція робить механізм аналітики надійним інструментом, що допоможе приймати рішення, не змушуючи команду жертвувати продуктивністю критично важливих сервісів під час найгарячіших годин роботи клубу.

3.7 Кешування й підвищення продуктивності

У межах підсистеми кешування реалізовано двошаровий підхід, що поєднує локальну пам’ять процесу й віддалений кластер Redis, завдяки чому сервер реагує швидко. Початковий запит до бази даних підхоплюється задля кешування. Формується ключ із сигнатури запиту, шукається збіг у локальному сховищі й, за відсутності, переходить до Redis. Якщо й там порожньо, дані беруться з бази даних, серіалізуються у JSON і зберігаються у два шари з різною тривалістю життя: пів хвилини в оперативній пам’яті й кілька хвилин у Redis-кластері. Така комбінація дозволяє упродовж коротких серій однакових звернень узагалі не торкатися мережі, а водночас зберегти консистентність між репліками, коли клієнтів стає надто багато.

Якщо користувач робить запит вже на отримання даних, то спочатку цей запит потрапляє в сховище Redis. Якщо необхідні дані вже присутні в оперативній пам’яті, Redis миттєво віддає їх клієнтові, й ланцюжок завершується без залучення БД. Коли ключ у кеші відсутній система вже направляє запит до БД. Цей алгоритм зображено на рис. 3.4.

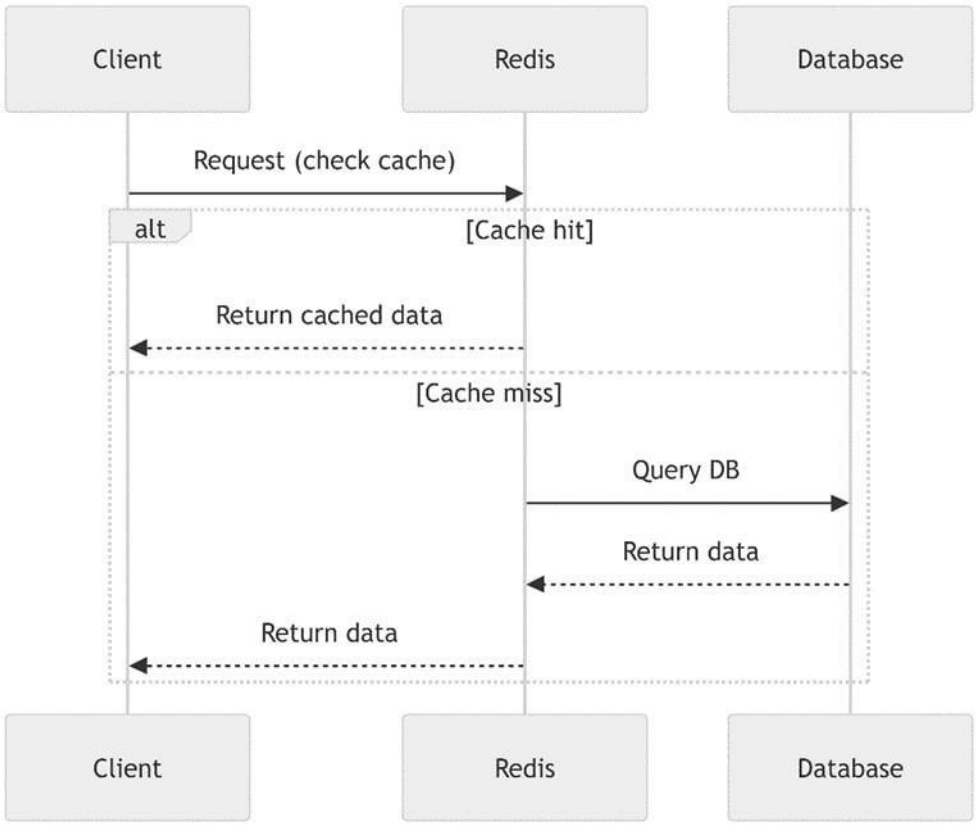


Рисунок 3.4 – Алгоритм обробки запитів з врахуванням шару кешування.

Операції, що змінюють стан генерують спеціальну подію. Цей сигнал поширюється внутрішнім шлюзом, тож ключі, що стосуються зміненого об’єкта, одразу зникають із оперативної пам’яті та Redis. Отже, наступне читання вже звернеться до бази й запише свіже значення. Така схематична «спостережуваність» запобігає ситуації, коли отримання застарілої інформації.

Результатом використання такої архітектури стало скорочення середнього часу відповіді й зниження навантаження на базу даних. При цьому жоден ключ доступу не зберігається в оперативній пам’яті процесу, що дозволяє безболісно горизонтально масштабуватися, запустивши додаткові репліки бекенду під балансувальником. Таким чином підсистема кешування виконує не лише роль пришвидшувача, а й основу еластичності всієї серверної платформи.

ВИСНОВОК ДО РОЗДІЛУ 3

Розроблена серверна частина застосунку для організації роботи фітнес-клубів побудована на засадах комплексного, модульного підходу, що дозволило реалізувати ряд важливих технічних та функціональних переваг. Важливо зазначити, що у процесі проектування було приділено особливу увагу надійності, безпеці та гнучкості архітектури.

Однією з ключових переваг розробленої інфраструктури є чітке розмежування відповідальності між компонентами. Вибір NestJS як базового фреймворку дозволив використовувати принцип інверсії управління та забезпечив належну модульність, завдяки чому стало можливим уникнути дублювання коду й спростити інтеграцію нових функцій. Особливо це стосується механізмів авторизації й автентифікації, платіжних операцій, керування тренуваннями та аналітикою, що були реалізовані як автономні модулі з чітко визначеними інтерфейсами взаємодії. Такий підхід дозволяє без суттєвих витрат інтегрувати додаткові сервіси у майбутньому, зокрема зовнішні платіжні шлюзи або нові сервіси аналітики й прогнозування.

Особливу увагу приділено підсистемі безпеки, яка базується на використанні JWT-токенів, багаторівневій системі перевірки прав доступу та спеціалізованих охоронцях. Вона ефективно запобігає несанкціонованому доступу до ресурсів застосунку та гарантує високий рівень захисту персональних даних користувачів. Водночас перевірки на рівні підписів запитів від платіжних шлюзів надають додаткову впевненість у достовірності транзакцій, що є критичним для забезпечення фінансової надійності системи.

Суттєвою перевагою запропонованого рішення є реалізація підсистеми кешування з використанням Redis. Це забезпечує значне скорочення часу відповіді сервера і стабільність роботи при підвищеному навантаженні, особливо у періоди пікових звернень. Такий підхід дозволяє значно знизити

навантаження на базу даних PostgreSQL, що сприяє загальному підвищенню продуктивності системи.

Функціональність, пов’язана з керуванням тренувальним процесом, а саме каталог вправ і конструктор планів тренувань, дозволяє ефективно структурувати тренувальні програми та надавати клієнтам персоналізований сервіс. Можливість швидкого адаптування існуючих наборів вправ під конкретні потреби тренерів і клієнтів, а також збереження історії змін дозволяє ретельно відстежувати прогрес і проводити аналіз ефективності тренувальних планів.

Реалізація платіжного функціоналу, зокрема інтеграція з Monobank API, забезпечує високу надійність транзакцій, використовуючи захищений механізм підпису та ідемпотентності. Такий підхід не тільки підвищує довіру клієнтів, але й дозволяє легко додавати нові платіжні сервіси в майбутньому, не змінюючи основної бізнес-логіки.

Описана в розділі система аналітики дозволяє ефективно агрегувати й аналізувати дані про діяльність клубу, створюючи чіткі статистичні звіти щодо відвідуваності, завантаження персоналу й фінансових операцій. Завдяки інтегрованому механізму кешування та обчислювальної оптимізації вона здатна швидко обробляти великі обсяги інформації, що є особливо важливим для оперативного управління клубом.

З вище викладеного слідує, що реалізована архітектура серверної частини застосунку повністю відповідає заявленим цілям і завданням, забезпечуючи високу продуктивність, масштабованість, безпеку та зручність використання. Запропоноване рішення не тільки задовольняє поточні потреби фітнес-клубів, але й забезпечує достатній запас для подальшого розвитку й адаптації до змін бізнес-процесів.

РОЗДІЛ 4
ДОСЛІДЖЕННЯ ТА АНАЛІЗ РОЗРОБЛЕНОЇ СИСТЕМИ

4.1 Перевірка роботи основних сценаріїв

4.1.1 Алгоритм аутентифікації

Під час реєстрації користувач робить запит до точки доступу зазначаючи відповідні дані (email, ПІБ та пароль), зображені на рис. 4.1.

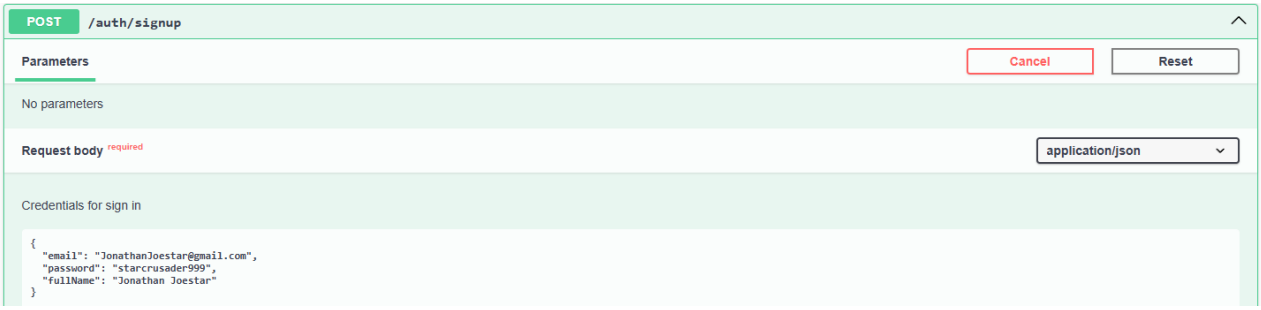


Рисунок 4.1 – Приклад запиту реєстрації

Система перевіряє, чи вже існує обліковий запис з таким email. Якщо ні — створюється новий користувач. Після створення користувача повертається результат операції, що зображений на рис. 4.2.

Рисунок 4.2 – Відповідь сервера при успішній реєстрації

Під час входу через потібно зазначити email і пароль. У вікні форми запиту ці параметри заповнюються тестовими значеннями, зображені на рис. 4.3, які не відповідають раніше зареєстрованому користувачеві у системі.

Рисунок 4.3 – Тестові значення параметрів реєстрації

У разі успішного запиту, зображено на рис. 4.4, генерується JWT-токен. Цей токен надалі використовується для авторизації.

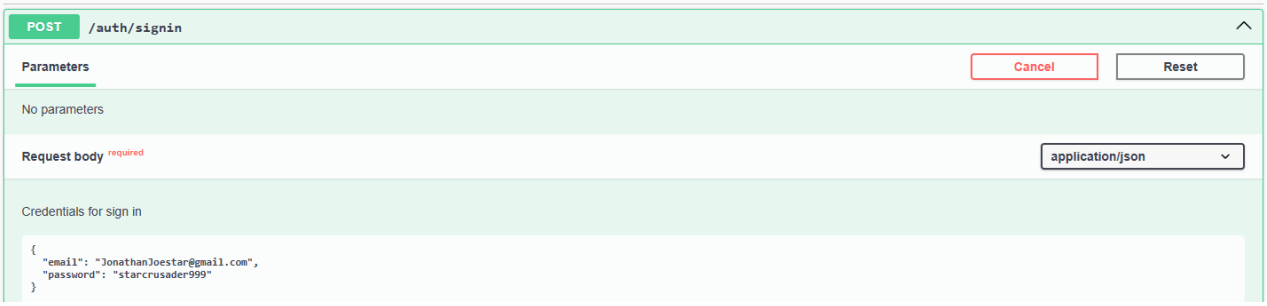


Рисунок 4.4 – Приклад успішного запиту

Після відправлення запиту система має повернутає відповідь, що містить токен доступу, приклад якої зображено на рис. 4.5.

Рисунок 4.5 – Приклад відповіді сервера

Цей токен є ключовим елементом подальшої взаємодії з іншими захищеними маршрутами, приклад запиту до якого зображено на рис. 4.6. У разі успішної відповіді з'являється об'єкт типу JSON, що містить поле «токен доступу». Якщо введено неправильні дані, система повертає повідомлення про помилку з відповідним HTTP-статусом, як-от 401 (Unauthorized), приклад зображено на рис. 4.7.

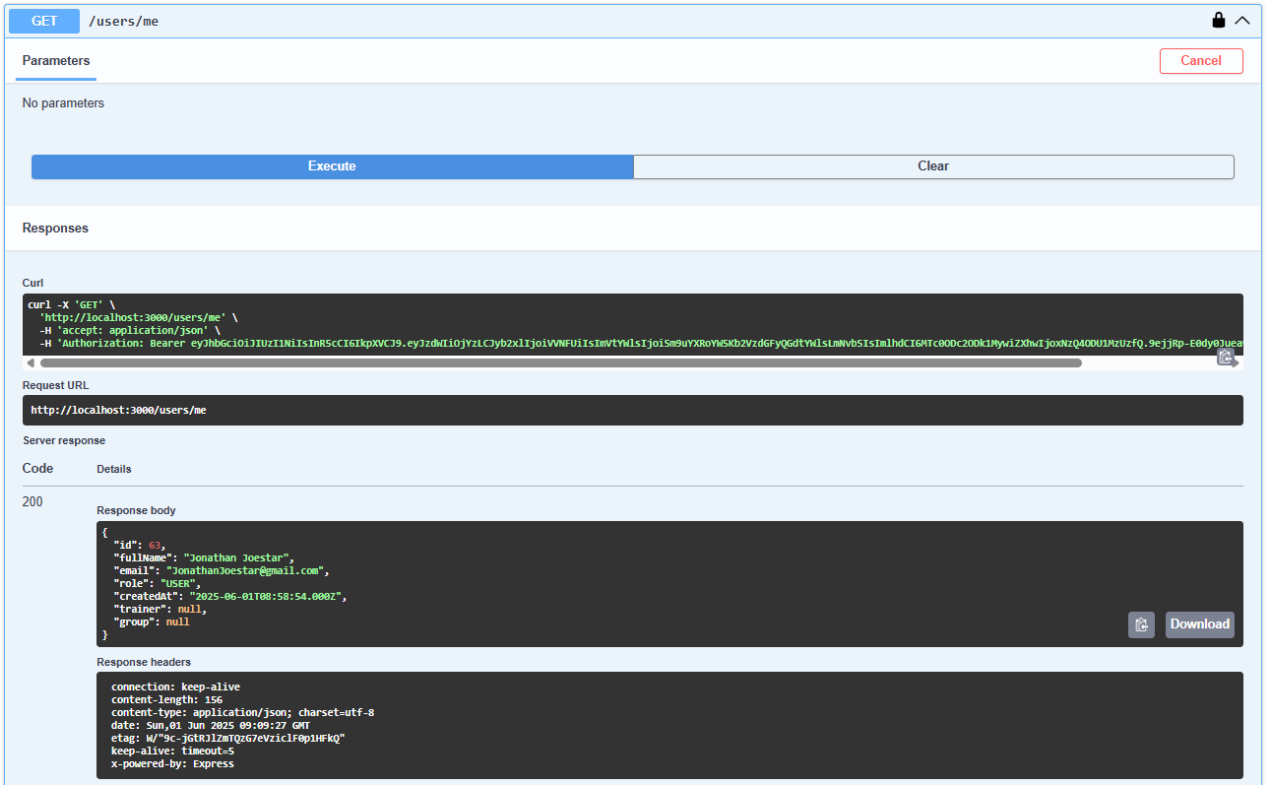


Рисунок 4.6 – Авторизований запит до захищеної точки доступу

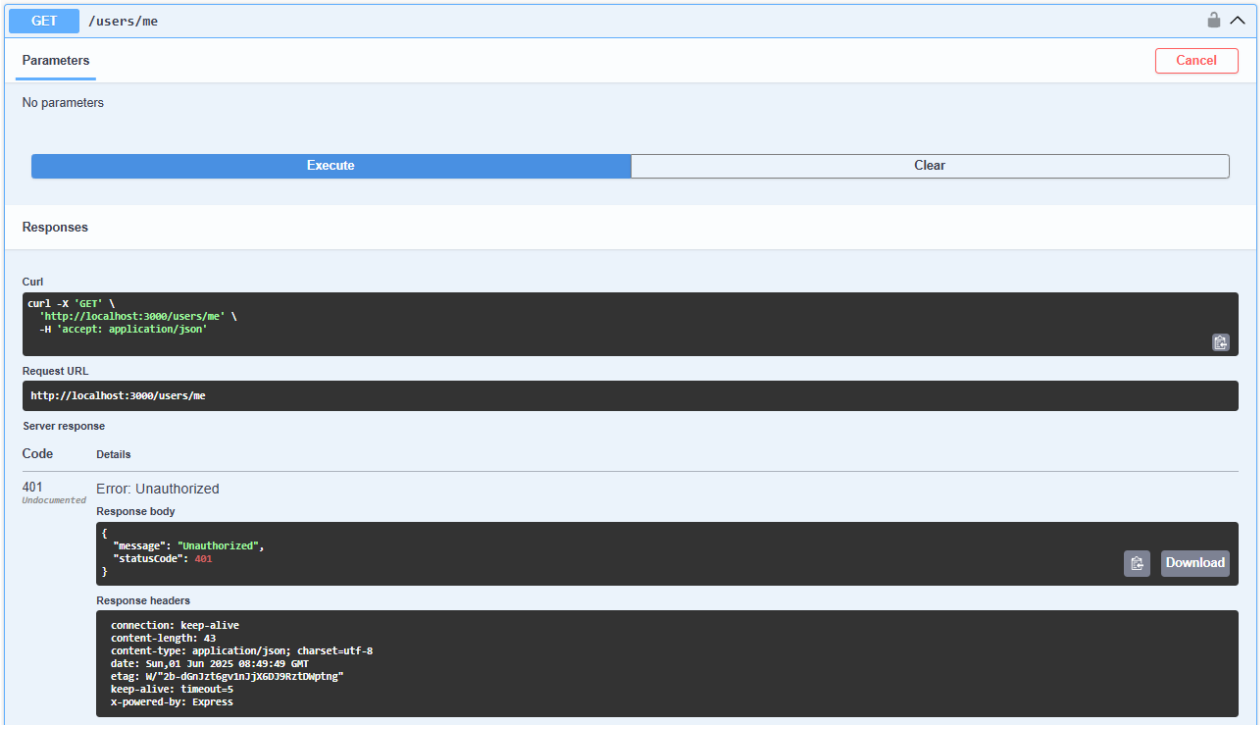


Рисунок 4.7 – Запит без авторизації

Swagger дозволяє повторно використати отриманий токен для авторизації запитів до інших маршрутів. У верхній частині інтерфейсу передбачено можливість вставлення токена в поле авторизації, зображено на рис. 4.8.

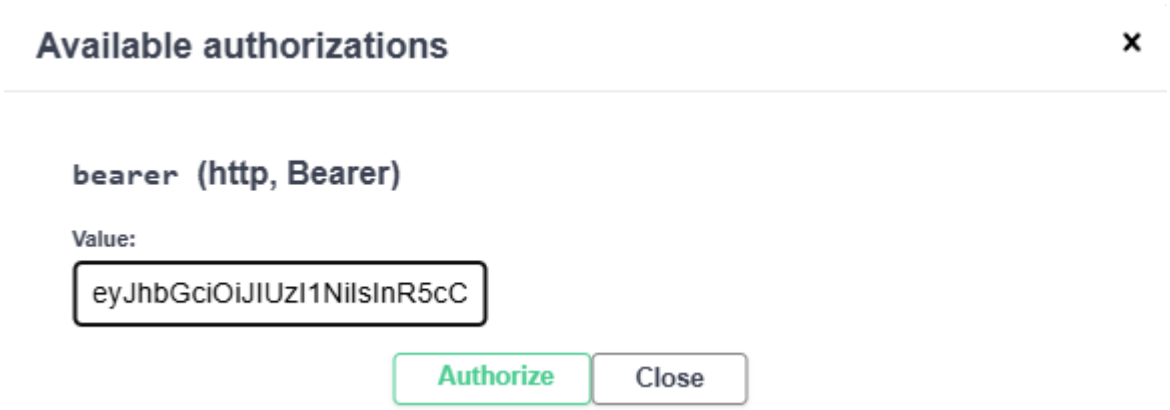


Рисунок 4.8– Вікно авторизації інтерфейсу Swagger

Завдяки цьому можна безпосередньо з інтерфейсу протестувати будь-який приватний маршрут, наприклад, отримання інформації про користувача або запис на тренування. Це не лише пришвидшує перевірку доступності функцій, але й дозволяє наочно відстежити, як система реагує на валідний або прострочений токен.

Поведінка при відсутності токена або використанні його модифікованої версії дозволяє оцінити, наскільки стабільно працює механізм перевірки автентичності та чи обробляються потенційні спроби несанкціонованого доступу. У таких випадках, відповідь містить чітке повідомлення про помилку й не допускає до внутрішніх ресурсів системи, зображено на рис. 4.9.

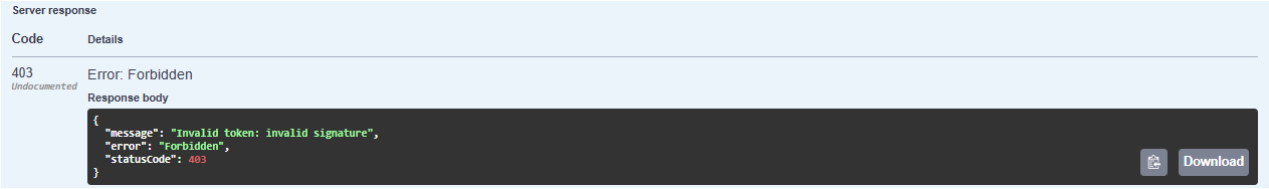


Рисунок 4.9 – Відповідь сервера у разі модифікації токена

4.1.2 Механізм запису на тренування

Механізм запису на тренування є одним із ключових елементів системи — саме він дає змогу користувачам реєструватися на обрані сесії. Процес тестування стартує з авторизації: спочатку проходиться автентифікація, отримується токен доступу, який потім додається у відповідний заголовок. Це забезпечує виконання захищених запитів від імені конкретного користувача.

Після успішної авторизації користувач переходить до точки доступу, яка відповідає за функцію запису. У запиті потрібно заповнити необхідні поля, зображені на рис. 4.10, і після цього надіслати запит. Приклад заповненого вірно запиту, зображено на рис. 4.11.

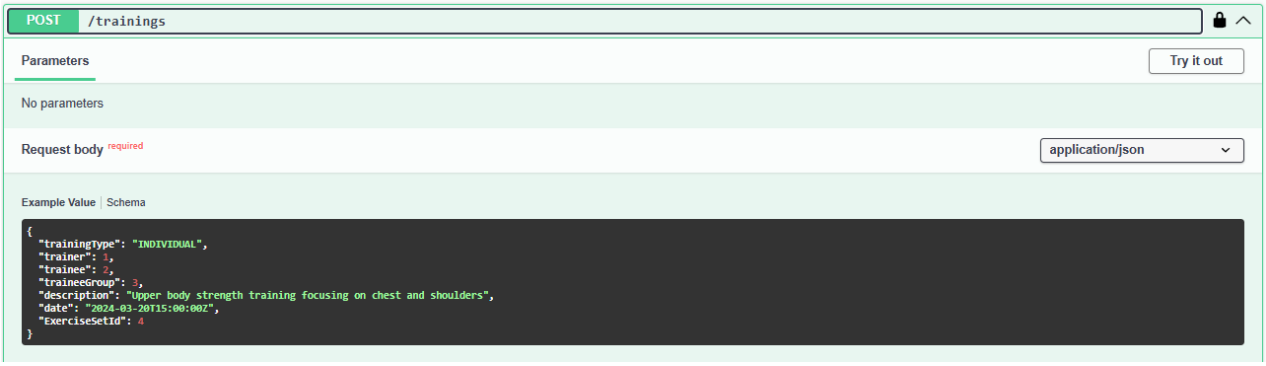


Рисунок 4.10 – Поля запиту на створення тренування

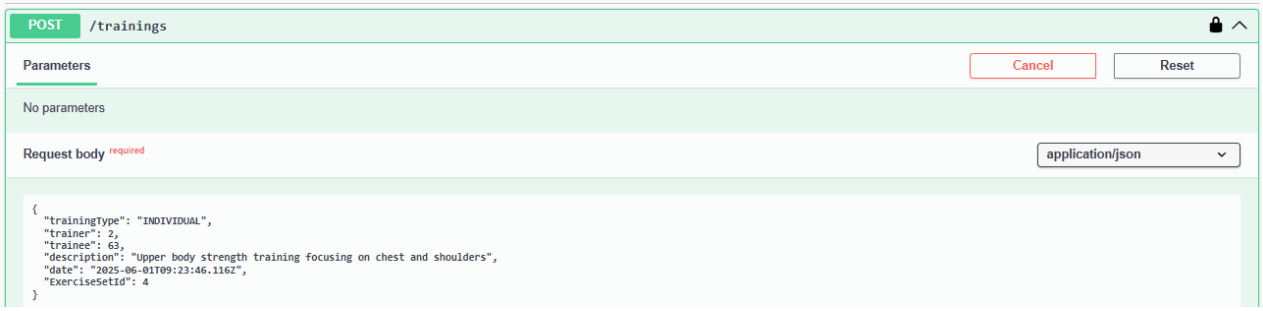


Рисунок 4.11 – Вірно заповнений запит

Варто зазначити, що записати на тренування до будь якого тренера може тільки адміністратор. Тренер же може записати користувача на тренування тільки до себе. Користувач не може створити тренування з тренером, але він може створити тренування без тренера, таким чином показавши свій намір прийти в відповідний час в фітнес-клуб. Також ці записи про тренування користувача використовуються підсистемою статистики для створення статистичних звітів.1

В залежності від даних сервер надішле відповідь. Якщо запит оброблено успішно, користувач отримає статус «успіх» та відповідне повідомлення, зображено на рис. 4.12. У разі помилки система надає зрозуміле повідомлення про причину відмови, зображено на рис. 4.13.

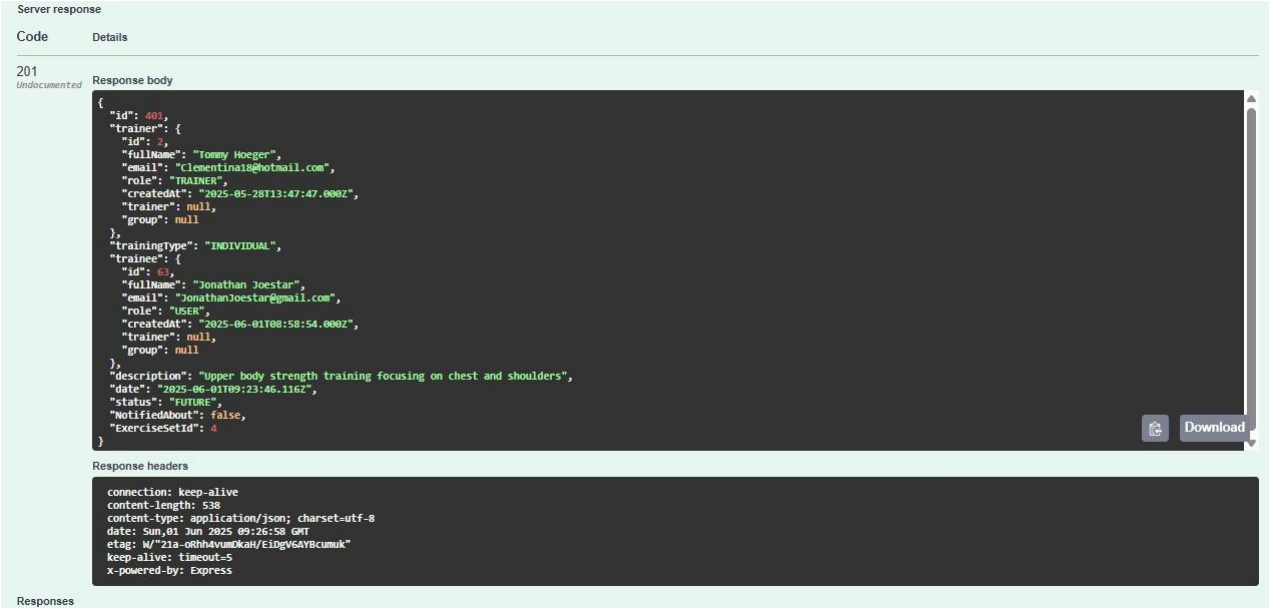


Рисунок 4.12 – Відповідь сервера на вірний запит

Рисунок 4.13 – Відповідь сервера на запит де користувач, що вказаний тренером, ним не є

Також сервер повертає помилку валідації, якщо, один або декілька обов'язкових параметрів відсутній. Відповідь сервера на запит без вказання обов'язкового поля набору вправ зображено на рис. 4.14.

Рисунок 4.14 – Відповідь сервера на запит без обов'язкового поля

4.1.3 Запити на статистичні звіти

Механізм отримання статистичних звітів — важлива частина системи, адже він дозволяє як користувачам, так і адміністраторам аналізувати тренувальний процес, відвідуваність, активність і ключові показники.

Для отримання звіту теж потрібно пройити автентифікацію. Це відкриває можливість надсилати запити до захищених точок доступу, які відповідають за формування статистики. Система розпізнає рівень доступу користувача і дозволяє йому робити лише ті звіти, на які він має право. Користувач обирає потрібний йому статистичний звіт і робить запит. На рис. 4.15 зображено поля потрібні для запиту користувача щодо його статистики відвідувань за певний період.

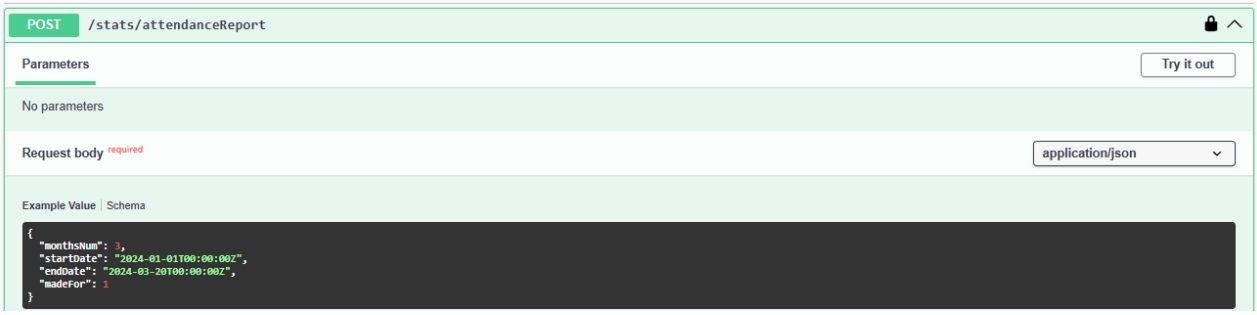


Рисунок 4.15 – Форма для запиту на звіт з відвідуваності

Якщо запит успішний, користувач отримує структуровану інформацію зі статистичними даними, відповідь сервера зображено на рис. 4.16.

Рисунок 4.16 – Відповідь сервера на запит з статистики відвідувань

Якщо ж параметри неправильні або відсутній доступ, система має повернутає інформативне повідомлення про помилку. Звичайний користувач може отримати звіт тільки про своє відвідування а не про відвідування іншого. Відповідь на такий запит зображено на рис. 4.17.

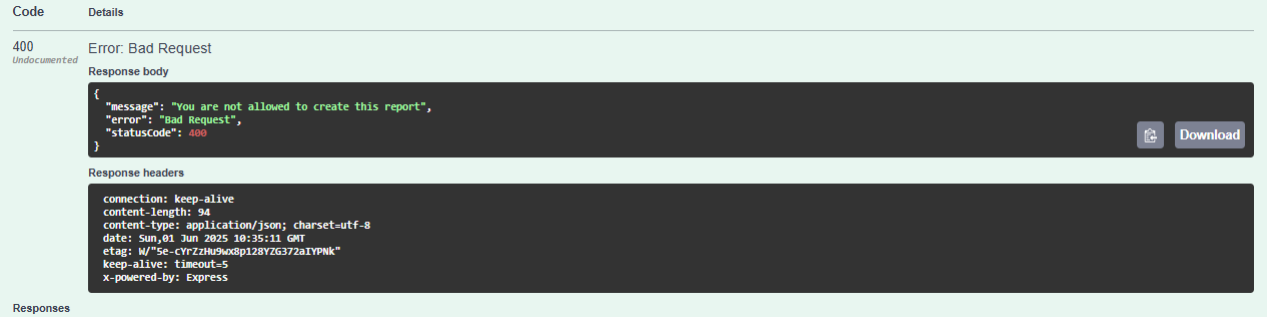


Рисунок 4.17 – Відповідь сервера на неправильний запит на отримання статистики відвідувань

Також для прикладу можна перевірити сценарій, коли не вказано період за котрий потрібно сформувати звіт, такий запит зображено на рис. 4.18. Це допомагає переконатися, що система коректно реагує на різні варіанти запитів та повертає або очікувані результати, або чітке повідомлення про відсутність інформації.

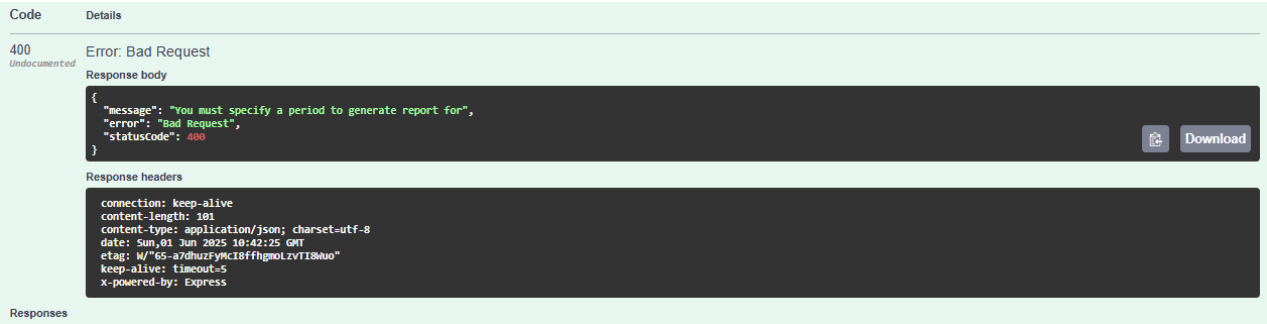


Рисунок 4.18 – Відповідь сервера на неправильний запит без вказаної дати

4.1.4 Механізм оплати

Механізм оплати є критично важливою частиною системи — він забезпечує можливість здійснення фінансових операцій для оплати тренувань, абонементів чи інших послуг.

Перший крок — автентифікація. Це відкриває доступ до захищених ендпоінтів, пов’язаних із оплатою. Далі користувач переходить до ендпоінту для покупки клубного членства, де потрібно вказати на який період він бажає його придбати. Після заповнення запит, надсилається на сервер. Сервер у відповіді повертає платіжне посилання на інтерфейс монобанк, зображений на рис. 4.19. Перейшовщи за посиланням користувач потрапляє на сторінку Монобанк з оплатою, її зображено на рис. 4.20.

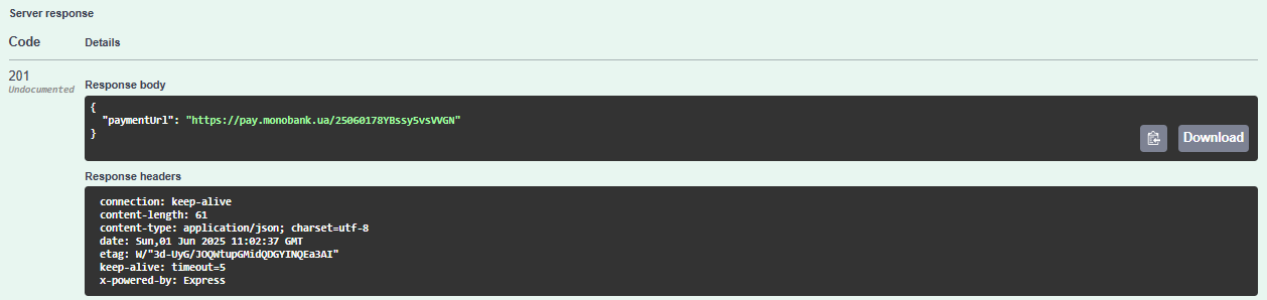


Рисунок 4.19 – Відповідь сервера на запит щодо оплати клубного членства

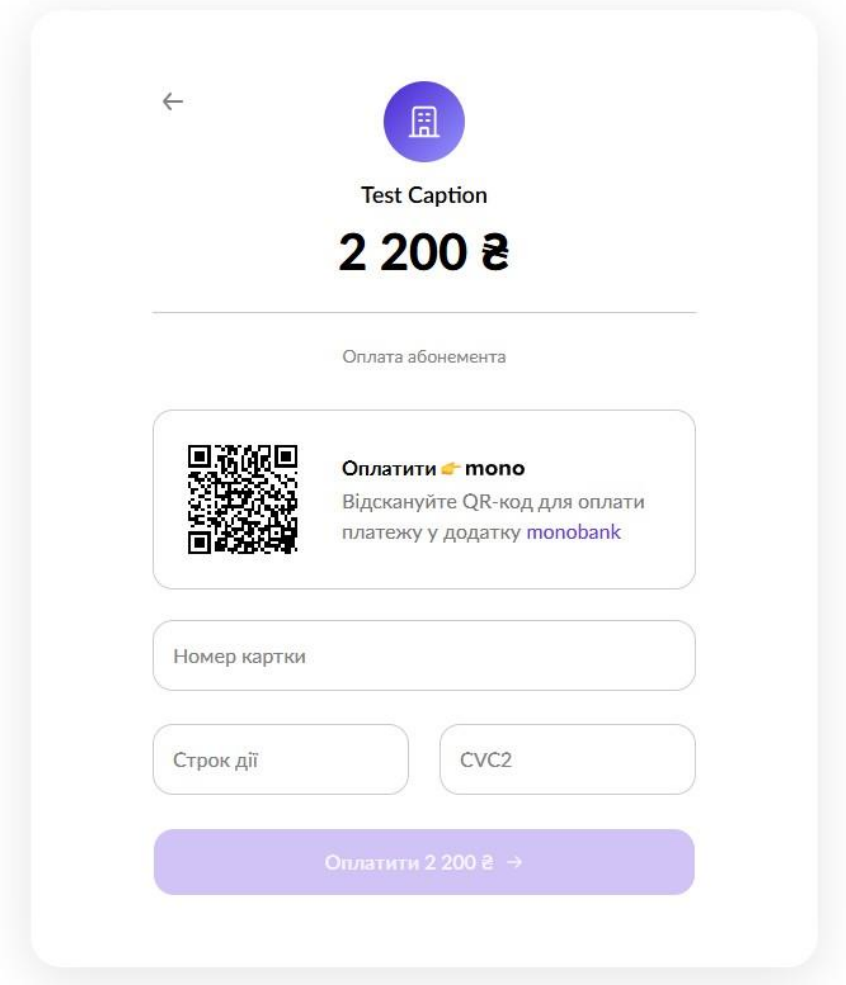


Рисунок 4.20 – Сторінка оплати монобанк

Наступний крок — аналіз відповіді сервера. У разі успішної ініціації оплати користувач отримує підтвердження та деталі транзакції, зображено на рис. 4.21. Якщо ж мають місце помилки — наприклад, через некоректні дані, недостатній баланс або технічні збої — монобанк поверне інформативне повідомлення, що дозволить зрозуміти причину відмови.

Рисунок 4.21 – Сторінка успішної оплати монобанк

Після цього монобанк відправляє запит до відповідної точки доступу додатку з даними, щодо проведення платежу. Якщо все пройшло успішно, сервер оновить статус абонемента користувача на активний. На рис. 4.22 зображено дані запиту на отримання інформації щодо клубного членства.

Рисунок 4.22 – Інформація щодо оплаченого членства

Якщо подивитися на поля відповіді сервера, то можна побачити, що є поле щодо оновлення. Тобто після успішної оплати клубне членство стало активним.

4.2 Аудит безпеки

4.2.1 Перевірка захисту точок доступу

Перевірка захисту точок доступу є одним з ключових етапів тестування безпеки, адже саме через них відбувається взаємодія користувачів із сервером. Мета — переконатися, що всі критичні точки доступу надійно захищені, а неавторизовані чи неавтентифіковані користувачі не можуть отримати доступ до захищених ресурсів або виконувати заборонені дії.

Першим етапом є визначення списку ендпоінтів, що підлягають захисту — запити до персональних даних, запис на тренування, оплату, керування групами чи статистику. Для кожного з них спочатку здійснюється запит без авторизації, тобто без токена. Перевіряється, чи система повертає очікувану заборону доступу, та чи містить відповідь пояснення щодо необхідності авторизації, дану відповідь зображено на рис. 4.23.

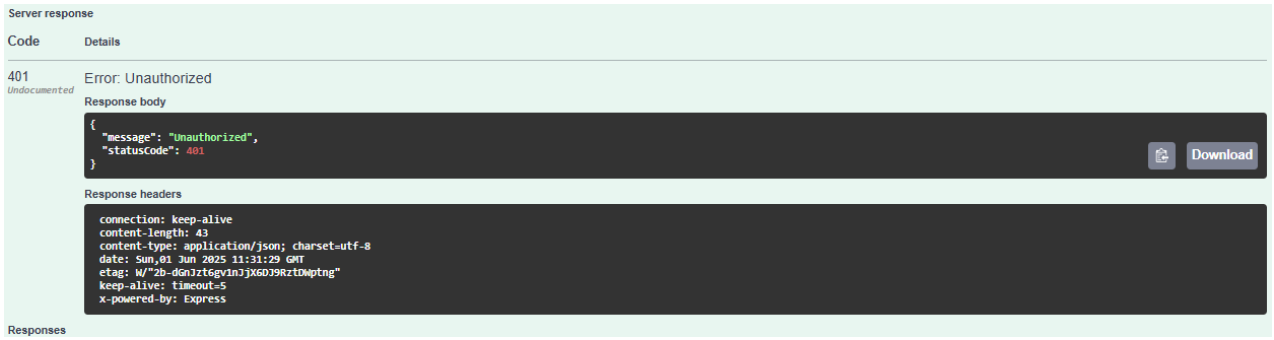


Рисунок 4.23 – Відповідь сервера щодо необхідності авторизуватись

Далі перевіряється реакція системи на запити користувача з обмеженими правами. Наприклад, якщо звичайний користувач намагається виконати дії, доступні лише адміністраторам або тренерам, система повинна відмовити в доступі з відповідним повідомленням, зображено на рис. 4.24.

Рисунок 4.24 – Відповідь сервера щодо заборони доступу

Окремо тестується захист від несанкціонованого редагування або видалення даних. Важливо перевірити, що користувач без належних прав не може змінити або видалити інформацію, яка йому не належить. Такі запити повинні блокуватися з відповідними повідомленнями про помилку. На рис. 4.25 зображено спробу користувача, що не є адміністратором змінити дані щодо іншого користувача. На рис. 4.26 показано відповідь сервера.

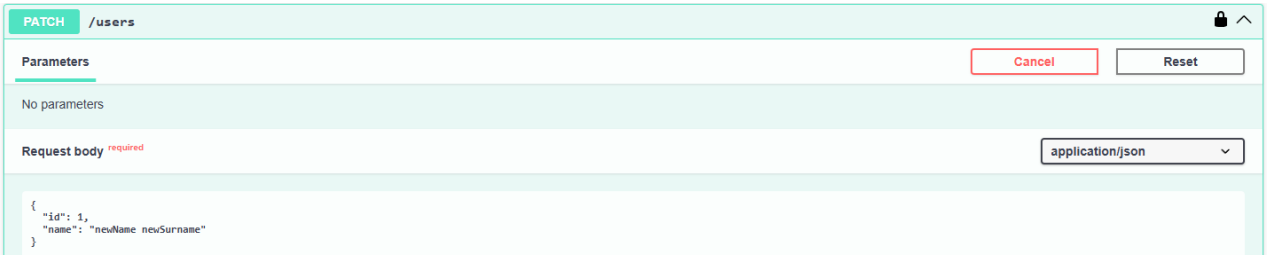


Рисунок 4.25 – Несанкціонований запит на зміну даних щодо користувача

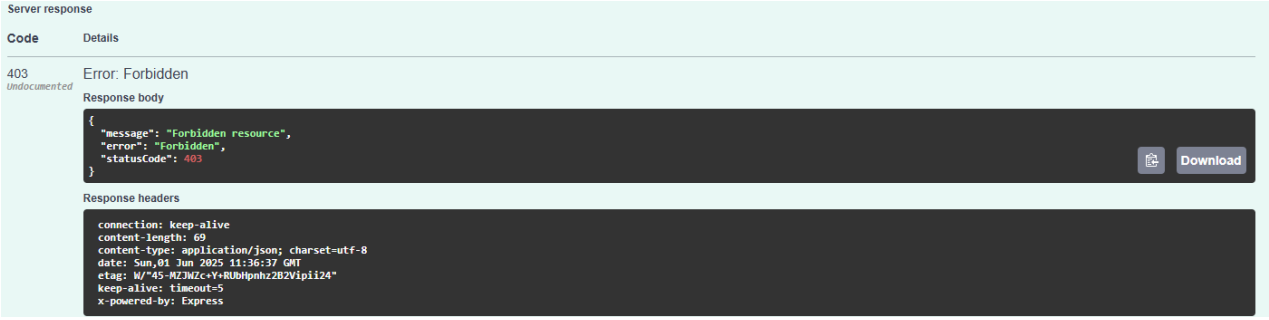


Рисунок 4.26 – Відповідь сервера на відповідний запит

Також необхідно перевірити реакцію системи на прострочені або недійсні токени, такий запит зображено на рис. 4.27. У таких випадках очікується повернення повідомлення про недійсність або завершення терміну дії токена — це вказує на наявність належної перевірки автентичності.

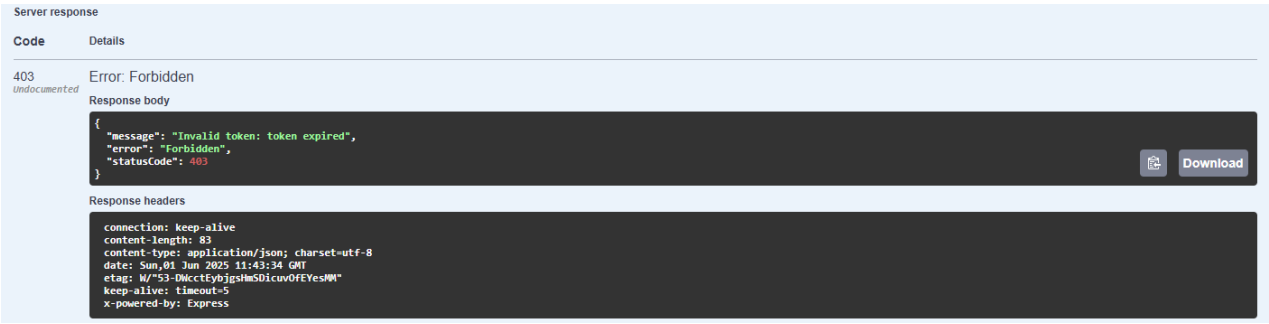


Рисунок 4.27 – Відповідь сервера при використанні токена у якого завершився термін дії

Загалом, дане тестування дозволяє впевнитися, що всі критичні точки доступу системи надійно захищені, а механізми автентифікації та контролю доступу працюють відповідно до вимог безпеки.

4.2.2 Валідація платежів

Тестування валідації платежів є критичною, оскільки саме коректність обробки фінансових транзакцій визначає надійність системи та довіру користувачів.

Спершу необхідно створити запит на оплату абонемента через відповідну точку доступу. Безпека процесу тестування забезпечується через використання тестових ключів та токенів Монобанку, які мають обмежені права та не дозволяють здійснювати реальні фінансові операції. Під час тестування перевіряється, чи коректно система генерує унікальний ідентифікатор транзакції та чи правильно формується запит до інтерфейсу Монобанку, логування відправлених даних зображено на рис. 4.28. Така архітектура тестування відповідає сучасним вимогам до безпеки розробки фінансових систем та дозволяє уникнути потенційних ризиків.

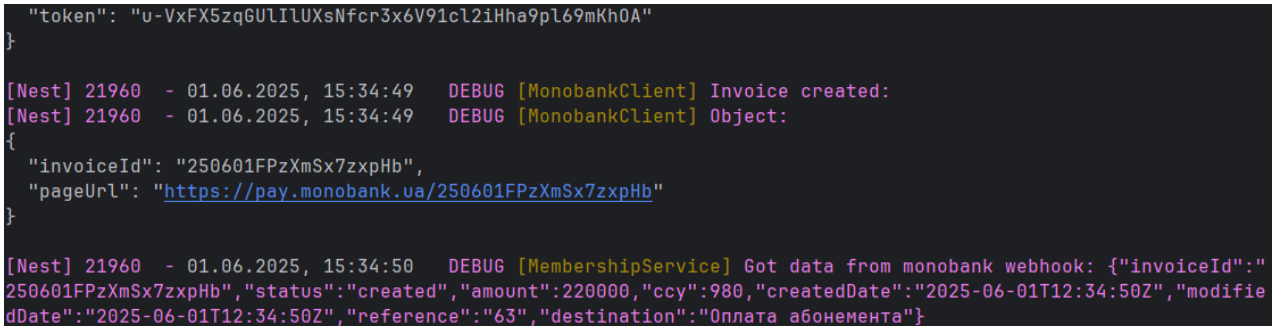


Рисунок 4.28 – Інформація запиту до інтерфейса

Наступним етапом є отримання вебхуку від Монобанку, який сповіщає про статус платежу, зображено на рис. 4.29. Для цього використовується окрема точка доступу, захищений спеціальним охоронцем MonobankWebhookGuard.

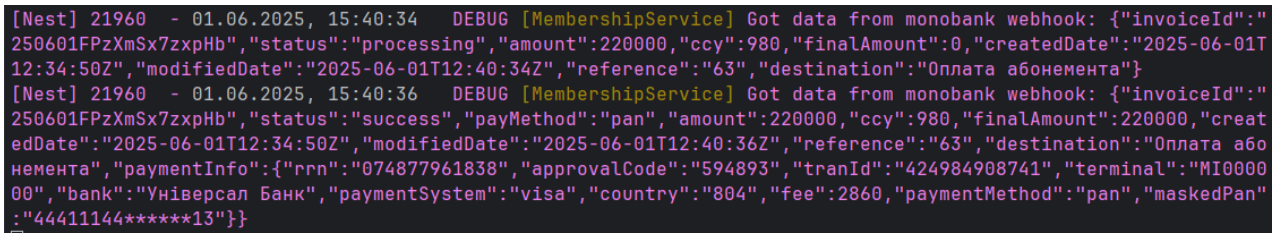


Рисунок 4.29 – Отримання запиту від інтерфейса монобанк, щодо статусу платежу

Після оплати отриманий запит і система має перевірити підпис, знайти відповідний запис про абонемент та оновити його статус, частина оновлених даних згідно останнього запиту зображено на рис. 4.30

Рисунок 4.30 – Частина оновленого запису про клубне членство

Від отримання недостовірних запитів про успішну оплату захищає MonobankWebhookGuard. Він забезпечує достовірність запитів, що отримуються від платіжної системи Монобанк. Він перевіряє криптографічний цифровий підпис. Спробуємо підробити запит про успішну оплату. Для цього проаналізуємо достовірний запит від Монобанк, що зображено на рис. 4.31.

```
{
  "invoiceId":"250601FPzXmSx7zxpHb",
 "status":"success",
 "payMethod":"pan",
 "amount":220000,
 "ccy":980,
 "finalAmount":220000,
 "createdDate":"2025-06-01T12:34:50Z",
 "modifiedDate":"2025-06-01T12:40:36Z",
 "reference":"63",
 "destination":"Оплата абонемента",
 "paymentInfo":
 {
   "rrn":"074877961838",
   "approvalCode":"594893",
   "tranId":"424984908741",
   "terminal":"MI000000",
   "bank":"Універсал Банк",
   "paymentSystem":"visa",
   "country":"804",
   "fee":2860,
   "paymentMethod":"pan",
   "maskedPan":"44411144******13"
 }
}
```


Рисунок 4.31 – Запит від інтерфейсу монобанк

Створимо новий абонемент, що чекатиме на оплату, інформацію про нього зображено на рис. 4.32.


```
{
  "user_id":63,
 "payment_url":"https://pay.monobank.ua/250601E5uxuFoMzUQJKz",
 "paid_at":null,
 "created_at":"2025-06-01T13:15:41+00:00",
  "updated_at":"2025-06-01T13:15:41+00:00",
  "status":"pending"
}
```


Рисунок 4.32 – Новий абонемент, що чекає на оплату

Після цього підготуємо «шахрайський» запит за допомоги curl, його зображено на рис. 4.33.

```
curl -X POST http://localhost:3000/membership/callback0-mono \
  -H "Content-Type: application/json" \
  -d '{
    "invoiceId": "250601E5uxuFoMzUQJKz",
    "status": "success",
    "payMethod": "pan",
    "amount": 220000,
    "ccy": 980,
    "finalAmount": 220000,
    "createdDate": "2025-06-01T14:25:10Z",
    "modifiedDate": "2025-06-01T14:29:44Z",
    "reference": "77",
    "destination": "Оплата підписки",
    "paymentInfo": {
      "rrn": "074877962145",
      "approvalCode": "482516",
      "tranId": "424984909111",
      "terminal": "MI000000",
      "bank": "Універсал Банк",
      "paymentSystem": "visa",
      "country": "804",
      "fee": 2650,
      "paymentMethod": "pan",
      "maskedPan": "44411144******92"
    }
  }'
```

Рисунок 4.33 – Шахрайський запит

Тепер спробуємо його виконати шахрайський запит. В результаті виконання отримаємо відповідь, що підпис є недостовірним, зображено на рис. 4.34. Це підтверджує працездатність захисту.

```
PS C:\Users\denis> Invoke-WebRequest -Uri "http://localhost:3000/membership/callback-mono" -Method POST -Headers @{ "Content-Type" = "application/json"; "x-sign" = "MEUCIQD8Oe+H7hxI+lOZ3fQ8oIsOoy3gQkn1OmM0iM1FP5p2egIgOQ4YvHsXz3Y5m2so3Q+fDBvXe2qv5nPa7lRJk+rKh0Q=" } -Body '{"invoiceId":"250601E5uxuFoMzUQJKz","status":"success","payMethod":"pan","amount":220000,"ccy":980,"finalAmount":220000,"createdDate":"2025-06-01T14:25:10Z","modifiedDate":"2025-06-01T14:29:44Z","reference":"77","destination":"Оплата підписки","paymentInfo":{"rrn":"074877962145","approvalCode":"482516","tranId":"424984909111","terminal":"MI000000","bank":"Універсал Банк","paymentSystem":"visa","country":"804","fee":2650,"paymentMethod":"pan","maskedPan":"44411144******92"}}'
Invoke-WebRequest : {"message":"Invalid signature","error":"Unauthorized","statusCode":401}
At line:1 char:1
+ Invoke-WebRequest -Uri "http://localhost:3000/membership/callback-mon ...
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidOperation: (System.Net.HttpWebRequest:HttpWebRequest) [Invoke-WebRequest], WebException
    + FullyQualifiedErrorId : WebCmdletWebResponseException,Microsoft.PowerShell.Commands.InvokeWebRequestCommand
PS C:\Users\denis>
```

Рисунок 4.34 – Результат виконання шахрайського запиту

4.3 Оцінка реалізованої архітектури

Оцінюючи реалізовану архітектуру, можна виокремити низку вагомих переваг, які забезпечують її ефективність і зручність подальшого розвитку. Першою з яких є чітка модульна структура, оскільки поділ функціональності на окремі компоненти полегшує підтримку та розширення системи. Розділення на підсистеми забезпечує легкість внесення змін у конкретні частини без ризику вплинути на інші.

Важливо відзначити те, що об’єднання підсистем через єдиний механізм авторизації створює умовний центральний вузол, що дозволяє забезпечити безпеку та керованість доступу до ресурсів. Реалізація перевірок авторизації та автентифікації забезпечує безпеку чутливих даних.

Інтеграція з платіжною системою Monobank відбувається через асинхронні вебхуки, що дозволяє зменшити затримки під час обробки фінансових операцій. Це дозволяє швидко реагувати на зміну статусу платежу та стрияє підвищенні стійкості до навантажень, адже система не витрачає ресурси на очікування відповіді платіжного шлюзу. Такий підхід мінімізує ймовірність виникнення некоректних транзакцій.

Не менш вагомою перевагою є реалізація кешування на транзакційному рівні, що значною мірою підвищує продуктивність. Використання Redis дає змогу позбутися зайвих звертань до PostgreSQL у випадку частої запитуваної інформації, наприклад даних про розклад тренувань чи статистику клієнта. Завдяки цьому забезпечується плавність роботи системи у періоди пікових навантажень, коли одночасно звертається велика кількість користувачів

Однією з помітних позитивних рис є гнучкість використання базового сервісу для виконання операцій, що мінімізує дублювання коду і сприяє

єдиному стилю взаємодії з базою даних. Таке рішення дозволяє легко адаптувати існуючий код до змінених бізнес-вимог.

Використання деплойменту з Docker-контейнерами і можливість розгортання кожного модуля незалежно один від одного сприяють горизонтальному масштабуванню. За необхідності можна створити кластер контейнерів для модулів тренувань, статистики або оплати, в залежності від того, де саме фіксується найбільше навантаження. Під час тестування таблиці було заповнено тестовими даними. У такому стані контейнер використовував близько 60 мб оперативної пам’яті.

Отже, оцінюючи реалізовану архітектуру, можна стверджувати, що вона демонструє баланс між простотою розширюваності, продуктивністю і безпекою. З вище викладеного слідує, що обрана структура здатна підтримувати високу ефективність роботи сервісу навіть за значної кількості одночасних користувацьких запитів.

ВИСНОВОК ДО РОЗДІЛУ 4

У даному розділі було виконано докладний аналіз і перевірку роботи реалізованої серверної частини веб-застосунку для організації роботи фітнес-клубів, що дозволило об’єктивно оцінити якість розробленої системи за декількома критичними напрямками: коректність основних сценаріїв, безпека та ефективність архітектури.

Перевірка основних сценаріїв використання, таких як механізми аутентифікації, запису на тренування, формування статистичних звітів і здійснення платежів, підтвердила правильність їх роботи та повну відповідність заявленим вимогам. Процес реєстрації користувачів і отримання JWT-токенів було ретельно протестовано, що дозволило переконатися у коректності автентифікації та авторизації. Завдяки інтегрованому Swagger UI, процес тестування спрощується і забезпечує зрозумілість взаємодії з API для різних категорій користувачів.

Тестування механізму запису на тренування засвідчило, що система належним чином реагує на помилки у запитах, а також на спроби несанкціонованого доступу. Водночас функціонал належно перевіряє права доступу користувачів, дозволяючи їм взаємодіяти виключно з тими ресурсами, які відповідають їхнім ролям, що додатково підтверджує ефективність механізмів безпеки.

Валідація платіжних операцій та інтеграції з Monobank API продемонструвала високий рівень надійності обробки фінансових транзакцій. Завдяки суворій перевірці цифрового підпису вебхуків, було підтверджено, що система ефективно захищена від можливих атак на фінансові процеси. Використання механізму ідемпотентності дозволило уникнути помилок, пов’язаних з дублюванням або повторною обробкою платежів, а ретельний

аналіз шахрайських запитів підтвердив працездатність захисту від несанкціонованих дій.

Аудит безпеки точок доступу став одним із найважливіших етапів дослідження, який засвідчив ефективність реалізованих рішень для захисту даних і бізнес-логіки. Тестування показало, що серверна частина чітко розмежовує права користувачів і адекватно реагує на будь-які неавторизовані спроби доступу. Особливо важливим стало тестування поведінки системи на прострочені чи недійсні токени, що підтвердило надійність механізмів автентифікації та авторизації.

Оцінка реалізованої архітектури підкреслила низку важливих переваг системи, таких як модульна структура, ефективна інтеграція платіжних механізмів, кешування за допомогою Redis, а також продуманий підхід до авторизації та аналітики. Зокрема, архітектура дозволяє ефективно масштабуватися за умов зростання кількості користувачів, що забезпечує необхідну гнучкість для подальшого розвитку застосунку.

З вище викладеного слідує, що проведені дослідження та аналіз реалізованої системи підтвердили її високу якість, стабільність роботи та відповідність початково сформульованим цілям і технічним вимогам. Отримані результати свідчать, що розроблена серверна частина здатна ефективно працювати у реальних умовах експлуатації, забезпечуючи необхідний рівень безпеки, продуктивності та гнучкості, що робить її перспективною для подальшого впровадження у фітнес-клубах.

ВИСНОВКИ

У рамках даної дипломної роботи було виконано проектування та реалізацію серверної частини веб-застосунку для організації роботи фітнес-клубів. Головною метою дослідження було створення універсального рішення, яке дозволило б стандартизувати та спростити процеси створення програмних продуктів для автоматизації діяльності фітнес-індустрії. Поставлена мета була досягнута завдяки комплексному підходу до аналізу вимог, вибору оптимальних технологій, проектуванню архітектури та її практичній реалізації з подальшим ретельним тестуванням.

На етапі аналізу існуючих рішень було детально досліджено бізнес-логіку й функціональні можливості таких популярних продуктів, як Mindbody і ABC Trainerize. Це дало змогу виокремити ключові функції, які обов'язково мають бути реалізовані в уніфікованій платформі, зокрема керування тренуваннями, аналітика, модуль оплати та забезпечення безпеки. Важливо, що саме ці функції становлять ядро розробленого програмного продукту.

У другому розділі роботи було здійснено аналіз і вибір технологічного стеку для проекту, який включає мову TypeScript, фреймворк NestJS, систему управління базами даних PostgreSQL і ORM-інструмент MikroORM. Обрані технології виявилися ефективними для вирішення поставлених завдань, оскільки забезпечують високий рівень безпеки, продуктивності, простоти розширення та підтримки коду. Важливим додатковим аспектом було використання Redis для кешування, WebSocket для миттєвої взаємодії з користувачем і Swagger для інтерактивної документації API, що значно спростило процес розробки й тестування.

Третій розділ роботи містить детальний опис практичної реалізації основних підсистем сервера. В ньому показано, як чітко розмежовані модулі взаємодіють

між собою, забезпечуючи комплексну функціональність. Особливу увагу приділено механізму безпеки, який поєднує багаторівневу перевірку прав доступу й автентифікацію на основі JWT, що гарантує захист від несанкціонованого доступу. Також було детально описано роботу підсистеми керування тренувальним процесом, що містить каталог вправ і конструктор планів тренувань, модуль оплати з інтеграцією Monobank API, підсистему аналітики та ефективний механізм кешування, які у сукупності забезпечують стабільність роботи під час високих навантажень.

У четвертому розділі було здійснено детальну перевірку роботи ключових сценаріїв системи, таких як реєстрація, авторизація, запис на тренування, здійснення платежів та формування статистичних звітів. Це дозволило підтвердити повну відповідність системи заявленим вимогам та засвідчити її готовність до використання у реальних умовах. Особливу увагу було приділено безпеці, валідації платежів та контролю доступу, які продемонстрували високу надійність і стійкість до можливих атак або шахрайських дій.

Отже, реалізована серверна частина застосунку повністю відповідає сучасним технічним та бізнес-вимогам фітнес-індустрії. Обрана архітектура є збалансованим поєднанням продуктивності, безпеки та гнучкості, завдяки чому система легко адаптується до змін і масштабування. Реалізоване рішення дозволяє значно скоротити витрати часу й ресурсів на розробку нових програмних продуктів, забезпечуючи фітнес-клубам зручний і безпечний інструмент для ефективного управління своєю діяльністю.

З вище викладеного слідує, виконана робота досягла поставленої мети й вирішила сформульовані завдання, пропонуючи ринку сучасне, надійне та масштабоване серверне рішення для автоматизації бізнес-процесів фітнес-клубів, що відкриває широкі можливості для подальшого впровадження і розвитку.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Mindbody – головна сторінка [Електронний ресурс] – Режим доступу до ресурсу: https://www.mindbodyonline.com/
2. Mindbody app [Електронний ресурс] – Режим доступу до ресурсу: https://www.mindbodyonline.com/business/mindbody-app
3. Mindbody - scheduling [Електронний ресурс] – Режим доступу до ресурсу: https://www.mindbodyonline.com/business/scheduling
4. Trainerize – головна сторінка [Електронний ресурс] – Режим доступу до ресурсу: https://www.trainerize.com/features/stripe-payments/
5. Довідник користувача Trainerize [Електронний ресурс] – Режим доступу до ресурсу: https://help.trainerize.com/hc/en-us/articles/211163423-Other-Ways-of-Collecting-Payments
6. Довідник Trainerize для бізнесу [Електронний ресурс] – Режим доступу до ресурсу: https://www.trainerize.com/pricing
7. Форум для користувачів Trainerize [Електронний ресурс] – Режим доступу до ресурсу: https://ideas.trainerize.com/forums/167887-feature-requests-ideas-and-suggestions
8. Документація мови Python, вступ [Електронн1ий ресурс] – Режим доступу до ресурсу: https://docs.python.org/3/tutorial/introduction.html
9. Документація django [Електронний ресурс] – Режим доступу до ресурсу: https://docs.djangoproject.com/en/5.2/
10. Документація FastAPI [Електронний ресурс] – Режим доступу до ресурсу: https://fastapi.tiangolo.com/
11. Документація Node.js [Електронний ресурс] – Режим доступу до ресурсу: https://nodejs.org/docs/latest/api/

12. Документація TypeScript [Електронний ресурс] – Режим доступу до ресурсу: https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html
13. MongoDB, BSON [Електронний ресурс] – Режим доступу до ресурсу: https://www.mongodb.com/docs/manual/reference/bson-types/
14. Гнучкі смехи MongoDB BSON [Електронний ресурс] – Режим доступу до ресурсу: https://www.mongodb.com/docs/manual/core/data-model-design/
15. Sharding MongoDB [Електронний ресурс] – Режим доступу до ресурсу: https://www.mongodb.com/docs/manual/sharding/
16. Транзакції PostgreSQL [Електронний ресурс] – Режим доступу до ресурсу: https://www.postgresql.org/docs/current/tutorial-transactions.html
17. Індекси PostgreSQL [Електронний ресурс] – Режим доступу до ресурсу: https://www.postgresql.org/docs/current/indexes.html
18. Модулі в NestJS [Електронний ресурс] – Режим доступу до ресурсу: https://docs.nestjs.com/modules
19. Провайдери в NestJS [Електронний ресурс] – Режим доступу до ресурсу: https://docs.nestjs.com/providers
20. Контролери в NestJS [Електронний ресурс] – Режим доступу до ресурсу: https://docs.nestjs.com/controllers
21. Маршрутизація в Express.js [Електронний ресурс] – Режим доступу до ресурсу: https://expressjs.com/en/guide/routing.html
22. Посередники в Express.js [Електронний ресурс] – Режим доступу до ресурсу: https://expressjs.com/en/guide/writing-middleware.html
23. Обробка помилок в Express.js [Електронний ресурс] – Режим доступу до ресурсу: https://expressjs.com/en/guide/error-handling.html
24. Інтерфейс Swagger [Електронний ресурс] – Режим доступу до ресурсу: https://swagger.io/tools/swagger-ui

ДОДАТОК 1

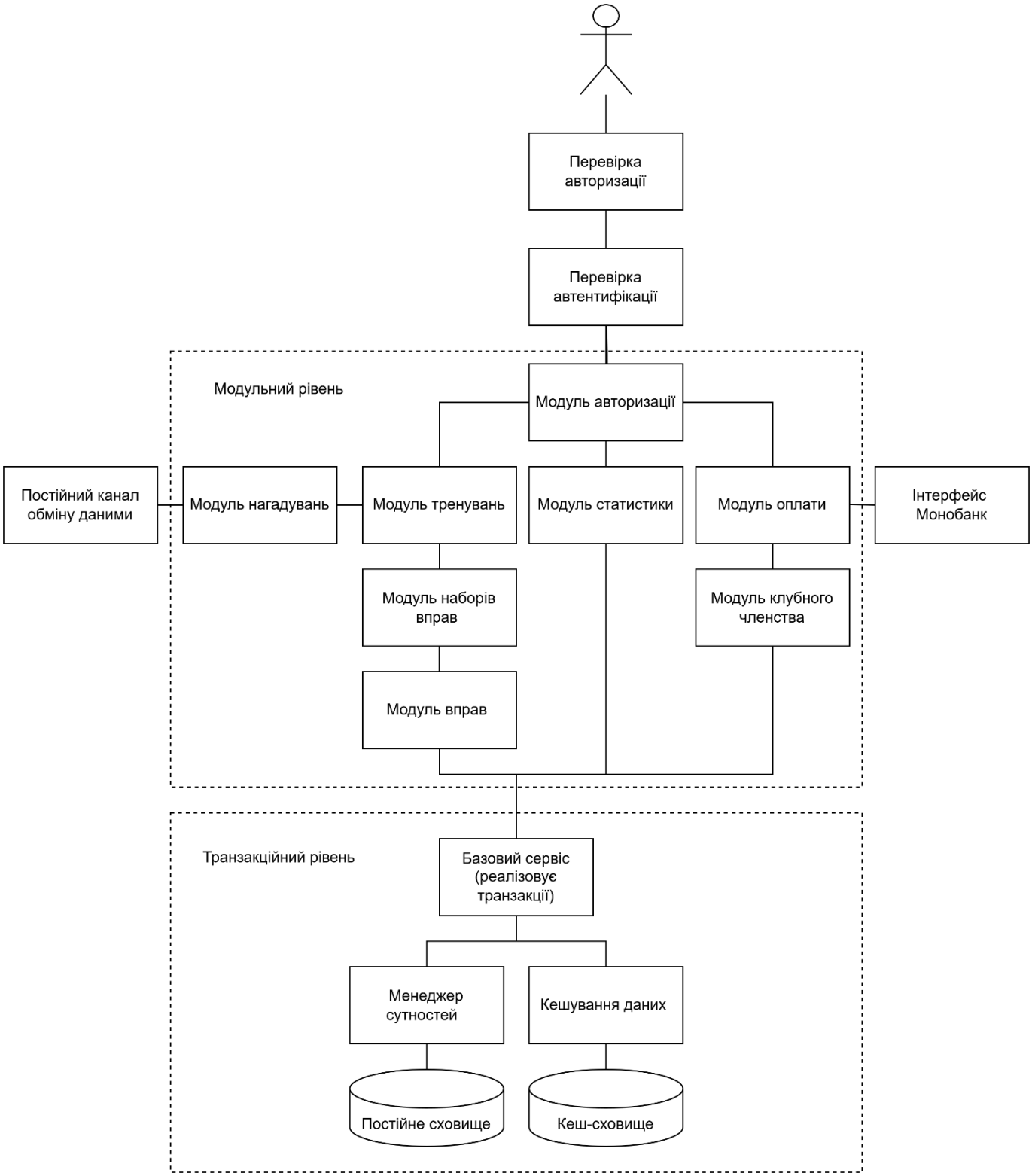
Серверна частина веб-застосунку для організації роботи фітнес-клубів

Структурна схема системи

ІАЛЦ.467200 Д1

Аркушів 1

Київ 2025 р



					ІАЛЦ.467200.004 Д1	
		№ докум.	Підпис	Дата		
Розробив		Шерстюк Д. М.			**Серверна частина веб-застосунку для організації роботи фітнес-клубів** Структурна схема системи	Літ. / Аркуш 1 / Аркушів 1
Перевірив		Волокита А. М.				
Н. Контр.		Пономаренко А. М.				КПІ ім. Ігоря Сікорського, ФІОТ,ІМ-13
Затвердив						

ДОДАТОК 2

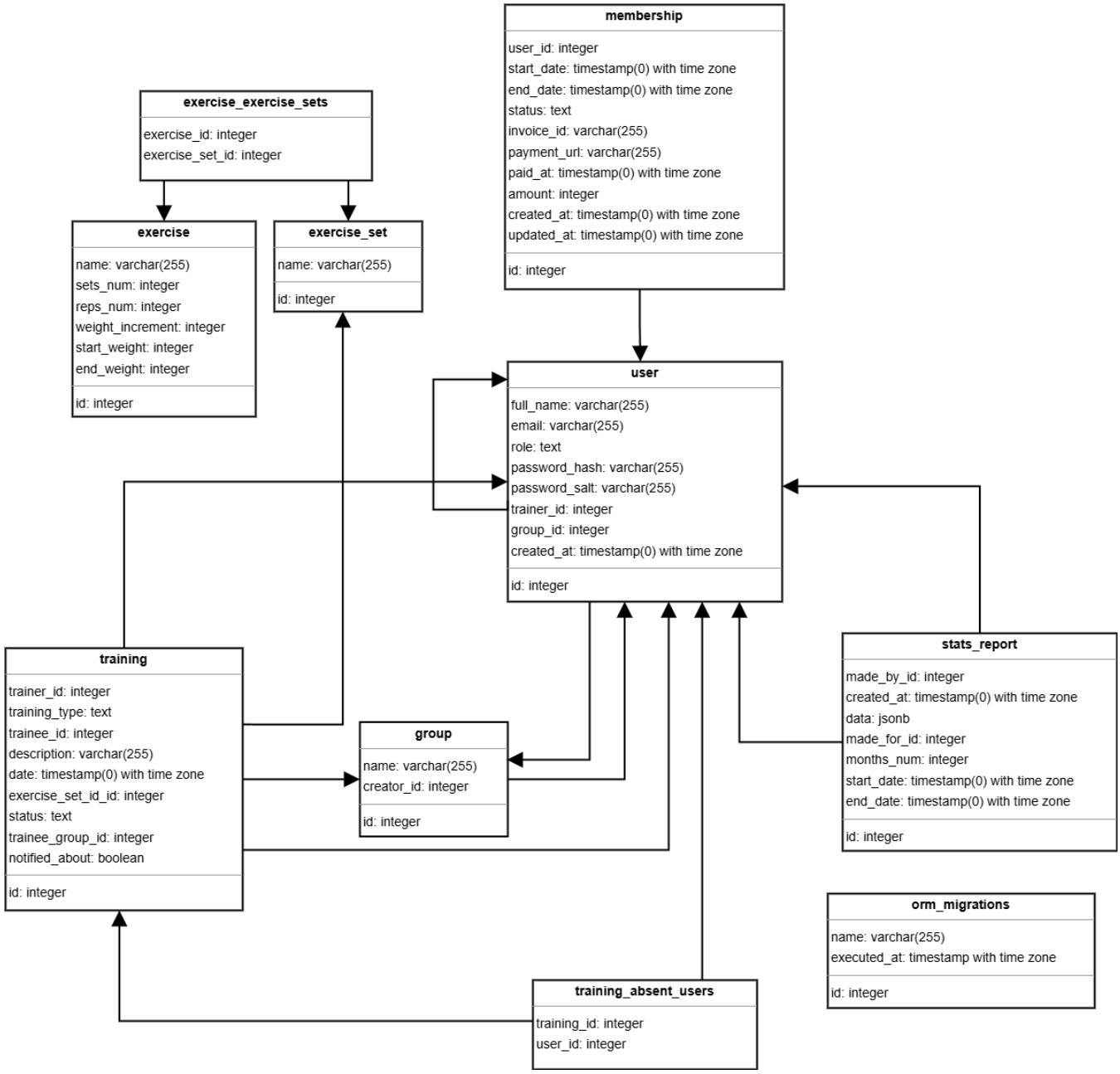
Серверна частина веб-застосунку для організації
роботи фітнес-клубів

ER-діаграма таблиць бази даних (функціональна схема)

ІАЛЦ.467200 Д2

Аркушів 1

Київ 2025 р



					ІАЛЦ.467200.005 Д2		
		№ докум.	Підпис	Дата			
Розробив		Шерстюк Д. М.			**Серверна частина веб-застосунку для організації роботи фітнес-клубів** Функціональна схема (діаграма класів)	Літ.	Аркуш 1 / Аркушів 1
Перевірив		Волокита А. М.					
Н. Контр.		Пономаренко А. М.				КПІ ім. Ігоря Сікорського, ФІОТ,ІМ-13	
Затвердив							

ДОДАТОК З

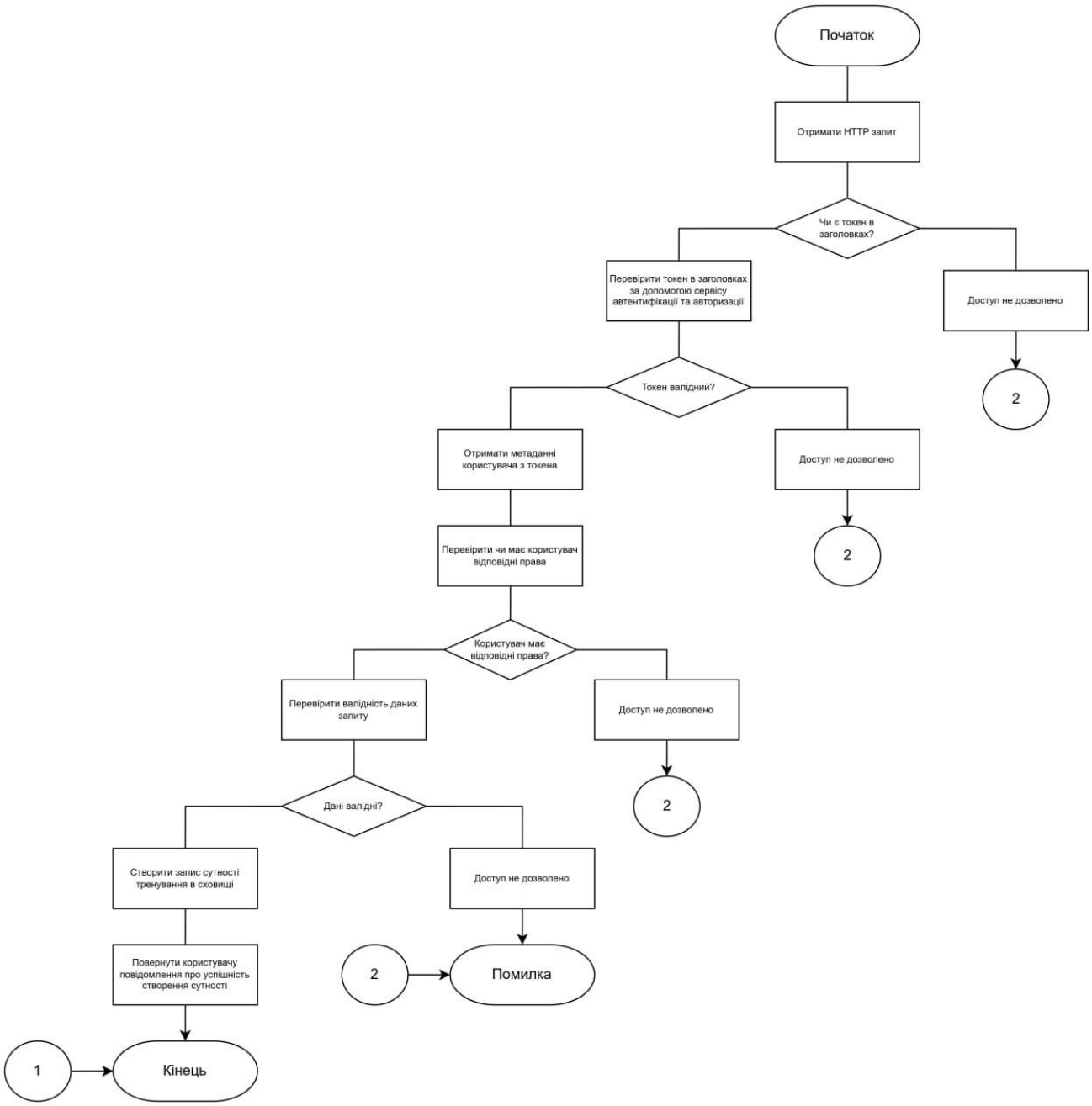
Серверна частина веб-застосунку для організації роботи фітнес-клубів

Алгоритм запису на тренування

ІАЛЦ.467200 Д3

Аркушів 1

Київ 2025 р



					ІАЛЦ.467200.006 ДЗ		
		№ докум.	Підпис	Дата			
Розробив		Шерстюк Д. М.			**Серверна частина веб-застосунку для організації роботи фітнес-клубів** Алгоритм дій програмного забезпечення	Літ.	Аркуш 1 / Аркушів 1
Перевірив		Волокита А. М.					
Н. Контр.		Пономаренко А. М.				КПІ ім. Ігоря Сікорського, ФІОТ,ІМ-13	
Затвердив							

ДОДАТОК 4

Серверна частина веб-застосунку для організації роботи фітнес-клубів

Текст програмного коду

ІАЛЦ.467200 Д4

Аркушів 50

Київ 2025 р

```
// ----- FILE: app.module.ts -----

import { Module } from '@nestjs/common';
import { MikroOrmModule } from '@mikro-orm/nestjs';
import { UsersModule } from './users/users.module';
import { TrainingsModule } from './trainings/trainings.module';
import { GroupsModule } from './groups/groups.module';
import { StatsModule } from './stats/stats.module';
import { ExerciseModule } from './exercise/exercise.module';
import MikroOrmConfig from './mikro-orm.config';
import { NotificationsModule } from './notifications/notifications.module';
import { ExerciseSetModule } from './exercise-set/exercise-set.module';
import { MembershipModule } from './membership/membership.module';
import { MonobankModule } from './monobank/monobank.module';

@Module({
  imports: [
    MikroOrmModule.forRoot(MikroOrmConfig),
    UsersModule,
    TrainingsModule,
    GroupsModule,
    StatsModule,
    ExerciseModule,
    NotificationsModule,
    StatsModule,
    ExerciseSetModule,
    MembershipModule,
    MonobankModule,
  ],
})
export class AppModule {}

// ----- FILE: config.ts -----

import { config as envConfig } from 'dotenv';

envConfig();

export const config = {
  redisUrl: process.env.REDIS_URL,
  jwtSecret: process.env.JWT_SECRET,
  apiUrl: process.env.MONOBANK_API_URL,
  redirectUrl: process.env.REDIRECT_URL,
  webhookUrl: `${process.env.BASE_API_URL}/membership/callback-mono`,
  monobankToken: process.env.MONOBANK_TOKEN,
};

// ----- FILE: main.ts -----

import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
```

```
import { DocumentBuilder, SwaggerModule } from '@nestjs/swagger';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);

  const config = new DocumentBuilder()
    .setTitle('Training API')
    .setDescription('Serverside training app')
    .setVersion('1.0')
    .addBearerAuth()
    .build();

  const document = SwaggerModule.createDocument(app, config);
  SwaggerModule.setup('/', app, document);

  await app.listen(process.env.PORT ?? 3000);
}
bootstrap();

// ----- FILE: mikro-orm.config.ts -----

import { defineConfig } from '@mikro-orm/core';
import { PostgreSqlDriver } from '@mikro-orm/postgresql';
import { config as envConfig } from 'dotenv';

envConfig();

export default defineConfig({
  entities: ['dist/**/*.entity.js'],
  entitiesTs: ['src/**/*.entity.ts'],
  dbName: process.env.DB_NAME,
  user: process.env.DB_USER,
  password: process.env.DB_PASSWORD,
  host: process.env.DB_HOST,
  port: Number(process.env.DB_PORT),
  driver: PostgreSqlDriver,
  allowGlobalContext: true,
  migrations: {
    tableName: 'orm_migrations',
    path: 'src/database/migrations',
    transactional: true,
    disableForeignKeys: true,
    allOrNothing: true,
    dropTables: true,
    safe: false,
    emit: 'ts',
  },
});

// ----- FILE: auth\auth.controller.ts -----

import { Body, Controller, Post } from '@nestjs/common';
import { AuthService } from './auth.service';
import { SignInDto } from './dto/sign-in.dto';
import { SignUpDto } from './dto/sign-up.dto';
import { ApiBody, ApiResponse, ApiTags } from '@nestjs/swagger';
import { JwtDto } from './dto/jwt.dto';
import { SignUpResultDto } from './dto/sign-up-result.dto';
@Controller('auth')
@ApiTags('auth')
export class AuthController {
  constructor(private readonly authService: AuthService) {}
```

```
  @Post('signin')
  @ApiBody({
  type: SignInDto,
  required: true,
    description: 'Credentials for sign in',
  })
  @ApiResponse({
    status: 201,
    description: 'JWT session token',
    type: JwtDto,
  })
  @ApiResponse({
    status: 400,
    description: 'Invalid username or password',
  })
  signin(@Body() dto: SignInDto): Promise<JwtDto> {
    return this.authService.signin(dto);
  }

  @Post('signup')
  @ApiBody({
  type: SignUpDto,
  required: true,
    description: 'Credentials for sign in',
  })
  @ApiResponse({
    status: 201,
    description: 'Sign up result',
    type: SignUpResultDto,
  })
  @ApiResponse({
    status: 400,
    description: 'User with this email already exists or invalid credentials',
  })
  signup(@Body() dto: SignUpDto): Promise<SignUpResultDto> {
    return this.authService.signup(dto);
  }
}

// ----- FILE: auth\auth.module.ts -----

import { Global, Module, forwardRef } from '@nestjs/common';
import { AuthService } from './auth.service';
import { UsersModule } from '../users/users.module';
import { AuthController } from './auth.controller';
import { AuthGuard } from './guards/auth.guard';
import { JwtModule } from '@nestjs/jwt';

@Module({
  imports: [
    forwardRef(() => UsersModule),
    JwtModule.register({
      secret: process.env.JWT_SECRET || 'secret',
      signOptions: { expiresIn: '1d' },
    }),
  ],
  providers: [AuthService, AuthGuard],
  controllers: [AuthController],
  exports: [AuthService, AuthGuard],
})
@Global()
export class AuthModule {}
```

```
// ----- FILE: auth\auth.service.ts -----

import { BadRequestException, Injectable } from '@nestjs/common';
import { UsersService } from '../users/users.service';
import { SignInDto } from './dto/sign-in.dto';
import { User } from '../users/entities/user.entity';
import * as bcrypt from 'bcrypt';
import { SignUpDto } from './dto/sign-up.dto';
import { SignUpResultDto } from './dto/sign-up-result.dto';
import { JwtDto } from './dto/jwt.dto';
import { JwtService } from '@nestjs/jwt';
import { UserRole } from '../users/enums/user-role.enum';

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

  async signup(dto: SignUpDto): Promise<SignUpResultDto> {
    const user = await this.usersService.findOne({ email: dto.email });

    if (user) {
      throw new BadRequestException('User with this email already exists');
    }

    const newUser = new User();

    newUser.email = dto.email;
    newUser.fullName = dto.fullName;
    newUser.role = UserRole.USER;

    const salt = await bcrypt.genSalt();
    const hashed = await bcrypt.hash(dto.password, salt);

    newUser.passwordHash = hashed;
    newUser.passwordSalt = salt;

    await this.usersService.createOne(newUser);

    return {
      success: true,
    };
  }

  async signin(dto: SignInDto): Promise<JwtDto> {
    const user = await this.usersService.findOne({ email: dto.email });

    if (!user) {
      throw new BadRequestException('Invalid username or password');
    }

    const match = await bcrypt.compare(dto.password, user.passwordHash);

    if (!match) {
      throw new BadRequestException('Invalid username or password');
    }

    const payload = {
      sub: user.id,
      role: user.role,
      email: user.email,
```

```
    };

    return {
      accessToken: this.jwtService.sign(payload),
    };
  }

  validate(token: string): { sub: number; role: UserRole; email: string } {
    return this.jwtService.verify(token);
  }
}

// ----- FILE: auth\decorator\required-role.decorator.ts -----

import { UserRole } from '../../users/enums/user-role.enum';
import { SetMetadata } from '@nestjs/common';

export function RequiredRole(role: UserRole) {
  return SetMetadata('requiredRole', role);
}

// ----- FILE: auth\decorator\user-meta.decorator.ts -----

import { createParamDecorator, ExecutionContext } from '@nestjs/common';
import { UserMetadata } from '../types/user-metadata.type';

export const UserMeta = createParamDecorator(
  (data: unknown, ctx: ExecutionContext) => {
  const req = ctx.switchToHttp().getRequest();
  return {
      userId: req.userId,
      userEmail: req.userEmail,
      userRole: req.userRole,
      userIsTrainer:  req.userIsTrainer,
      userTrainerId: req.userTrainerId,
      userGroup: req.userGroup,
    } as UserMetadata;
  },
);

// ----- FILE: auth\guards\auth.guard.ts -----

import {
  CanActivate,
  ExecutionContext,
  ForbiddenException,
  Inject,
  UnauthorizedException,
} from '@nestjs/common';
import { AuthService } from '../auth.service';

export class AuthGuard implements CanActivate {
  constructor(
    @Inject(AuthService)
    private readonly authService: AuthService,
  ) {}

  canActivate(context: ExecutionContext): boolean {
    const request = context.switchToHttp().getRequest();

    if (!('authorization' in request.headers)) {
      throw new UnauthorizedException();
    }
```

```
    const token = request.headers.authorization.split(' ')[1];

    let payload;

    try {
      payload = this.authService.validate(token);
    } catch (e) {
      throw new ForbiddenException(`Invalid token: ${e.message}`);
    }

    request.userId = payload.sub;
    request.userRole = payload.role;
    request.userEmail = payload.email;

    return true;
  }
}

// ----- FILE: auth\guards\roles.guard.ts -----

import { CanActivate, ExecutionContext, Inject } from '@nestjs/common';
import { Reflector } from '@nestjs/core';
import { UserRole } from '../../users/enums/user-role.enum';

export class RolesGuard implements CanActivate {
  constructor(
    @Inject(Reflector)
    private readonly reflector: Reflector,
  ) {}

  canActivate(context: ExecutionContext): boolean {
    const requiredRole = this.reflector.getAllAndOverride<UserRole>(
      'requiredRole',
      [context.getHandler(), context.getClass()],
    );

    if (!requiredRole) {
      return true;
    }

    const request = context.switchToHttp().getRequest();

    return request.userRole === requiredRole;
  }
}

// ----- FILE: auth\types\user-metadata.type.ts -----

import { UserRole } from '../../users/enums/user-role.enum';

export type UserMetadata = {
  userId: number;
  userEmail: string;
  userRole: UserRole;
};

// ----- FILE: cache\cache.injections.ts -----

export const REDIS = 'REDIS';

// ----- FILE: cache\cache.module.ts -----
```

```
import { DynamicModule, Module } from '@nestjs/common';
import * as Redis from 'ioredis';

import { CacheService } from './cache.service';
import { REDIS } from './cache.injections';
import { config } from '../config';

@Module({})
export class CacheModule {
  static forRoot(redisURL: string): DynamicModule {
    return {
      module: CacheModule,
      providers: [
      CacheService,
        {
          provide: REDIS,
          useFactory: () => {
            const redisUrl = new URL(redisURL);
            const options = {
              host: redisUrl.hostname,
              port: Number(redisUrl.port),
              password: redisUrl.username,
              tls: redisUrl.protocol == 'rediss:' ? {} : undefined,
              maxRetriesPerRequest: 1000,
              retryStrategy: () => 1000,
              reconnectOnError: function (err) {
              console.error(err.message);
                const targetError = 'READONLY';
                if (err.message.slice(0, targetError.length) === targetError) {
                  return 2;
                }
              },
            } as Redis.RedisOptions;

            return new Redis.Redis(options);
          },
        },
      ],
      exports: [CacheService],
      global: true,
    };
  }

  static forRootFromConfig(): DynamicModule {
    if (!config.redisUrl) {
      throw new Error('Redis URL is not configured');
    }
    return CacheModule.forRoot(config.redisUrl);
  }
}

// ----- FILE: cache\cache.service.ts -----

import {
  Inject,
  Injectable,
  Logger,
  OnApplicationShutdown,
} from '@nestjs/common';
import Redis from 'ioredis';

import { REDIS } from './cache.injections';
```

```
@Injectable()
export class CacheService implements OnApplicationShutdown {
  private readonly logger: Logger = new Logger(CacheService.name);
  constructor(
    @Inject(REDIS)
    private readonly redisClient: Redis,
  ) {}

  async onApplicationShutdown(): Promise<void> {
    await this.redisClient.quit();
  }

  async get<T>(key: string): Promise<T | null> {
    try {
      const result = await this.redisClient.get(key);
      if (!result) {
        return null;
      }
      return JSON.parse(result) as T;
    } catch (error) {
      this.logger.error(`REDIS GET FAILED: ${error.message}`);
      return null;
    }
  }

  async set<T>(key: string, value: T, ttl?: number): Promise<void> {
    try {
      if (ttl) {
        await this.redisClient.set(key, JSON.stringify(value), 'EX', ttl);
        return;
      }
      await this.redisClient.set(key, JSON.stringify(value));
    } catch (error) {
      this.logger.error(`REDIS SET FAILED: ${error.message}`);
    }
  }

  async wrap<T>(
    key: string,
    fn: Promise<T> | (() => T) | (() => Promise<T>),
    ttl: number,
  ): Promise<T> {
    const value = await this.get<T>(key);
    if (value) {
      return value;
    }
    const result = await (typeof fn === 'function' ? fn() : fn);
    await this.set(key, result, ttl);
    return result;
  }

  async delete(key: string): Promise<void> {
    try {
      await this.redisClient.del(key);
    } catch (error) {
      this.logger.error(`REDIS DELETE FAILED: ${error.message}`);
    }
  }

  async deletePattern(pattern: string): Promise<void> {
    try {
      const keys = await this.redisClient.keys(pattern);
```

```
      if (!keys.length) {
        return;
      }

      await this.redisClient.del(...keys);
    } catch (error) {
      this.logger.error(`REDIS DELETE PATTERN FAILED: ${error.message}`);
    }
  }
}

// ----- FILE: cache\decorators\through-cache.decorator.ts -----

import * as md5 from 'md5';

// Method decorator that caches the result of the method in the cache service for the specified ttl.
// CacheService must be injected in the class constructor where decorator is used.
// The key is generated from the method name and the arguments passed to it (md5 hash of the stringified arguments)
export function ThroughCache(ttl?: number): MethodDecorator {
  return function (
    target: any,
    propertyKey: string,
    descriptor: PropertyDescriptor,
  ) {
    const originalMethod = descriptor.value;

    descriptor.value = async function (...args: unknown[]) {
      const className = target.constructor.name;
      const key = `${className}:${propertyKey}:${md5(
        JSON.stringify(args),
      ).slice(0, 8)}`;

      if (!this.cacheService) {
        throw new Error(
          'CacheService not found in context. It must be injected in the class constructor where decorator is used.',
        );
      }

      return this.cacheService.wrap(key, originalMethod.apply(this, args), ttl);
    };
  };
}

// ----- FILE: common\basic-crud.service.ts -----

import { BasicEntity } from './basic-entity';
import { EntityRepository } from '@mikro-orm/postgresql';
import { CacheService } from '../cache/cache.service';
import {
  EntityManager,
  FilterQuery,
  FindOptions,
  RequiredEntityData,
  EntityData,
  FindOneOptions,
  FindOneOrFailOptions,
} from '@mikro-orm/core';
import { ThroughCache } from '../cache/decorators/through-cache.decorator';
import { NotFoundException } from '@nestjs/common';

export class BasicCrudService<T extends BasicEntity> {
  constructor(
    protected readonly entityClass: new () => T,
```

```
  protected readonly entityRepository: EntityRepository<T>,
  protected readonly cacheService: CacheService,
  protected readonly entityManager: EntityManager,
) {}

async findOne(
  args: FilterQuery<T>,
  options?: FindOneOptions<T>,
): Promise<T | null> {
  return await this.entityRepository.findOne(args, options);
}

async findOneOrFail(
  args: FilterQuery<T>,
  options?: FindOneOrFailOptions<T>,
): Promise<T> {
  try {
    return await this.entityRepository.findOneOrFail(args, options);
  } catch {
    throw new NotFoundException(`${this.entityClass.name} not found`);
  }
}

@ThroughCache(60)
async findOneCached(
args: FilterQuery<T>,
  options?: FindOneOptions<T>,
): Promise<T | null> {
  return await this.findOne(args, options);
}

async findMany(args: FilterQuery<T>, options?: FindOptions<T>): Promise<T[]> {
  return await this.entityRepository.find(args, options);
}

@ThroughCache(60)
async findManyCached(
args: FilterQuery<T>,
  options?: FindOptions<T>,
): Promise<T[]> {
  return await this.findMany(args, options);
}

async upsert(entity: RequiredEntityData<T>): Promise<T | null> {
  const newEntity = this.entityRepository.create(entity);

  await Promise.all([
    this.flushCrudCache(),
    this.entityManager.upsert(newEntity),
  ]);
  return await this.findOne(newEntity as FilterQuery<T>);
}

async createOne(data: RequiredEntityData<T>): Promise<T | null> {
  const entity = this.entityRepository.create(data);

  await Promise.all([
    this.flushCrudCache(),
    this.entityManager.persistAndFlush(entity as T),
  ]);

  return await this.findOne(entity as FilterQuery<T>);
}
```

```
  async updateOne(
    args: FilterQuery<T>,
    data: EntityData<T>,
  ): Promise<T | null> {
    const entity = await this.findOne(args);

    if (!entity) {
      throw new NotFoundException(
        `No ${this.entityClass.name} found to update`,
      );
    }

    await Promise.all([
      this.flushCrudCache(),
      this.entityRepository.nativeUpdate(args,  data),
    ]);

    return await this.findOne(entity);
  }

  async deleteOne(args: FilterQuery<T>): Promise<T | null> {
    const entity = await this.findOne(args);

    if (!entity) {
      throw new NotFoundException(
        `No ${this.entityClass.name} found to delete`,
      );
    }

    await Promise.all([
      this.flushCrudCache(),
      this.entityManager.remove(entity).flush(),
    ]);

    return entity;
  }

  async flushCrudCache(): Promise<void> {
    return await this.cacheService.deletePattern(`${this.constructor.name}:*`);
  }
}

// ----- FILE: common\basic-entity.ts -----

import { wrap, EntityDTO } from '@mikro-orm/core';

export class BasicEntity {
  toSafeEntity(
    ignoreFields: (keyof this)[] = [],
  ): Omit<EntityDTO<this>, keyof this> {
    return wrap(this).toObject(ignoreFields as string[]);
  }
}

// ----- FILE: exercise\exercise.controller.ts -----

import { AuthGuard } from '../auth/guards/auth.guard';
import {
  Body,
  Controller,
  Delete,
  Get,
```

```
  Param,
  ParseIntPipe,
  Patch,
  Post,
  Query,
  UseGuards,
} from '@nestjs/common';
import { ApiBearerAuth, ApiOkResponse, ApiTags } from '@nestjs/swagger';
import { ExerciseService } from './exercise.service';
import { FindExerciseArgs } from './args/find-exercise.args';
import { CreateExerciseDto } from './dto/create-exercise.dto';
import { UpdateExerciseDto } from './dto/update-exercise.dto';

@Controller('exercise')
@UseGuards(AuthGuard)
@ApiTags('exercise')
export class ExerciseController {
  constructor(private readonly exerciseService: ExerciseService) {}

  @Get()
  @ApiBearerAuth()
  @ApiOkResponse({
    description: 'Returns all exercises',
  })
  findAll(@Query() args: FindExerciseArgs) {
    return this.exerciseService.findAll(args);
  }

  @ApiBearerAuth()
  @Get(':id')
  @ApiOkResponse({
    description: 'Returns an exercise',
  })
  findOne(@Param('id', ParseIntPipe) id: number) {
    return this.exerciseService.findOneSafe(id);
  }

  @Post()
  @ApiBearerAuth()
  @ApiOkResponse({
    description: 'Creates an exercise',
  })
  createExercise(@Body() dto: CreateExerciseDto) {
    return this.exerciseService.create(dto);
  }

  @ApiBearerAuth()
  @Patch()
  @ApiOkResponse({
    description: 'Updates an exercise',
  })
  update(@Body() dto: UpdateExerciseDto) {
    return this.exerciseService.update(dto);
  }

  @ApiBearerAuth()
  @Delete(':id')
  @ApiOkResponse({
    description: 'Deletes an exercise',
  })
  remove(@Param('id', ParseIntPipe) id: number) {
    return this.exerciseService.remove(id);
  }
```

```
}

// ----- FILE: exercise\exercise.module.ts -----

import { Module } from '@nestjs/common';
import { ExerciseService } from './exercise.service';
import { ExerciseController } from './exercise.controller';
import { MikroOrmModule } from '@mikro-orm/nestjs';
import { CacheModule } from '../cache/cache.module';
import { Exercise } from './entities/exercise.entity';

@Module({
  imports: [
    MikroOrmModule.forFeature([Exercise]),
    CacheModule.forRootFromConfig(),
  ],
  controllers: [ExerciseController],
  providers: [ExerciseService],
  exports: [ExerciseService],
})
export class ExerciseModule {}

// ----- FILE: exercise\exercise.service.ts -----

import { Injectable } from '@nestjs/common';
import { CacheService } from '../cache/cache.service';
import { EntityManager, FilterQuery } from '@mikro-orm/core';
import { ExerciseRepository } from './repositories/exercise.repository';
import { Exercise } from './entities/exercise.entity';
import { BasicCrudService } from '../common/basic-crud.service';
import { CreateExerciseDto } from './dto/create-exercise.dto';
import { FindExerciseArgs } from './args/find-exercise.args';
import { UpdateExerciseDto } from './dto/update-exercise.dto';

@Injectable()
export class ExerciseService extends BasicCrudService<Exercise> {
  constructor(
    protected readonly cacheService: CacheService,
    protected readonly exerciseRepository: ExerciseRepository,
    protected readonly entityManager: EntityManager,
  ) {
    super(Exercise, exerciseRepository, cacheService, entityManager);
  }

  async findAll(args: FindExerciseArgs) {
    const filter: FilterQuery<Exercise> = {};

    if (args.name) {
      filter.name = { $eq: args.name };
    }

    if (args.startWeight) {
      filter.startWeight = { $eq: args.startWeight };
    }

    if (args.endWeight) {
      filter.endWeight = { $eq: args.endWeight };
    }

    if (args.weightIncrement) {
      filter.weightIncrement = { $eq: args.weightIncrement };
    }
```

```
    if (args.setsNum) {
      filter.setsNum = { $eq: args.setsNum };
    }

    if (args.repsNum) {
      filter.repsNum = { $eq: args.repsNum };
    }

    return this.findMany(filter);
  }

  async findOneSafe(id: number) {
    const filter: FilterQuery<Exercise> = { id };

    return this.findOneOrFail(filter);
  }

  async create(dto: CreateExerciseDto) {
    return this.createOne({
      ...dto,
      exerciseSets: [],
    });
  }

  async update(dto: UpdateExerciseDto) {
    const { id } = dto;
    const filter: FilterQuery<Exercise> = { id };

    return this.updateOne(filter, dto);
  }

  async remove(id: number) {
    const filter: FilterQuery<Exercise> = { id };

    return this.deleteOne(filter);
  }
}

// ----- FILE: exercise\entities\exercise.entity.ts -----

import { Entity, ManyToMany, PrimaryKey, Property } from '@mikro-orm/core';
import { ExerciseRepository } from '../repositories/exercise.repository';
import { BasicEntity } from '../../common/basic-entity';
import { ApiProperty } from '@nestjs/swagger';
import { ExerciseSet } from '../../exercise-set/entities/exercise-set.entity';
@Entity({ repository: () => ExerciseRepository })
export class Exercise extends BasicEntity {
  @PrimaryKey({ autoincrement: true })
  @ApiProperty()
  id: number;

  @Property()
  @ApiProperty()
  name: string;

  @Property()
  @ApiProperty()
  startWeight: number;

  @Property()
  @ApiProperty()
  endWeight: number;
```

```
  @Property()
  @ApiProperty()
  weightIncrement: number;

  @Property()
  @ApiProperty()
  setsNum: number;

  @Property()
  @ApiProperty()
  repsNum: number;

  @ManyToMany(() => ExerciseSet)
  @ApiProperty()
  exerciseSets: ExerciseSet[];
}

// ----- FILE: exercise\repositories\exercise.repository.ts -----

import { EntityRepository } from '@mikro-orm/postgresql';
import { Exercise } from '../entities/exercise.entity';

export class ExerciseRepository extends EntityRepository<Exercise> {}

// ----- FILE: exercise-set\exercise-set.controller.ts -----

import { AuthGuard } from '../auth/guards/auth.guard';
import {
  Body,
  Controller,
  Delete,
  Get,
  Param,
  ParseIntPipe,
  Patch,
  Post,
  Query,
  UseGuards,
} from '@nestjs/common';
import { ApiBearerAuth, ApiOkResponse, ApiTags } from '@nestjs/swagger';
import { ExerciseSetService } from './exercise-set.service';
import { FindExerciseSetArgs } from './args/find-exercise-set.args';
import { CreateExerciseSetDto } from './dto/create-exercise-set.dto';
import { UpdateExerciseSetDto } from './dto/update-exercise-set.dto';

@Controller('exerc-set')
@UseGuards(AuthGuard)
@ApiTags('exerc-set')
export class ExerciseSetController {
  constructor(private readonly exerciseSetService: ExerciseSetService) {}

  @Get()
  @ApiBearerAuth()
  @ApiOkResponse({
    description: 'Returns all exercise sets',
  })
  findAll(@Query() args: FindExerciseSetArgs) {
    return this.exerciseSetService.findAll(args);
  }

  @ApiBearerAuth()
  @Get(':id')
  @ApiOkResponse({
```

```
    description: 'Returns an exercise set',
  })
  findOne(@Param('id', ParseIntPipe) id: number) {
    return this.exerciseSetService.findOneSafe(id);
  }

  @Post()
  @ApiBearerAuth()
  @ApiOkResponse({
    description: 'Creates an exercise set',
  })
  createExerciseSet(@Body() dto: CreateExerciseSetDto) {
    return this.exerciseSetService.create(dto);
  }

  @ApiBearerAuth()
  @Patch()
  @ApiOkResponse({
    description: 'Updates an exercise set',
  })
  update(@Body() dto: UpdateExerciseSetDto) {
    return this.exerciseSetService.update(dto);
  }

  @ApiBearerAuth()
  @Delete(':id')
  @ApiOkResponse({
    description: 'Deletes an exercise set',
  })
  remove(@Param('id', ParseIntPipe) id: number) {
    return this.exerciseSetService.remove(id);
  }
}

// ----- FILE: exercise-set\exercise-set.module.ts -----

import { Module } from '@nestjs/common';
import { ExerciseSetService } from './exercise-set.service';
import { ExerciseSetController } from './exercise-set.controller';
import { MikroOrmModule } from '@mikro-orm/nestjs';
import { CacheModule } from '../cache/cache.module';
import { ExerciseModule } from '../exercise/exercise.module';
import { ExerciseSet } from './entities/exercise-set.entity';

@Module({
  imports: [
    MikroOrmModule.forFeature([ExerciseSet]),
    CacheModule.forRootFromConfig(),
    ExerciseModule,
  ],
  controllers: [ExerciseSetController],
  providers: [ExerciseSetService],
  exports: [ExerciseSetService],
})
export class ExerciseSetModule {}

// ----- FILE: exercise-set\exercise-set.service.ts -----

import { Injectable } from '@nestjs/common';
import { CacheService } from '../cache/cache.service';
import { EntityManager, FilterQuery } from '@mikro-orm/core';
import { BasicCrudService } from '../common/basic-crud.service';
import { ExerciseService } from '../exercise/exercise.service';
```

```
import { ExerciseSet } from './entities/exercise-set.entity';
import { ExerciseSetRepository } from './repositories/exercise-set.repository';
import { FindExerciseSetArgs } from './args/find-exercise-set.args';
import { CreateExerciseSetDto } from './dto/create-exercise-set.dto';
import { UpdateExerciseSetDto } from './dto/update-exercise-set.dto';

@Injectable()
export class ExerciseSetService extends BasicCrudService<ExerciseSet> {
 constructor(
  protected readonly cacheService: CacheService,
  protected readonly exerciseSetRepository: ExerciseSetRepository,
  protected readonly entityManager: EntityManager,
  protected readonly exerciseService: ExerciseService,
 ) {
  super(ExerciseSet, exerciseSetRepository, cacheService, entityManager);
 }

 async findAll(args: FindExerciseSetArgs) {
  const filter: FilterQuery<ExerciseSet> = {};

  if (args.name) {
   filter.name = { $eq: args.name };
  }

  return this.findMany(filter);
 }

 async findOneSafe(id: number) {
  const filter: FilterQuery<ExerciseSet> = { id };

  return this.findOneOrFail(filter);
 }

 async create(dto: CreateExerciseSetDto) {
  const exercises = await Promise.all(
   dto.exerciseIds.map((id) => this.exerciseService.findOneSafe(id)),
  );

  return this.createOne({
   name: dto.name,
   exercises,
   trainings: [],
  });
 }

 async update(dto: UpdateExerciseSetDto) {
  const { id, exerciseIds, ...rest } = dto;
  const filter: FilterQuery<ExerciseSet> = { id };

  const updateData: Partial<ExerciseSet> = { ...rest };

  if (exerciseIds) {
   const exercises = await Promise.all(
    exerciseIds.map((id) => this.exerciseService.findOneSafe(id)),
   );

   updateData.exercises = exercises;
  }

  return this.updateOne(filter, updateData);
 }

 async remove(id: number) {
```

```
    const filter: FilterQuery<ExerciseSet> = { id };

    return this.deleteOne(filter);
  }

  async getExerciseIds(setId: number): Promise<number[]> {
    const exerciseSet = await this.findOneSafe(setId);
    await this.entityManager.populate(exerciseSet, ['exercises']);
    return exerciseSet.exercises.map((exercise) => exercise.id);
  }
}

// ----- FILE: exercise-set\entities\exercise-set.entity.ts -----

import {
  Entity,
  ManyToMany,
  OneToMany,
  PrimaryKey,
  Property,
} from '@mikro-orm/core';
import { BasicEntity } from '../../common/basic-entity';
import { ApiProperty } from '@nestjs/swagger';
import { Training } from '../../trainings/entities/training.entity';
import { Exercise } from '../../exercise/entities/exercise.entity';
import { ExerciseSetRepository } from '../repositories/exercise-set.repository';

@Entity({ repository: () => ExerciseSetRepository })
export class ExerciseSet extends BasicEntity {
@PrimaryKey({ autoincrement: true })
@ApiProperty()
  id: number;

  @Property()
  @ApiProperty()
  name: string;

  @ManyToMany(() => Exercise, (exercise) => exercise.exerciseSets, {
    mappedBy: 'exerciseSets',
  })
  @ApiProperty()
  exercises: Exercise[];

  @OneToMany(() => Training, (training) => training.ExerciseSetId)
  @ApiProperty()
  trainings: Training[];
}

// ----- FILE: exercise-set\repositories\exercise-set.repository.ts -----

import { EntityRepository } from '@mikro-orm/postgresql';
import { ExerciseSet } from '../entities/exercise-set.entity';

export class ExerciseSetRepository extends EntityRepository<ExerciseSet> {}

// ----- FILE: groups\groups.controller.ts -----

import { AuthGuard } from '../auth/guards/auth.guard';
import {
  Body,
  Controller,
  Delete,
  Get,
```

```
  Param,
  ParseIntPipe,
  Patch,
  Post,
  Query,
  UseGuards,
} from '@nestjs/common';
import { ApiBearerAuth, ApiOkResponse, ApiTags } from '@nestjs/swagger';
import { GroupsService } from './groups.service';
import { FindGroupArgs } from './args/find-group.args';
import { CreateGroupDto } from './dto/create-group.dto';
import { UpdateGroupDto } from './dto/update-group.dto';
import { RolesGuard } from '../auth/guards/roles.guard';
import { RequiredRole } from '../auth/decorator/required-role.decorator';
import { UserRole } from '../users/enums/user-role.enum';
import { UserMeta } from '../auth/decorator/user-meta.decorator';
import { UserMetadata } from '../auth/types/user-metadata.type';

@Controller('groups')
@UseGuards(AuthGuard)
@ApiTags('groups')
export class GroupsController {
  constructor(private readonly groupsService: GroupsService) {}

  @Get()
  @ApiBearerAuth()
  @UseGuards(RolesGuard)
  @RequiredRole(UserRole.ADMIN)
  @ApiOkResponse({
    description: 'Returns all groups',
  })
  findAll(@Query() args: FindGroupArgs) {
    return this.groupsService.findAll(args);
  }

  @ApiBearerAuth()
  @Get(':id')
  @ApiOkResponse({
  description: 'Creates a group',
  })
  findOne(@Param('id', ParseIntPipe) id: number) {
    return this.groupsService.findOneSafe(id);
  }

  @Post()
  @ApiBearerAuth()
  @ApiOkResponse({
    description: 'Creates a group',
  })
  createGroup(@Body() dto: CreateGroupDto, @UserMeta() meta: UserMetadata) {
    return this.groupsService.create(dto, meta.userId);
  }

  @ApiBearerAuth()
  @Patch()
  @UseGuards(RolesGuard)
  @RequiredRole(UserRole.ADMIN)
  @ApiOkResponse({
    description: 'Updates a group',
  })
  update(@Body() dto: UpdateGroupDto, @UserMeta() meta: UserMetadata) {
    return this.groupsService.update(dto, meta);
  }
```

```
  @ApiBearerAuth()
  @Delete(':id')
  @ApiOkResponse({
  description: 'Deletes a group',
  })
  remove(
    @Param('id', ParseIntPipe) id: number,
    @UserMeta() meta: UserMetadata,
  ) {
    return this.groupsService.remove(id, meta);
  }
}

// ----- FILE: groups\groups.module.ts -----

import { Module, forwardRef } from '@nestjs/common';
import { GroupsController } from './groups.controller';
import { GroupsService } from './groups.service';
import { MikroOrmModule } from '@mikro-orm/nestjs';
import { Group } from './entities/group.entity';
import { CacheModule } from '../cache/cache.module';
import { UsersModule } from '../users/users.module';

@Module({
  imports: [
    MikroOrmModule.forFeature([Group]),
    CacheModule.forRootFromConfig(),
    forwardRef(() => UsersModule),
  ],
  controllers: [GroupsController],
  providers: [GroupsService],
  exports: [GroupsService],
})
export class GroupsModule {}

// ----- FILE: groups\groups.service.ts -----

import { Injectable, Inject, forwardRef } from '@nestjs/common';
import { CacheService } from '../cache/cache.service';
import { EntityManager, FilterQuery } from '@mikro-orm/core';
import { GroupRepository } from './repositories/group.repository';
import { Group } from './entities/group.entity';
import { BasicCrudService } from '../common/basic-crud.service';
import { CreateGroupDto } from './dto/create-group.dto';
import { FindGroupArgs } from './args/find-group.args';
import { UpdateGroupDto } from './dto/update-group.dto';
import { UsersService } from '../users/users.service';
import { UserMetadata } from '../auth/types/user-metadata.type';

@Injectable()
export class GroupsService extends BasicCrudService<Group> {
  constructor(
    protected readonly cacheService: CacheService,
    protected readonly groupRepository: GroupRepository,
    protected readonly entityManager: EntityManager,
    @Inject(forwardRef(() => UsersService))
    private readonly usersService: UsersService,
  ) {
    super(Group, groupRepository, cacheService, entityManager);
  }

  async findAll(args: FindGroupArgs) {
```

```
    const filter: FilterQuery<Group> = {};

    if (args.name) {
      filter.name = { $eq: args.name };
    }

    return this.findMany(filter);
  }

  async findOneSafe(id: number) {
    const filter: FilterQuery<Group> = { id };

    return this.findOneOrFail(filter);
  }

  async create(dto: CreateGroupDto, creatorId: number) {
    const creator = await this.usersService.findOne(creatorId);
    if (!creator) {
      throw new Error(`User with id ${creatorId} not found`);
    }
    return this.createOne({
      ...dto,
      users: [],
      creator: creator,
    });
  }

  async update(dto: UpdateGroupDto, meta: UserMetadata) {
    const group = await this.findOneSafe(dto.id);
    if (meta.userRole == 'ADMIN' || group.creator.id == meta.userId) {
      const { id } = dto;
      const filter: FilterQuery<Group> = { id };

      return this.updateOne(filter, dto);
    }
    throw new Error('You are not allowed to update this group');
  }

  async remove(id: number, meta: UserMetadata) {
    const group = await this.findOneSafe(id);
    if (meta.userRole == 'ADMIN' || group.creator.id == meta.userId) {
      const filter: FilterQuery<Group> = { id };

      return this.deleteOne(filter);
    }
    throw new Error('You are not allowed to delete this group');
  }

  async findAllMembers(group: Group) {
    return this.usersService.complexSearch({ group_id: group.id });
  }
}

// ----- FILE: groups\entities\group.entity.ts -----

import {
  Entity,
  OneToMany,
  OneToOne,
  PrimaryKey,
  Property,
} from '@mikro-orm/core';
import { BasicEntity } from '../../common/basic-entity';
```

```
import { ApiProperty } from '@nestjs/swagger';
import { GroupRepository } from '../repositories/group.repository';
import { User } from '../../users/entities/user.entity';

@Entity({ repository: () => GroupRepository })
export class Group extends BasicEntity {
@PrimaryKey({ autoincrement: true })
@ApiProperty()
  id: number;

  @OneToMany({ entity: () => User, mappedBy: 'group' })
  @ApiProperty()
  users: User[];

  @Property()
  @ApiProperty()
  name: string;

  @OneToOne({ entity: () => User, nullable: true })
  @ApiProperty({ type: () => User })
  creator: User;
}

// ----- FILE: groups\repositories\group.repository.ts -----

import { EntityRepository } from '@mikro-orm/postgresql';
import { Group } from '../entities/group.entity';

export class GroupRepository extends EntityRepository<Group> {}

// ----- FILE: membership\membership.controller.ts -----

import {
  Body,
  Controller,
  Post,
  HttpCode,
  HttpStatus,
  UseGuards,
  Get,
} from '@nestjs/common';
import { MembershipService } from './membership.service';
import { CreateMembershipDto } from './dto/create-membership.dto';
import { ApiBearerAuth, ApiOkResponse, ApiTags } from '@nestjs/swagger';
import { AuthGuard } from '../auth/guards/auth.guard';
import { UserMetadata } from '../auth/types/user-metadata.type';
import { UserMeta } from '../auth/decorator/user-meta.decorator';
import { MonobankWebhookGuard } from '../monobank/guards/monobank-webhook.guard';

@Controller('membership')
@ApiTags('membership')
export class MembershipController {
  constructor(private readonly membershipService: MembershipService) {}

  @Post('membership-invoice')
  @UseGuards(AuthGuard)
  @ApiBearerAuth()
  @ApiOkResponse({
    description: 'Creates a membership invoice for the current user',
  })
  async create(
    @Body() dto: CreateMembershipDto,
    @UserMeta() meta: UserMetadata,
```

```
  ) {
    return this.membershipService.create(dto, meta);
  }

  @Post('callback-mono')
  @UseGuards(MonobankWebhookGuard)
  @HttpCode(HttpStatus.OK)
  async handleWebhook(@Body() body: any) {
    await this.membershipService.handleWebhook(body);
    return { ok: true };
  }

  @Get('mine')
  @UseGuards(AuthGuard)
  @ApiBearerAuth()
  @ApiOkResponse({
    description: 'Returns the membership for the current user',
  })
  async getMembership(@UserMeta() meta: UserMetadata) {
    return this.membershipService.getMembership(meta);
  }
}

// ----- FILE: membership\membership.module.ts -----

import { Module } from '@nestjs/common';
import { MembershipService } from './membership.service';
import { MembershipController } from './membership.controller';
import { MonobankModule } from '../monobank/monobank.module';
import { MikroOrmModule } from '@mikro-orm/nestjs';
import { Membership } from './entity/membership.entity';
import { UsersModule } from '../users/users.module';

@Module({
  imports: [
    MikroOrmModule.forFeature([Membership]),
    MonobankModule,
    UsersModule,
  ],
  controllers: [MembershipController],
  providers: [MembershipService],
})
export class MembershipModule {}

// ----- FILE: membership\membership.service.ts -----

import {
  Injectable,
  Logger,
  Inject,
  BadRequestException,
} from '@nestjs/common';
import { InjectRepository } from '@mikro-orm/nestjs';
import { EntityManager } from '@mikro-orm/core';
import { CreateMembershipDto } from './dto/create-membership.dto';
import { UsersService } from '../users/users.service';
import { MembershipStatus } from './enum/membership-status.enum';
import { MonobankClient } from '../monobank/monobank-client';
import { Membership } from './entity/membership.entity';
import { MONOBANK_CONFIG, UAN_CCY } from '../monobank/constants';
import { IMonobankConfig } from '../monobank/interface/monobank-config.interface';
import { BasicCrudService } from '../common/basic-crud.service';
import { MembershipRepository } from './repositories/membership.repository';
```

```
import { CacheService } from '../cache/cache.service';
import { MONTHLY_MEMBERSHIP_PRICE } from './constants';
import { UserMetadata } from '../auth/types/user-metadata.type';

@Injectable()
export class MembershipService extends BasicCrudService<Membership> {
  private readonly logger = new Logger(MembershipService.name);

  constructor(
    @Inject(MONOBANK_CONFIG) private readonly config: IMonobankConfig,
    @InjectRepository(Membership)
    protected readonly membershipRepository: MembershipRepository,
    private readonly userService: UsersService,
    private readonly monobankClient: MonobankClient,
    protected readonly cacheService: CacheService,
    protected readonly entityManager: EntityManager,
  ) {
    super(Membership, membershipRepository, cacheService, entityManager);
  }

  async create(
    dto: CreateMembershipDto,
    meta: UserMetadata,
  ): Promise<{ paymentUrl: string }> {
    const user = await this.userService.findOneOrFail(meta.userId);
    const membership = await this.getMembership(meta);

    if (membership && membership.status == 'active') {
      throw new BadRequestException('You have already paid for membership');
    }

    const amount = dto.monthNum * MONTHLY_MEMBERSHIP_PRICE;

    const invoicePayload = {
      amount: amount,
      ccy: UAN_CCY,
      redirectUrl: this.config.redirectUrl,
      merchantPaymInfo: {
        reference: user.id.toString(),
        destination: `Оплата абонемента`,
        comment: `Абонемент для користувача ID: ${meta.userId}`,
      },
      webhookUrl: this.config.webhookUrl,
    };

    const { invoiceId, pageUrl } = await this.monobankClient.invoiceCreate({
      ...invoicePayload,
      token: this.config.monobankToken,
    });

    const now = new Date();
    await this.createOne({
    user,
      amount: amount,
      startDate: undefined,
      endDate: undefined,
      status: MembershipStatus.PENDING,
      invoiceId,
      paymentUrl: pageUrl,
      createdAt: now,
      updatedAt: now,
    });
```

```
    return { paymentUrl: pageUrl };
  }

  async handleWebhook(data: any): Promise<void> {
    this.logger.debug(
      `Got data from monobank webhook: ${JSON.stringify(data)}`,
    );
    const { invoiceId, status, amount } = data;

    const membership = await this.findOneOrFail({ invoiceId });

    if (status === 'success') {
      const monthNum = amount / MONTHLY_MEMBERSHIP_PRICE;
      membership.startDate = new Date();
      membership.endDate = new Date(
        new Date().setMonth(membership.startDate.getMonth() + monthNum),
      );
      membership.status = MembershipStatus.ACTIVE;
      membership.paidAt = new Date();
      membership.updatedAt = new Date();
    } else if (status === 'failure') {
      membership.status = MembershipStatus.CANCELLED;
    }
    await this.updateOne({ id: membership.id }, membership);
  }

  async getMembership(meta: UserMetadata) {
    const user = await this.userService.findOne(meta.userId);
    const membership = await this.findOne({
      user,
      status: MembershipStatus.ACTIVE,
    });
    if (
      membership &&
      membership.status == 'active' &&
      membership.endDate &&
      membership.endDate < new Date()
    ) {
      membership.status = MembershipStatus.EXPIRED;
    }

    if (membership) {
      // eslint-disable-next-line @typescript-eslint/no-unused-vars
      const { invoiceId, paymentUrl, amount, ...membershipData } = membership;
      return membershipData;
    } else {
      return null;
    }
  }
}

// ----- FILE: membership\entity\membership.entity.ts -----

import { Entity, Property, PrimaryKey, ManyToOne, Enum } from '@mikro-orm/core';
import { ApiProperty } from '@nestjs/swagger';
import { User } from '../../users/entities/user.entity';
import { MembershipStatus } from '../enum/membership-status.enum';
import { BasicEntity } from '../../common/basic-entity';

@Entity()
export class Membership extends BasicEntity {
  @PrimaryKey({ autoincrement: true })
  @ApiProperty()
```

```
  id: number;

  @ManyToOne(() => User)
  @ApiProperty({ type: () => User })
  user: User;

  @Property({ type: 'datetime', nullable: true })
  @ApiProperty()
  startDate?: Date;

  @Property({ type: 'datetime', nullable: true })
  @ApiProperty({ required: false })
  endDate?: Date;

  @Enum(() => MembershipStatus)
  @ApiProperty({ enum: MembershipStatus })
  status: MembershipStatus;

  @Property({ nullable: true })
  @ApiProperty({ required: false, description: 'Monobank invoice ID' })
  invoiceId?: string;

  @Property({ nullable: true })
  @ApiProperty({
    required: false,
    description: 'Link to Monobank invoice page',
  })
  paymentUrl?: string;

  @Property({ type: 'datetime', nullable: true })
  @ApiProperty({
    required: false,
    description: 'Payment confirmation timestamp',
  })
  paidAt?: Date;

  @Property({ nullable: true })
  @ApiProperty({ required: false, description: 'Amount paid in UAH' })
  amount?: number;

  @Property({ type: 'datetime', onCreate: () => new Date() })
  @ApiProperty()
  createdAt: Date;

  @Property({ type: 'datetime', onUpdate: () => new Date() })
  @ApiProperty()
  updatedAt: Date;
}

// ----- FILE: membership\repositories\membership.repository.ts -----

import { EntityRepository } from '@mikro-orm/postgresql';
import { Membership } from '../entity/membership.entity';

export class MembershipRepository extends EntityRepository<Membership> {}

// ----- FILE: monobank\monobank-client.ts -----

import {
  Inject,
  Injectable,
  InternalServerErrorException,
  Logger,
```

```
} from '@nestjs/common';

import axios, { AxiosInstance } from 'axios';
import { MONOBANK_CONFIG, ENDPOINTS } from './constants';
import { IMonobankConfig } from './interface/monobank-config.interface';
import { IMonobankInvoiceResponse } from './interface/monobank-invoice-response.interface';
import { CreateInvoiceDto } from './dto/create-invoice.dto';

@Injectable()
export class MonobankClient {
  private readonly logger = new Logger(MonobankClient.name);
  private readonly axiosInstance: AxiosInstance;

  constructor(
    @Inject(MONOBANK_CONFIG) private readonly config: IMonobankConfig,
  ) {
    this.axiosInstance = axios.create({
      baseURL: this.config.apiUrl,
    });
  }

  private createHeaders(token: string): Record<string, string> {
    return {
      'X-Token': token,
    };
  }

  public async invoiceCreate(
    dto: CreateInvoiceDto,
  ): Promise<IMonobankInvoiceResponse> {
    try {
      this.logger.debug('Trying to create invoice with payload:', dto);

      const response = await this.axiosInstance.post(
        ENDPOINTS.INVOICE_CREATE,
        dto,
        {
          headers: this.createHeaders(dto.token),
        },
      );

      this.logger.debug('Invoice created: ', response.data);

      const { invoiceId, pageUrl } = response.data;

      return { invoiceId, pageUrl };
    } catch (error) {
      this.logger.error('Failed to create invoice:', error);
      throw new InternalServerErrorException(
        `Failed to create invoice: ${error.message}`,
      );
    }
  }

  public async getPublicKey(token: string): Promise<string | undefined> {
    try {
      const response = await this.axiosInstance.get(ENDPOINTS.PUBLIC_KEY, {
        headers: this.createHeaders(token),
      });

      return response.data.key;
    } catch (error) {
      this.logger.error('Failed to get public key:', error);
```

```
      throw new InternalServerErrorException(
        `Failed to get public key: ${error.message}`,
      );
    }
  }
}

// ----- FILE: monobank\monobank-config.provider.ts -----

import { config as envConfig } from '../config';
import { IMonobankConfig } from './interface/monobank-config.interface';
import { MONOBANK_CONFIG } from './constants';

function assertEnvVar(value: string | undefined, name: string): string {
  if (!value) {
    throw new Error(`Missing environment variable: ${name}`);
  }
  return value;
}

export const monobankConfigProvider = {
  provide: MONOBANK_CONFIG,
  useValue: {
    apiUrl: assertEnvVar(envConfig.apiUrl, 'MONOBANK_API_URL'),
    monobankToken: assertEnvVar(envConfig.monobankToken, 'MONOBANK_TOKEN'),
    redirectUrl: assertEnvVar(envConfig.redirectUrl, 'REDIRECT_URL'),
    webhookUrl: assertEnvVar(envConfig.webhookUrl, 'WEBHOOK_URL'),
  } satisfies IMonobankConfig,
};

// ----- FILE: monobank\monobank.module.ts -----

import { Module } from '@nestjs/common';
import { MonobankClient } from './monobank-client';
import { monobankConfigProvider } from './monobank-config.provider';

@Module({
  providers: [MonobankClient, monobankConfigProvider],
  exports: [MonobankClient, monobankConfigProvider],
})
export class MonobankModule {}

// ----- FILE: monobank\guards\monobank-webhook.guard.ts -----

import {
  CanActivate,
  ExecutionContext,
  Inject,
  UnauthorizedException,
} from '@nestjs/common';
import { MonobankClient } from '../monobank-client';
import { MONOBANK_CONFIG } from '../constants';
import { IMonobankConfig } from '../interface/monobank-config.interface';
import * as crypto from 'crypto';

export class MonobankWebhookGuard implements CanActivate {
  constructor(
    @Inject(MonobankClient)
    private readonly monobankClient: MonobankClient,
    @Inject(MONOBANK_CONFIG) private readonly config: IMonobankConfig,
  ) {}

  async canActivate(context: ExecutionContext): Promise<boolean> {
```

```
    const request = context.switchToHttp().getRequest();

    const publicKeyBase64 = await this.monobankClient.getPublicKey(
      this.config.monobankToken,
    );
    if (!publicKeyBase64) return false;

    const signatureBase64 = request.headers['x-sign'];

    const signatureBuf = Buffer.from(signatureBase64, 'base64');
    const publicKeyBuf = Buffer.from(publicKeyBase64, 'base64');
    const verify = crypto.createVerify('sha256');

    verify.write(JSON.stringify(request.body));
    verify.end();
    const result = verify.verify(publicKeyBuf, signatureBuf);
    if (!result) throw new UnauthorizedException('Invalid signature');
    return result;
  }
}

// ----- FILE: notifications\notifications.gateway.ts -----

import { WebSocketGateway, WebSocketServer } from '@nestjs/websockets';
import { Server } from 'socket.io';

@WebSocketGateway({
  cors: {
    origin: '*',
  },
})
export class NotificationsGateway {
  @WebSocketServer()
  server: Server;

  sendReminder(userId: string, message: string) {
    this.server.to(userId).emit('reminder', { message });
  }
}

// ----- FILE: notifications\notifications.module.ts -----

import { Module } from '@nestjs/common';
import { NotificationsService } from './notifications.service';
import { NotificationsGateway } from './notifications.gateway';
import { ScheduleModule } from '@nestjs/schedule';
import { TrainingsModule } from '../trainings/trainings.module';
import { MikroOrmModule } from '@mikro-orm/nestjs';
import { Training } from '../trainings/entities/training.entity';
import { GroupsModule } from '../groups/groups.module';

@Module({
  imports: [
    ScheduleModule.forRoot(),
    TrainingsModule,
    MikroOrmModule.forFeature([Training]),
    GroupsModule,
  ],
  providers: [NotificationsService, NotificationsGateway],
  exports: [NotificationsService],
})
export class NotificationsModule {}
```

```
// ----- FILE: notifications\notifications.service.ts -----

import { Injectable } from '@nestjs/common';
import { NotificationsGateway } from './notifications.gateway';
import { Cron, CronExpression } from '@nestjs/schedule';
import { TrainingsService } from '../trainings/trainings.service';
import { FindTrainingArgs } from '../trainings/args/find-training.args';
import { TrainingStatus } from '../trainings/enums/training-status';
import { TrainingType } from '../trainings/enums/training-type.enum';
import { GroupsService } from '../groups/groups.service';

@Injectable()
export class NotificationsService {
  constructor(
    private readonly notificationsGateway: NotificationsGateway,
    private readonly trainingsService: TrainingsService,
    private readonly groupsService: GroupsService,
  ) {}

  @Cron(CronExpression.EVERY_30_MINUTES)
  async checkUpcomingTrainings() {
    const args = new FindTrainingArgs();

    args.status = TrainingStatus.FUTURE;

    const upcomingTrainings = await this.trainingsService.complexSearch(args);

    for (const training of upcomingTrainings) {
      const timeUntilTraining = training.date.getTime() - Date.now();
      const threeHoursInMs = 3 * 60 * 60 * 1000;

      if (timeUntilTraining <= threeHoursInMs && timeUntilTraining > 0) {
        const message = `Reminder: training starts at ${training.date.toLocaleTimeString()}`;

        if (training.trainingType === TrainingType.INDIVIDUAL) {
          if (training.trainee?.id) {
          this.notificationsGateway.sendReminder(
              training.trainee.id.toString(),
              message,
            );
          }
        } else if (training.trainingType === TrainingType.GROUP) {
          if (training.traineeGroup) {
            const groupMembers = await this.groupsService.findAllMembers(
              training.traineeGroup,
            );
            for (const member of groupMembers) {
              if (member?.id) {
                this.notificationsGateway.sendReminder(
                  member.id.toString(),
                  message,
                );
              }
            }
          }
        }
      }
    }
  }
}

// ----- FILE: stats\stats.controller.ts -----
```

```
import {
  Body,
  Controller,
  Get,
  Param,
  ParseIntPipe,
  Post,
  UseGuards,
} from '@nestjs/common';
import { AuthGuard } from '../auth/guards/auth.guard';
import { ApiBearerAuth, ApiOkResponse, ApiTags } from '@nestjs/swagger';
import { StatsReportService } from './stats.service';
import { CreateStatsReportDto } from './dto/create-stats-report.dto';
import { UserMetadata } from '../auth/types/user-metadata.type';
import { UserMeta } from '../auth/decorator/user-meta.decorator';
import { GetGeneralStatsDto } from './dto/get-general-stats.dto';
import { GetTrainerStatsDto } from './dto/get-trainer-stats.dto';
import { RolesGuard } from '../auth/guards/roles.guard';
import { RequiredRole } from '../auth/decorator/required-role.decorator';
import { UserRole } from '../users/enums/user-role.enum';

@Controller('stats')
@UseGuards(AuthGuard)
@ApiTags('stats')
export class StatsController {
  constructor(private readonly statsReportService: StatsReportService) {}

  @Post('attendanceReport')
  @ApiBearerAuth()
  @ApiOkResponse({
    description: 'Returns attendance report report',
  })
  createAttendanceReport(
    @UserMeta() meta: UserMetadata,
    @Body() dto: CreateStatsReportDto,
  ) {
    return this.statsReportService.createAttendanceReport(dto, meta);
  }

  @Post('general')
  @ApiBearerAuth()
  @UseGuards(RolesGuard)
  @RequiredRole(UserRole.ADMIN)
  @ApiOkResponse({
    description: 'Returns general statistics for the specified period',
  })
  getGeneralStats(
    @Body() dto: GetGeneralStatsDto,
    @UserMeta() meta: UserMetadata,
  ) {
    return this.statsReportService.getGeneralStats(dto, meta);
  }

  @Post('trainerStats')
  @ApiBearerAuth()
  @UseGuards(RolesGuard)
  @RequiredRole(UserRole.ADMIN || UserRole.TRAINER)
  @ApiOkResponse({
    description: 'Returns trainer statistics for the specified period',
  })
  getTrainerStats(
    @Body() dto: GetTrainerStatsDto,
    @UserMeta() meta: UserMetadata,
```

```
  ) {
    return this.statsReportService.getTrainerStats(dto, meta);
  }

  @Get('report:id')
  @ApiBearerAuth()
  @UseGuards(RolesGuard)
  @ApiOkResponse({
    description: 'Returns report from database',
  })
  getReport(
    @Param('id', ParseIntPipe) id: number,
    @UserMeta() meta: UserMetadata,
  ) {
    return this.statsReportService.getReport(id, meta);
  }
}

// ----- FILE: stats\stats.module.ts -----

import { Module } from '@nestjs/common';
import { StatsController } from './stats.controller';
import { StatsReportService } from './stats.service';
import { MikroOrmModule } from '@mikro-orm/nestjs';
import { CacheModule } from '../cache/cache.module';
import { StatsReport } from './entities/stats-report.entity';
import { UsersModule } from '../users/users.module';
import { TrainingsModule } from '../trainings/trainings.module';
import { ExerciseSetModule } from '../exercise-set/exercise-set.module';
import { ExerciseModule } from '../exercise/exercise.module';
import { GroupsModule } from '../groups/groups.module';

@Module({
  imports: [
    MikroOrmModule.forFeature([StatsReport]),
    CacheModule.forRootFromConfig(),
    UsersModule,
    TrainingsModule,
    ExerciseSetModule,
    ExerciseModule,
    GroupsModule,
  ],
  controllers: [StatsController],
  providers: [StatsReportService],
  exports: [StatsReportService],
})
export class StatsModule {}

// ----- FILE: stats\stats.service.ts -----

import { BadRequestException, Injectable } from '@nestjs/common';
import { StatsReport } from './entities/stats-report.entity';
import { BasicCrudService } from '../common/basic-crud.service';
import { CacheService } from '../cache/cache.service';
import { EntityManager } from '@mikro-orm/core';
import { StatsReportRepository } from './repositories/stats-report.repository';
import { User } from '../users/entities/user.entity';
import { CreateStatsReportDto } from './dto/create-stats-report.dto';
import { UsersService } from '../users/users.service';
import { TrainingsService } from '../trainings/trainings.service';
import { TrainingStatus } from '../trainings/enums/training-status';
import { FindTrainingArgs } from '../trainings/args/find-training.args';
import { ExerciseSetService } from '../exercise-set/exercise-set.service';
```

```
import { ExerciseService } from '../exercise/exercise.service';
import { GetGeneralStatsDto } from './dto/get-general-stats.dto';
import { GetTrainerStatsDto } from './dto/get-trainer-stats.dto';
import { UserMetadata } from '../auth/types/user-metadata.type';
import { Training } from '../trainings/entities/training.entity';
import { GroupsService } from '../groups/groups.service';
import { UserRole } from '../users/enums/user-role.enum';

@Injectable()
export class StatsReportService extends BasicCrudService<StatsReport> {
 constructor(
  protected readonly cacheService: CacheService,
  protected readonly statsReportRepository: StatsReportRepository,
  protected readonly entityManager: EntityManager,
  protected readonly usersService: UsersService,
  protected readonly trainingsService: TrainingsService,
  protected readonly exerciseSetService: ExerciseSetService,
  protected readonly exerciseService: ExerciseService,
  protected readonly groupsService: GroupsService,
 ) {
  super(StatsReport, statsReportRepository, cacheService, entityManager);
 }

 async create(dto: CreateStatsReportDto, madeBy: User, data: any) {
  return this.createOne({
   ...dto,
   madeBy,
   data,
   createdAt: new Date(),
  });
 }

 async getReport(id: number, meta: UserMetadata): Promise<StatsReport> {
  const report = await this.findOneOrFail(id);
  const requester = await this.usersService.findOneOrFail(meta.userId);

  if (meta.userRole !== UserRole.ADMIN && report.madeBy !== requester) {
   throw new BadRequestException();
  }
  return report;
 }

 async createAttendanceReport(dto: CreateStatsReportDto, meta: UserMetadata) {
  if (!dto.madeFor) {
   throw new BadRequestException(
    'You must specify a user to generate report for',
   );
  }

  if (!dto.monthsNum || !dto.startDate || !dto.endDate) {
   throw new BadRequestException(
    'You must specify a period to generate report for',
   );
  }

  const madeFor = await this.usersService.findOne(dto.madeFor);

  if (!madeFor) {
   throw new BadRequestException(`User with id ${meta.userId} not found`);
  }

  if (meta.userRole == 'USER' && dto.madeFor != meta.userId) {
   throw new BadRequestException(
```

```
    'You are not allowed to create this report',
  );
}

if (meta.userRole == 'TRAINER' && madeFor.trainer.id != meta.userId) {
  throw new BadRequestException(
    'You are not allowed to create this report',
  );
}

const madeBy = await this.usersService.findOneOrFail(meta.userId);

const allTrainings: Training[] = [];

const individualArgs = new FindTrainingArgs();
individualArgs.status = TrainingStatus.FINISHED;
individualArgs.trainee = madeFor.id;
const individualTrainings =
  await this.trainingsService.complexSearch(individualArgs);
allTrainings.push(...individualTrainings);

if (madeFor.group) {
  const groupArgs = new FindTrainingArgs();
  groupArgs.status = TrainingStatus.FINISHED;
  groupArgs.traineeGroup = madeFor.group.id;
  const groupTrainings =
    await this.trainingsService.complexSearch(groupArgs);

  allTrainings.push(...groupTrainings);
}

let startDate: Date;
let endDate: Date;

if (dto.startDate && dto.endDate) {
  startDate = new Date(dto.startDate);
  endDate = new Date(dto.endDate);
} else {
  endDate = new Date();
  startDate = new Date();
  startDate.setMonth(endDate.getMonth() - dto.monthsNum);
}

const trainingsInPeriod = allTrainings.filter(
  (training) => training.date >= startDate && training.date <= endDate,
);

const exerciseStats = new Map();

for (const training of trainingsInPeriod) {
  if (training.ExerciseSetId) {
    const exerciseSetIds = await this.exerciseSetService.getExerciseIds(
      training.ExerciseSetId,
    );
    if (exerciseSetIds) {
      for (const exerciseId of exerciseSetIds) {
        const exercise = await this.exerciseService.findOneSafe(exerciseId);
        const currentWeight = exercise.endWeight;

        if (!exerciseStats.has(exercise.name)) {
          exerciseStats.set(exercise.name, {
          minWeight: currentWeight,
          maxWeight: currentWeight,
```

```
            count: 1,
           });
          } else {
           const stats = exerciseStats.get(exercise.name);
           if (currentWeight < stats.minWeight) {
           }
           if (currentWeight > stats.maxWeight) {
            stats.maxWeight = currentWeight;
           }
           stats.count++;
          }
         }
        }
       }
      }

      const totalTrainings = trainingsInPeriod.length;
      const weeksInPeriod = Math.max(
       1,
       Math.ceil(
        (endDate.getTime() - startDate.getTime()) / (7 * 24 * 60 * 60 * 1000),
       ),
      );
      const averageTrainingsPerWeek = totalTrainings / weeksInPeriod;

      const data = {
       totalTrainings,
       averageTrainingsPerWeek,
       period: {
        start: startDate,
        end: endDate,
       },
       breakdown: {
        individualTrainings: individualTrainings.filter(
         (t) => t.date >= startDate && t.date <= endDate,
        ).length,
        groupTrainings: trainingsInPeriod.length - individualTrainings.length,
       },
       exercises: Array.from(exerciseStats.entries()).map(([name, stats]) => ({
        name,
        minWeight: stats.minWeight,
        maxWeight: stats.maxWeight,
       })),
      };

      return this.create(dto, madeBy, data);
     }

     async getGeneralStats(dto: GetGeneralStatsDto, meta: UserMetadata) {
      const madeBy = await this.usersService.findOneOrFail(meta.userId);

      const creationDto: CreateStatsReportDto = new CreateStatsReportDto();

      let startDate: Date;
      let endDate: Date;

      if (dto.monthsNum) {
       endDate = new Date();
       startDate = new Date();
       startDate.setMonth(endDate.getMonth() - dto.monthsNum);
      } else if (dto.startDate && dto.endDate) {
       startDate = new Date(dto.startDate);
       endDate = new Date(dto.endDate);
```

```
  } else {
    endDate = new Date();
    startDate = new Date();
    startDate.setMonth(endDate.getMonth() - 1);
  }

  creationDto.startDate = startDate;
  creationDto.endDate = endDate;

  const trainingArgs = new FindTrainingArgs();
  trainingArgs.status = TrainingStatus.FINISHED;
  const allTrainings =
    await this.trainingsService.complexSearch(trainingArgs);
  const trainingsInPeriod = allTrainings.filter(
    (training) => training.date >= startDate && training.date <= endDate,
  );

  const allUsers = await this.usersService.complexSearch({});

  const newUsers = allUsers.filter(
    (user) =>
      user.createdAt &&
      user.createdAt >= startDate &&
      user.createdAt <= endDate,
  );

  const activeUserIds = new Set(
    trainingsInPeriod
      .map((training) => training.trainee?.id)
      .filter((id): id is number => id !== undefined),
  );
  const inactiveUsers = allUsers.filter(
    (user) => user.id && !activeUserIds.has(user.id),
  );

  const data = {
    period: {
      start: startDate,
      end: endDate,
    },
    totalTrainings: trainingsInPeriod.length,
    newUsers: newUsers.length,
    inactiveUsers: inactiveUsers.length,
    breakdown: {
      individualTrainings: trainingsInPeriod.filter((t) => !t.traineeGroup)
        .length,
      groupTrainings: trainingsInPeriod.filter((t) => t.traineeGroup).length,
    },
  };

  console.log(data);

  return this.create(creationDto, madeBy, data);
}

async getTrainerStats(dto: GetTrainerStatsDto, meta: UserMetadata) {
  if (meta.userRole != 'ADMIN' && meta.userId != dto.madeFor) {
  throw new BadRequestException(
      'You are not allowed to create this report',
    );
  }

  const creationDto: CreateStatsReportDto = new CreateStatsReportDto();
```

```
const madeBy = await this.usersService.findOneOrFail(meta.userId);
creationDto.madeFor = dto.madeFor;

let startDate: Date;
let endDate: Date;

if (dto.monthsNum) {
  endDate = new Date();
  startDate = new Date();
  startDate.setMonth(endDate.getMonth() - dto.monthsNum);
} else if (dto.startDate && dto.endDate) {
  startDate = new Date(dto.startDate);
  endDate = new Date(dto.endDate);
} else {
  endDate = new Date();
  startDate = new Date();
  startDate.setMonth(endDate.getMonth() - 1);
}

creationDto.startDate = startDate;
creationDto.endDate = endDate;

const trainingArgs = new FindTrainingArgs();
trainingArgs.status = TrainingStatus.FINISHED;
trainingArgs.trainer = dto.madeFor;
const allTrainings =
  await this.trainingsService.complexSearch(trainingArgs);
const trainingsInPeriod = allTrainings.filter(
  (training) => training.date >= startDate && training.date <= endDate,
);

const uniqueTrainees = new Set<number>();

for (const training of trainingsInPeriod) {
  if (training.trainee) {
    uniqueTrainees.add(training.trainee.id);
  } else if (training.traineeGroup) {
    const groupMembers = await this.groupsService.findAllMembers(
      training.traineeGroup,
    );
    groupMembers.forEach(
      (member) => member.id && uniqueTrainees.add(member.id),
    );
  }
}

// Get all users from uniqueTrainees
const trainees = await Promise.all(
  Array.from(uniqueTrainees).map((id) => this.usersService.findOne({ id })),
);

// Count new trainees
const newTrainees = trainees.filter(
  (user) =>
    user &&
    user.createdAt &&
    user.createdAt >= startDate &&
    user.createdAt <= endDate,
).length;

const data = {
  period: {
    start: startDate,
```

```
      end: endDate,
    },
    totalTrainings: trainingsInPeriod.length,
    uniqueTrainees: uniqueTrainees.size,
    newTrainees,
    breakdown: {
      individualTrainings: trainingsInPeriod.filter((t) => !t.traineeGroup)
        .length,
      groupTrainings: trainingsInPeriod.filter((t) => t.traineeGroup).length,
    },
  };

  return this.create(creationDto, madeBy, data);
 }
}

// ----- FILE: stats\entities\stats-report.entity.ts -----

import { Entity, ManyToOne, PrimaryKey, Property } from '@mikro-orm/core';
import { StatsReportRepository } from '../repositories/stats-report.repository';
import { BasicEntity } from '../../common/basic-entity';
import { User } from '../../users/entities/user.entity';
import { ApiProperty } from '@nestjs/swagger';

@Entity({ repository: () => StatsReportRepository })
export class StatsReport extends BasicEntity {
@PrimaryKey({ autoincrement: true })
@ApiProperty()
 id: number;

 @ManyToOne({ entity: () => User, inversedBy: 'statsReports', nullable: true })
 @ApiProperty({ type: () => User })
 madeFor?: User;

 @ManyToOne({ entity: () => User, inversedBy: 'statsReports' })
 @ApiProperty({ type: () => User })
 madeBy: User;

 @Property()
 @ApiProperty()
 createdAt: Date;

 @Property({ type: 'jsonb' })
 @ApiProperty()
 data: Record<string, any>;

 @Property({ nullable: true })
 @ApiProperty()
 monthsNum?: number;

 @Property({ nullable: true })
 @ApiProperty()
 startDate?: Date;

 @Property({ nullable: true })
 @ApiProperty()
 endDate?: Date;
}

// ----- FILE: stats\repositories\stats-report.repository.ts -----

import { EntityRepository } from '@mikro-orm/postgresql';
import { StatsReport } from '../entities/stats-report.entity';
```

```
export class StatsReportRepository extends EntityRepository<StatsReport> {}

// ----- FILE: trainings\trainings.controller.ts -----

import {
  Body,
  Controller,
  Get,
  Post,
  Patch,
  Delete,
  Param,
  Query,
  UseGuards,
} from '@nestjs/common';
import { AuthGuard } from '../auth/guards/auth.guard';
import { ApiBearerAuth, ApiOkResponse, ApiTags } from '@nestjs/swagger';
import { TrainingsService } from './trainings.service';
import { RolesGuard } from '../auth/guards/roles.guard';
import { RequiredRole } from '../auth/decorator/required-role.decorator';
import { UserRole } from '../users/enums/user-role.enum';
import { Training } from './entities/training.entity';
import { FindTrainingArgs } from './args/find-training.args';
import { CreateTrainingDto } from './dto/create-training.dto';
import { UpdateTrainingDto } from './dto/update-training.dto';
import { UserMetadata } from '../auth/types/user-metadata.type';
import { UserMeta } from '../auth/decorator/user-meta.decorator';

@Controller('trainings')
@UseGuards(AuthGuard)
@ApiTags('trainings')
export class TrainingsController {
  constructor(private readonly trainingsService: TrainingsService) {}

  @Post()
  @ApiBearerAuth()
  @ApiOkResponse({
    description: 'Creates a new training',
    type: Training,
  })
  create(
    @Body() createTrainingDto: CreateTrainingDto,
    @UserMeta() meta: UserMetadata,
  ) {
    return this.trainingsService.create(createTrainingDto, meta);
  }

  @Get()
  @ApiBearerAuth()
  @UseGuards(RolesGuard)
  @RequiredRole(UserRole.ADMIN)
  @ApiOkResponse({
    description: 'Returns all trainings with filtering options',
    type: Training,
    isArray: true,
  })
  findAll(@Query() args: FindTrainingArgs) {
    return this.trainingsService.complexSearch(args);
  }

  @Get(':id')
  @ApiBearerAuth()
```

```
  @UseGuards(RolesGuard)
  @RequiredRole(UserRole.ADMIN)
  @ApiOkResponse({
    description: 'Returns a training by ID',
    type: Training,
  })
  findOne(@Param('id') id: string) {
    return this.trainingsService.findOne(+id);
  }

  @Patch(':id')
  @ApiBearerAuth()
  @ApiOkResponse({
    description: 'Updates a training',
    type: Training,
  })
  update(
    @Param('id') id: string,
    @Body() updateTrainingDto: UpdateTrainingDto,
    @UserMeta() meta: UserMetadata,
  ) {
    return this.trainingsService.update(+id, updateTrainingDto, meta);
  }

  @Delete(':id')
  @ApiBearerAuth()
  @ApiOkResponse({
  description: 'Deletes a training',
  })
  remove(@Param('id') id: string, @UserMeta() meta: UserMetadata) {
    return this.trainingsService.remove(+id, meta);
  }
}

// ----- FILE: trainings\trainings.module.ts -----

import { Module } from '@nestjs/common';
import { TrainingsController } from './trainings.controller';
import { TrainingsService } from './trainings.service';
import { MikroOrmModule } from '@mikro-orm/nestjs';
import { Training } from './entities/training.entity';
import { CacheModule } from '../cache/cache.module';
import { UsersModule } from '../users/users.module';
import { GroupsModule } from '../groups/groups.module';

@Module({
  imports: [
    MikroOrmModule.forFeature([Training]),
    CacheModule.forRootFromConfig(),
    UsersModule,
    GroupsModule,
  ],
  controllers: [TrainingsController],
  providers: [TrainingsService],
  exports: [TrainingsService],
})
export class TrainingsModule {}

// ----- FILE: trainings\trainings.service.ts -----

import {
  Injectable,
  NotFoundException,
```

```
  BadRequestException,
} from '@nestjs/common';
import { EntityManager } from '@mikro-orm/postgresql';
import { Training } from './entities/training.entity';
import { CreateTrainingDto } from './dto/create-training.dto';
import { UpdateTrainingDto } from './dto/update-training.dto';
import { TrainingType } from './enums/training-type.enum';
import { TrainingStatus } from './enums/training-status';
import { BasicCrudService } from '../common/basic-crud.service';
import { CacheService } from '../cache/cache.service';
import { TrainingRepository } from './repositories/training.repository';
import { plainToInstance } from 'class-transformer';
import { FindTrainingArgs } from './args/find-training.args';
import { UsersService } from '../users/users.service';
import { GroupsService } from '../groups/groups.service';
import { UserRole } from '../users/enums/user-role.enum';
import { UserMetadata } from '../auth/types/user-metadata.type';

@Injectable()
export class TrainingsService extends BasicCrudService<Training> {
  constructor(
    protected readonly cacheService: CacheService,
    protected readonly trainingRepository: TrainingRepository,
    protected readonly entityManager: EntityManager,
    private readonly usersService: UsersService,
    private readonly groupsService: GroupsService,
  ) {
    super(Training, trainingRepository, cacheService, entityManager);
  }

  async create(
    createTrainingDto: CreateTrainingDto,
    meta: UserMetadata,
  ): Promise<Training> {
    const training = new Training();

    if (createTrainingDto.trainer) {
      const user = await this.usersService.findOne({
        id: createTrainingDto.trainer,
      });
      if (user && user.role !== UserRole.TRAINER) {
        throw new BadRequestException(
          `User with ID ${createTrainingDto.trainer} is not a trainer`,
        );
      }
      training.trainer = user;
    }

    if (!createTrainingDto.ExerciseSetId) {
      throw new BadRequestException('Exercise set ID is required');
    }

    if (createTrainingDto.trainingType === TrainingType.INDIVIDUAL) {
      if (!createTrainingDto.trainee) {
        throw new BadRequestException(
          'Trainee ID is required for individual training',
        );
      }
      if (createTrainingDto.traineeGroup) {
        throw new BadRequestException(
          'Trainee group should not be specified for individual training',
        );
      }
```

```
  const trainee = await this.usersService.findOne({
    id: createTrainingDto.trainee,
  });
  if (!trainee) {
    throw new NotFoundException(
      `Trainee with ID ${createTrainingDto.trainee} not found`,
    );
  }
} else if (createTrainingDto.trainingType === TrainingType.GROUP) {
  if (!createTrainingDto.traineeGroup) {
    throw new BadRequestException(
      'Trainee group ID is required for group training',
    );
  }
  if (createTrainingDto.trainee) {
    throw new BadRequestException(
      'Trainee should not be specified for group training',
    );
  }

  const group = await this.groupsService.findOne({
    id: createTrainingDto.traineeGroup,
  });
  if (!group) {
    throw new NotFoundException(
      `Group with ID ${createTrainingDto.traineeGroup} not found`,
    );
  }
}

training.trainingType = createTrainingDto.trainingType;
if (createTrainingDto.trainingType === TrainingType.INDIVIDUAL) {
  const trainee = await this.usersService.findOne({
    id: createTrainingDto.trainee,
  });
  if (!trainee) {
    throw new NotFoundException(
      `Trainee with ID ${createTrainingDto.trainee} not found`,
    );
  }
  training.trainee = trainee;
}
if (createTrainingDto.trainingType === TrainingType.GROUP) {
  const group = await this.groupsService.findOne({
    id: createTrainingDto.traineeGroup,
  });
  if (!group) {
    throw new NotFoundException(
      `Group with ID ${createTrainingDto.traineeGroup} not found`,
    );
  }
  training.traineeGroup = group;
}
training.description = createTrainingDto.description || '';
training.date = createTrainingDto.date;
training.ExerciseSetId = createTrainingDto.ExerciseSetId;
training.status = TrainingStatus.FUTURE;
training.NotifiedAbout = false;

if (
  (meta.userRole == 'USER' && training.trainer !== null) ||
  (meta.userRole == 'USER' && training.trainee.id !== meta.userId)
```

```
  ) {
    throw new BadRequestException();
  }

  await this.entityManager.persistAndFlush(training);
  return training;
}

async complexSearch(args: FindTrainingArgs): Promise<Training[]> {
  const query = this.trainingRepository.qb().select('*');

  if (args.id) {
    query.andWhere({ id: args.id });
  }

  if (args.trainingType) {
    query.andWhere({ trainingType: args.trainingType });
  }

  if (args.status) {
    query.andWhere({ status: args.status });
  }

  if (args.trainer) {
    query.andWhere({ trainer: args.trainer });
  }

  if (args.trainee) {
    query.andWhere({ trainee: args.trainee });
  }

  if (args.traineeGroup) {
    query.andWhere({ traineeGroup: args.traineeGroup });
  }

  if (args.exerciseSetId) {
    query.andWhere({ ExerciseSetId: args.exerciseSetId });
  }

  if (args.description) {
    query.andWhere({ description: args.description });
  }

  if (args.date) {
    query.andWhere({ date: args.date });
  }

  if (args.NotifiedAbout) {
    query.andWhere({ NotifiedAbout: args.NotifiedAbout });
  }

  if (args.search) {
    query.andWhere({
      $or: [{ description: { $like: `%${args.search}%` } }],
    });
  }

  const result = await query.execute();
  return result.map((training) => plainToInstance(Training, training));
}

async update(
  id: number,
```

```
    updateTrainingDto: UpdateTrainingDto,
    meta: UserMetadata,
  ): Promise<Training> {
    const training = await this.findOne(id);

    if (!training) {
      throw new NotFoundException(`Training with ID ${id} not found`);
    }

    if (
      (meta.userRole == 'USER' && training.trainee.id !== meta.userId) ||
      (meta.userRole == 'TRAINER' &&
        training.trainer !== null &&
        training.trainer.id !== meta.userId)
    ) {
      throw new BadRequestException();
    }

    this.entityManager.assign(training, updateTrainingDto);
    await this.entityManager.flush();

    return training;
  }

  async remove(id: number, meta: UserMetadata): Promise<void> {
    const training = await this.findOne(id);

    if (!training) {
      throw new NotFoundException(`Training with ID ${id} not found`);
    }

    if (
      (meta.userRole == 'USER' && training.trainee.id !== meta.userId) ||
      (meta.userRole == 'TRAINER' &&
        training.trainer !== null &&
        training.trainer.id !== meta.userId)
    ) {
      throw new BadRequestException();
    }

    await this.entityManager.removeAndFlush(training);
  }
}

// ----- FILE: trainings\entities\training.entity.ts -----

import { BasicEntity } from '../../common/basic-entity';
import {
  Entity,
  PrimaryKey,
  Property,
  Enum,
  ManyToOne,
  ManyToMany,
} from '@mikro-orm/core';
import { TrainingRepository } from '../repositories/training.repository';
import { ApiProperty } from '@nestjs/swagger';
import { TrainingType } from '../enums/training-type.enum';
import { TrainingStatus } from '../enums/training-status';
import { User } from '../../users/entities/user.entity';
import { Group } from '../../groups/entities/group.entity';
import { ExerciseSet } from '../../exercise-set/entities/exercise-set.entity';
```

```
@Entity({ repository: () => TrainingRepository })
export class Training extends BasicEntity {
@PrimaryKey({ autoincrement: true })
@ApiProperty()
  id: number;

  @ManyToOne(() => User, { nullable: true })
  @ApiProperty({ type: () => User })
  trainer: null | User;

  @Enum(() => TrainingType)
  @ApiProperty({ enum: TrainingType })
  trainingType: TrainingType;

  @ManyToOne(() => User, { nullable: true })
  @ApiProperty({ type: () => User })
  trainee: User;

  @ManyToOne(() => Group, { nullable: true })
  @ApiProperty({ type: () => Group })
  traineeGroup: Group;

  @ManyToMany(() => User)
  @ApiProperty({ type: () => [User] })
  absentUsers: User[];

  @Property({ nullable: true })
  @ApiProperty()
  description: string;

  @Property()
  @ApiProperty()
  date: Date;

  @Enum(() => TrainingStatus)
  @ApiProperty({ enum: TrainingStatus })
  status: TrainingStatus;

  @Property()
  @ApiProperty()
  NotifiedAbout: boolean;

  @ManyToOne({ entity: () => ExerciseSet })
  @ApiProperty({ type: () => ExerciseSet })
  ExerciseSetId: number;
}

// ----- FILE: trainings\enums\training-status.ts -----

export enum TrainingStatus {
  FINISHED = 'FINISHED',
  FUTURE = 'FUTURE',
  CANCELLED = 'CANCELLED',
}

// ----- FILE: trainings\repositories\training.repository.ts -----

import { EntityRepository } from '@mikro-orm/postgresql';
import { Training } from '../entities/training.entity';

export class TrainingRepository extends EntityRepository<Training> {}

// ----- FILE: users\users.controller.ts -----
```

```
import { Body, Controller, Get, Patch, Query, UseGuards } from '@nestjs/common';
import { AuthGuard } from '../auth/guards/auth.guard';
import { ApiBearerAuth, ApiOkResponse, ApiTags } from '@nestjs/swagger';
import { UsersService } from './users.service';
import { RolesGuard } from '../auth/guards/roles.guard';
import { RequiredRole } from '../auth/decorator/required-role.decorator';
import { User } from './entities/user.entity';
import { FindUserArgs } from './args/find-user.args';
import { UserRole } from './enums/user-role.enum';
import { UserMeta } from '../auth/decorator/user-meta.decorator';
import { UserMetadata } from '../auth/types/user-metadata.type';
import { UpdateUserDto } from './dto/update-user.dto';

@Controller('users')
@UseGuards(AuthGuard)
@ApiTags('users')
export class UsersController {
  constructor(private readonly usersService: UsersService) {}

  @Get()
  @UseGuards(RolesGuard)
  @RequiredRole(UserRole.ADMIN)
  @ApiBearerAuth()
  @ApiOkResponse({
    description: 'Returns all users',
    type: User,
    isArray: true,
  })
  findAll(@Query() args: FindUserArgs) {
    return this.usersService.complexSearch(args);
  }

  @Get('me')
  @ApiBearerAuth()
  @ApiOkResponse({
    description: 'Returns the current user',
    type: User,
  })
  getMe(@UserMeta() meta: UserMetadata) {
    return this.usersService.findOneByIdSafe(meta.userId);
  }

  @Get('/me/clients')
  @UseGuards(RolesGuard)
  @RequiredRole(UserRole.TRAINER)
  @ApiBearerAuth()
  @ApiOkResponse({
    description: 'Returns all clients',
    type: User,
    isArray: true,
  })
  findAllClients(@UserMeta() meta: UserMetadata) {
    return this.usersService.complexSearch({ trainer_id: meta.userId });
  }

  @Patch()
  @UseGuards(RolesGuard)
  @RequiredRole(UserRole.ADMIN)
  @ApiBearerAuth()
  @ApiOkResponse({
    description: 'Updates a user',
  })
```

```
  update(@Body() dto: UpdateUserDto) {
    return this.usersService.update(dto);
  }
}

// ----- FILE: users\users.module.ts -----

import { Module, forwardRef } from '@nestjs/common';
import { UsersService } from './users.service';
import { UsersController } from './users.controller';
import { MikroOrmModule } from '@mikro-orm/nestjs';
import { User } from './entities/user.entity';
import { CacheModule } from '../cache/cache.module';
import { AuthModule } from '../auth/auth.module';
import { GroupsModule } from '../groups/groups.module';

@Module({
  imports: [
    MikroOrmModule.forFeature([User]),
    CacheModule.forRootFromConfig(),
    forwardRef(() => AuthModule),
    forwardRef(() => GroupsModule),
  ],
  controllers: [UsersController],
  providers: [UsersService],
  exports: [UsersService],
})
export class UsersModule {}

// ----- FILE: users\users.service.ts -----

import {
  Injectable,
  NotFoundException,
  BadRequestException,
} from '@nestjs/common';
import { BasicCrudService } from '../common/basic-crud.service';
import { User } from './entities/user.entity';
import { CacheService } from '../cache/cache.service';
import { UserRepository } from './repositories/user.repository';
import { EntityManager, FilterQuery } from '@mikro-orm/core';
import { plainToInstance } from 'class-transformer';
import { FindUserArgs } from './args/find-user.args';
import { UpdateUserDto } from './dto/update-user.dto';
import { GroupsService } from '../groups/groups.service';
import { UserRole } from './enums/user-role.enum';

@Injectable()
export class UsersService extends BasicCrudService<User> {
  constructor(
    protected readonly cacheService: CacheService,
    protected readonly userRepository: UserRepository,
    protected readonly entityManager: EntityManager,
    private readonly groupsService: GroupsService,
  ) {
    super(User, userRepository, cacheService, entityManager);
  }

  async findOneByIdSafe(id: number): Promise<Partial<User>> {
    const result = await this.findOne({ id });

    if (!result) {
      throw new NotFoundException(`User with id ${id} not found`);
```

```
  }

  return result.toSafeEntity();
}

async complexSearch(args: FindUserArgs): Promise<Partial<User>[]> {
  const query = this.userRepository.qb().select('*');

  if (args.id) {
    query.andWhere({ id: args.id });
  }

  if (args.email) {
    query.andWhere({ email: args.email });
  }

  if (args.fullName) {
    query.andWhere({ fullName: args.fullName });
  }

  if (args.role) {
    query.andWhere({ role: args.role });
  }

  if (args.trainer_id) {
    query.andWhere({ trainer_id: args.trainer_id });
  }

  if (args.search) {
    query.andWhere({
      $or: [
        { email: { $like: `%${args.search}%` } },
        { fullName: { $like: `%${args.search}%` } },
      ],
    });
  }

  const result = await query.execute();

  return result.map((user) => plainToInstance(User, user).toSafeEntity());
}

async update(dto: UpdateUserDto) {
  const filter: FilterQuery<User> = { id: dto.id };
  const updateData: Partial<User> = {};

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

  if (dto.email) {
    updateData.email = dto.email;
  }

  if (dto.role) {
    updateData.role = dto.role as UserRole;
  }

  if (dto.trainerId) {
    const trainer = await this.findOne({ id: dto.trainerId });
    if (!trainer) {
      throw new NotFoundException('Trainer not found');
```

```
      }
      if (trainer.role !== UserRole.TRAINER) {
        throw new BadRequestException('Specified user is not a trainer');
      }
      updateData.trainer = trainer;
    }

    if (dto.group_id) {
      const group = await this.groupsService.findOne({ id: dto.group_id });
      if (!group) {
        throw new NotFoundException('Group not found');
      }
      updateData.group = group;
    }

    if (dto.name) {
      updateData.fullName = dto.name;
    }

    return this.updateOne(filter, updateData);
  }
}

// ----- FILE: users\entities\user.entity.ts -----

import {
  Entity,
  Enum,
  ManyToOne,
  OneToMany,
  PrimaryKey,
  Property,
  ManyToMany,
} from '@mikro-orm/core';
import { UserRole } from '../enums/user-role.enum';
import { BasicEntity } from '../../common/basic-entity';
import { ApiProperty } from '@nestjs/swagger';
import { UserRepository } from '../repositories/user.repository';
import { Group } from '../../groups/entities/group.entity';
import { StatsReport } from '../../stats/entities/stats-report.entity';
import { Training } from '../../trainings/entities/training.entity';

@Entity({ repository: () => UserRepository })
export class User extends BasicEntity {
@PrimaryKey({ autoincrement: true })
@ApiProperty()
  id: number;

  @Property()
  @ApiProperty()
  fullName: string;

  @Property({ unique: true })
  @ApiProperty()
  email: string;

  @Enum(() => UserRole)
  @ApiProperty({
    enum: UserRole,
  })
  role: UserRole;

  @Property({ hidden: true })
```

```
  passwordHash: string;

  @Property({ hidden: true })
  passwordSalt: string;

  @Property({ onCreate: () => new Date() })
  @ApiProperty()
  createdAt: Date;

  @ManyToOne(() => User, { nullable: true })
  @ApiProperty({ type: () => User })
  trainer: User;

  @ManyToOne(() => Group, { nullable: true })
  @ApiProperty({ type: () => Group })
  group: Group;

  @OneToMany(() => StatsReport, (statsReport) => statsReport.madeBy)
  @ApiProperty({ type: () => [StatsReport] })
  statsReports: StatsReport[];

  @ManyToMany(() => Training, (training) => training.absentUsers, {
    mappedBy: 'absentUsers',
  })
  @ApiProperty({ type: () => [Training] })
  absentOnTrainings: Training[];
}

// ----- FILE: users\repositories\user.repository.ts -----

import { EntityRepository } from '@mikro-orm/postgresql';
import { User } from '../entities/user.entity';

export class UserRepository extends EntityRepository<User> {}
```