ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Застосунок для комплексного моніторингу та аналізу показників фізичної активності користувача»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Дмитро ЛИСЕНКО
(підпис)

Виконав: здобувач вищої освіти групи ПД-43

Дмитро ЛИСЕНКО

Керівник: Богдан КАЛИНЮК
асистент

Рецензент: _____

Київ 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____ Ірина ЗАМРІЙ
« ____ » _____ 2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Лисенку Дмитру Вікторовичу

1. Тема кваліфікаційної роботи: «Застосунок для комплексного моніторингу та аналізу показників фізичної активності користувача»
керівник кваліфікаційної роботи асистент кафедри ІПЗ Богдан КАЛИНЮК,
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «20» лютого 2026 року № 51.

2. Строк подання кваліфікаційної роботи «29» травня 2026 року.

3. Вихідні дані до кваліфікаційної роботи: матеріали про принципи проектування мікросервісної архітектури, алгоритми розрахунку спортивних тренувальних метрик, стандарти безпеки вебзастосунків, технічна документація фреймворку Spring, та систем управління баз даних,

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

1. Огляд та аналіз предметної галузі, збір даних та порівняння існуючих програмних аналогів для моніторингу тренувальної активності.
2. Визначення функціональних та нефункціональних вимог до розподіленої системи комплексного відстеження показників фізичного стану.

3. Проєктування мікросервісної архітектури застосунку, логічних схем баз даних та механізмів асинхронної взаємодії сервісів.
4. Програмна реалізація модулів управління тренувальним процесом, аналітичного сервісу та користувацького веб-інтерфейсу.

5. Перелік графічного матеріалу: *презентація*

1. Аналіз аналогів.
2. Вимоги до програмного забезпечення.
3. Програмні засоби реалізації.
4. Діаграма способів використання.
5. Модельні діаграми.
6. Архітектура системи.
7. Екранні форми.
8. Апробація результатів дослідження

6. Дата видачі завдання «23» лютого 2026 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	23.02-28.02.2026	
2	Аналіз та дослідження існуючих аналогів	01.03-07.03.2026	
3	Дослідження та огляд і сучасних підходів до розробки спортивних застосунків та визначення вимог	10.03-20.03.2026	
4	Проектування моделей та архітектури для автоматизованої аналізу фізичної активності	22.03-07.04.2026	
5	Реалізація застосунку на основі досліджень	08.04-24.04.2026	
6	Розробка та реалізація тестових сценаріїв	25.04-02.05.2026	
7	Оформлення роботи: вступ, висновки, реферат	03.05-07.05.2026	
8	Розробка демонстраційних матеріалів	08.05-10.05.2026	
9	Попередній захист роботи	11.05-22.05.2026	

Здобувач вищої освіти _____ Дмитро ЛИСЕНКО
(підпис)

Керівник
кваліфікаційної роботи _____ Богдан КАЛИНЮК
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: _58_ стор., __1__ табл., _20_ рис., _21_ джерел.

Мета роботи – спростити моніторинг фізичної активності шляхом створення програмного забезпечення для обліку тренувань, фіксації робочих ваг та відстеження і аналізу прогресу користувача.

Об'єкт дослідження – процес збору, збереження та аналітичної обробки даних щодо фізичної активності користувача під час тренувань.

Предмет дослідження – методи, засоби та технології розробки клієнт-серверного програмного забезпечення для моніторингу тренувального процесу на основі Java та мікросервісної архітектури.

Короткий зміст роботи: у роботі проаналізовано предметну галузь для автоматизованого розрахунку силових показників користувачів. Проаналізовано існуючі програмні засоби для моніторингу тренувальної активності: Hevy, Strong, Jefit та FitNotes. Розроблено мікросервісну архітектуру та реалізовано ключові функціональні можливості: безпечну аутентифікацію користувачів, ведення щоденника тренувань, управління персональними шаблонами та асинхронну обробку аналітики прогресу через Kafka. Проведено інтеграційне та функціональне тестування сервісів. В роботі використано мову Java та фреймворк Spring для серверної частини, React для створення користувацького інтерфейсу, а також СУБД PostgreSQL та MongoDB для збереження даних.

Сферою використання застосунку є цифровізація тренувального щоденника та автоматизація персонального моніторингу результатів фізичної активності.

КЛЮЧОВІ СЛОВА: JAVA, SPRING, KAFKA, POSTGRESQL, MONGODB, REDIS, JWT, FLYWAY, DOCKER, МІКРОСЕРВІСИ, FRONT-END, BACK-END, JUNIT5, MOCKITO, REACT, TESTCONTAINERS, GYM-TRACKER

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API – Application Programming Interface

CSS – Cascading Style Sheets

UX – User Experience

ACID – Atomicity, Consistency, Isolation, Durability

REST – Representational State Transfer

IDE – Integrated Development Environment

JSON – JavaScript Object Notation

BSON – Binary JSON

CLI – Command Line Interface

JWT – JSON Web Token

HTTP – HyperText Transfer Protocol

ORM – Object-Relational Mapping

CI – Continuous Integration

CD – Continuous Delivery

UUID – Universally Unique Identifier

ВСТУП

Актуальність: ще кілька років тому більшість спортменів вели паперові щоденники або взагалі покладалися на пам'ять. Проблема не в самому підході – а в тому, що без структурованих даних важко помітити зупинку в прогресі або вчасно відкоригувати навантаження. Цифрові інструменти дали змогу це виправити, але ринок пішов у бік соціальних мереж і геймифікації, а не точної аналітики. Однак більшість наявних на ринку рішень мають критичні недоліки: надмірну перевантаженість соціальними функціями, жорсткі фінансові обмеження на базові функції та затримки в обробці великих масивів аналітичних даних. В умовах високоінтенсивних тренувань критично важливою є зручність і швидкість роботи інтерфейсу та надійність зберігання кожної метрики. Саме тому розробка системи на базі мікросервісної архітектури, яка дозволяє асинхронно обробляти статистику без затримок для користувача, є актуальним завданням для сучасного ринку.

Об'єктом дослідження є процес моніторингу, фіксації та аналітичної обробки показників фізичної активності під час силових тренувань користувачів різного рівня підготовки.

Предметом дослідження є програмні засоби та архітектурні підходи (зокрема мікросервісна архітектура та подійно-орієнтована взаємодія сервісів), що застосовуються для побудови системи відстеження тренувального прогресу з використанням сучасного стеку Java та супутніх фреймфорків.

Метою роботи є спрощення та автоматизація моніторингу фізичної активності шляхом створення сучасного програмного забезпечення, що забезпечує швидке логування вправ, надійне збереження даних та миттєвий доступ до статистики та аналітики прогресу.

Методи дослідження: проведено вивчення спортивних методик розрахунку тренувального навантаження – тоннажу, інтенсивності та одноповторного максимуму. Паралельно проводився порівняльний огляд наявних ринкових

рішень: Hevy, Strong, FitNotes і Jefit. Зібрані спостереження лягли в основу вимог до системи. На стадії проєктування застосовувалися UML-діаграми та принципи системного аналізу для побудови мікросервісної структури. Для реалізації серверної частини обрано мову Java 21 та фреймворк Spring Boot 3. Для забезпечення асинхронної взаємодії використано брокер повідомлень Apache Kafka. Збереження даних реалізовано через PostgreSQL та MongoDB. Фронтенд частина розроблена з використанням бібліотеки React та Tailwind CSS. Тестування системи здійснювалося шляхом модульних та інтеграційних перевірок з використанням JUnit та Testcontainers.

Наукова новизна роботи полягає у поєднанні подійно-орієнтованого підходу до процесу логування тренувань, що дозволяє відокремити транзакційні операції запису підходів від затратних обчислень статистики і механізму безпеки на основі JWT-токенів із використанням Redis для ведення динамічних чорних списків, що забезпечує миттєве блокування доступу за використанням stateless архітектури. Такий підхід гарантує високу масштабованість та відмовостійкість системи при зростанні кількості користувачів.

Практична значущість результатів полягає у комбінації двох незалежних механізмів. По-перше, завершення тренування не блокує інтерфейс: Workout Service публікує подію в Kafka і миттєво повертає відповідь клієнту, а складні розрахунки (тоннаж, 1RM, рекорди) виконує Statistics Service асинхронно у фоні. По-друге, проблема відкликання JWT у stateless-архітектурі вирішена через динамічний чорний список у Redis: відкликаний токен стає недійсним негайно, а не через 30 хвилин після виходу користувача.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Специфіка відстеження показників силових тренувань

Силовий спорт має одну неочевидну особливість: результати тут майже повністю залежать від якості даних. На відміну від бігу чи плавання, де датчики автоматично фіксують ключові показники, у залі все потрібно вводити вручну – підхід за підходом. Паперові щоденники з цим справлялися, але погано: важко шукати, легко помилитися, а аналіз займав більше часу, ніж саме тренування. Цифрові інструменти вирішують цю проблему лише частково – більшість із них або переобтяжені зайвим функціоналом, або обмежують базові функції за платним бар'єром. Особливо це помітно у силових видах спорту, де результативність залежить не лише від техніки виконання вправ, а й від точності планування та контролю навантажень протягом тривалого періоду часу. В схожих умовах силові тренування потребують фіксації великої кількості тренувальних івентів та роботи зі складним статистичним функціоналом. Це значно ускладнює побудову зручних та надійних додатків для моніторингу.

Силовий тренінг побудований на простій ідеї: щоб м'язи росли, навантаження потрібно час від часу підвищувати. Без цього організм адаптується до поточного рівня роботи і перестає прогресувати [5]. Зростання може бути різним – додаткові кілограми на грифі, більше підходів за тренування або скорочений відпочинок між ними. Але щоб обирати правильний момент для підвищення, спортсмену потрібні дані. Саме тут і виникає потреба в точному журналі тренувань. Реалізація цього принципу потребує від спортсмена уважного контролю щонайменше трьох основних змінних: ваги, кількості повторень у кожному підході та тривалості відпочинку між підходами. Будь-які систематичні помилки під час фіксації цих параметрів можуть призвести до плато у

результатах, повільне відновлення або навіть до перетренованості та підвищення ризику травм. Таким чином, тренувальні дані в силовому спорті перестають бути просто цифрами в щоденнику – вони стають ключовим інструментом для відстеження прогресу організму та динамікою його змін.

Специфіка даних у сфері силового тренінгу полягає в їхній високій структурованості та ієрархічній організації. Окрема тренувальна сесія не є одноманітним набором повторюваних дій. Вона складається з комплексу вправ, кожна з яких, у свою чергу, поділяється на серії підходів різних типів: розминочні, робочі, підходи до відмови, дроп-сети тощо. Для кожного підходу фіксуються власні параметри: вага на тренажері, кількість повторень, спосіб виконання, а також інтенсивність та загальний об’єм роботи. З точки зору розробки програмного забезпечення, така структура вимагає побудови застосунку, здатного обробляти велику кількість дрібних транзакцій у майже реальному часі, одночасно зберігаючи логічні зв’язки між вправами, підходами, тренуваннями та довгостроковими циклами підготовки.

Додатковою складністю є те, що процес логування даних відбувається в особливих умовах. Під час тренування атлет перебуває у стані підвищеного фізичного та емоційного навантаження, часто – у дефіциті часу між підходами. Він змушений концентруватися на техніці виконання вправ, безпеці, контролі дихання та власних відчуттях. У таких умовах швидкий та зручний інтерфейс, стають критичними. Інтерфейс системи моніторингу повинен бути максимально простим, інтуїтивним та швидкодіючим – введення даних має займати кілька секунд і не відволікати користувача від основного заняття.

Також, специфіка предметної галузі виявляє потребу у високому рівні персоналізації та гнучкості. Кожен спортсмен має унікальну фізіологію та власну структуру тренування – від швидкого full-body тренування до фокусованого на конкретній групі м’язів. Застосунок не може бути жорстко обмежений лише стандартним набором вправ. Вона повинна дозволяти створювати кастомні шаблони, редагувати структуру під час активного тренування та забезпечувати

безпеку персональних даних. Побудова такої системи вимагає використання сучасних підходів до проєктування, які поєднують у собі надійність реляційних баз даних для транзакцій та гнучкість NoSQL-рішень для збереження розгалуженої аналітики.

1.2 Математичні моделі розрахунку тренувальних показників

Перш ніж реалізовувати сервіс аналітики, потрібно чітко визначити, що саме він рахуватиме. У силовому тренінгу найбільше говорять два числа: скільки роботи виконано за сесію і наскільки інтенсивно. Перше описує загальний обсяг – умовний «тоннаж». Друге дає уявлення про те, чи наближається спортсмен до своїх реальних можливостей. Саме ці показники надалі використовуються як вхідні дані для побудови графіків прогресу та прогнозування спортивних результатів користувача.

Базовою кількісною метрикою в силових тренуваннях є тренувальний об’єм, який у спеціалізованій літературі та програмному забезпеченні часто називають тоннажем [5]. Під тоннажем розуміють сумарну вагу вправ, піднятих спортсменом за певний проміжок часу – в межах окремого підходу, конкретної вправи або всієї тренувальної сесії. З точки зору алгоритмів, об’єм для однієї вправи обчислюється як сума добутків ваги снаряда та кількості повторень у кожному успішно виконаному підході. Це можна подати у вигляді формули (1.1):

$$V = \sum_{i=1}^{n} \left(w_i \cdot x_i \right), \tag{1.1}$$

де V – сумарний об’єм, w_i – вага обтяження в i-му підході, а x_i – кількість виконаних повторень. Ця метрика є дуже важливою для оцінки загального навантаження на нервову систему та м’язові волокна. У рамках мікросервісної архітектури застосунку відповідний розрахунок краще виконувати асинхронно після надходження події про завершення тренування. Це дає можливість зберігати

агреговані дані в денормалізованому вигляді та забезпечувати швидкий доступ до історії статистики.

Другим важливим показником є інтенсивність тренування, яка відображає рівень навантаження порівняно з максимальною можливістю організму. Оскільки робота з великими вагами на постійній основі є травмонебезпечною, з'являється потреба в об'єктивному способі оцінювання поточного силового потенціалу без прямого тестування максимуму. Для цього в спортивних метриках та аналогічних застосунках використовуються формула розрахунку одноповторного максимуму (1RM) [4]. Одноповторний максимум визначають як теоретичну вагу, яку спортсмен здатен підняти лише один раз із дотриманням правильної техніки виконання вправи.

Однією з найбільш поширених і зручних для реалізації в програмних алгоритмах є формула Еплі (1.2). Вона дає змогу обчислити умовний максимум на основі робочої ваги та кількості повторень у підході, що виконувався майже до відмови[4]. Математична формула має вигляд:

$$1RM = w \cdot (1 + 0.0333 \cdot r), \tag{1.2}$$

де w – робоча вага, а r – кількість виконаних повторень. Застосування цієї формули дозволяє системі автоматично оновлювати персональні рекорди користувача після кожної тренувальної сесії. Завдяки цьому реалізується принцип прогресивного навантаження: спортсмен отримує актуальні дані про свій поточний силовий рівень і може більш точно планувати подальші етапи тренування. Чим більше кількість повторів у підході буде перевищувати 10 разів, тим більше буде похибка у розрахунках.

Використання даних математичних формул у мікросервісну архітектуру вимагає окремого середовища. Усі основні розрахунки доцільно виконувати на стороні сервісу статистики, щоб уникнути змішування між різними програмними сервісами. Подія про завершення тренування, що надсилається через брокер повідомлень, ініціює ланцюх кроків: система спочатку валідує вхідні дані, далі

обчислює об’єм навантаження для кожної м’язової групи, а потім перевіряє, чи були нові персональні рекорди.

1.3 Дослідження існуючих програмних аналогів

Аналіз App Store та Google Play за запитом «workout tracker», одразу показує закономірність: переважна більшість застосунків намагається бути соціальною мережею зі спортивним ухилом. Стрічка активності, лайки під тренуваннями, підписки на атлетів – усе це займає перші екрани. Для атлета, якому потрібно просто зафіксувати підхід між сетами, такий інтерфейс – зайве навантаження, а не допомога.

Ще однією характерною особливістю наявних рішень є те, що переважна їх більшість реалізована у форматі мобільних застосунків. Використання смартфона справді є зручним безпосередньо під час занять у тренажерному залі, веб інтерфейс має всі ті ж функції, що і мобільний варіант, але дозволяє спортсмену використовувати на будь-якому пристрої з інтернетом, з доступов до повного функціоналу.

1.3.1 Аналіз застосунку Hevy

Застосунок Hevy на сьогодні є одним із лідерів ринку завдяки поєднанню функціонального трекера та соціальної мережі для спортсменів. Його популярність значною мірою пояснюється інтеграцією соціальних функцій у застосунок, що дає змогу користувачам не лише фіксувати власні результати, а й активно взаємодіяти з іншими користувачами та спільнотами. До ключових переваг системи можна віднести сучасний і дуже швидкий UI/UX, а також зручний каталог вправ із вбудованими відеоінструкціями [1].

Користувачі мають можливість ділитися тренуваннями з друзями, коментувати їхні досягнення та ставити лайки, що формує виражений ефект гейміфікації й забезпечує додаткову мотивацію до регулярних занять.

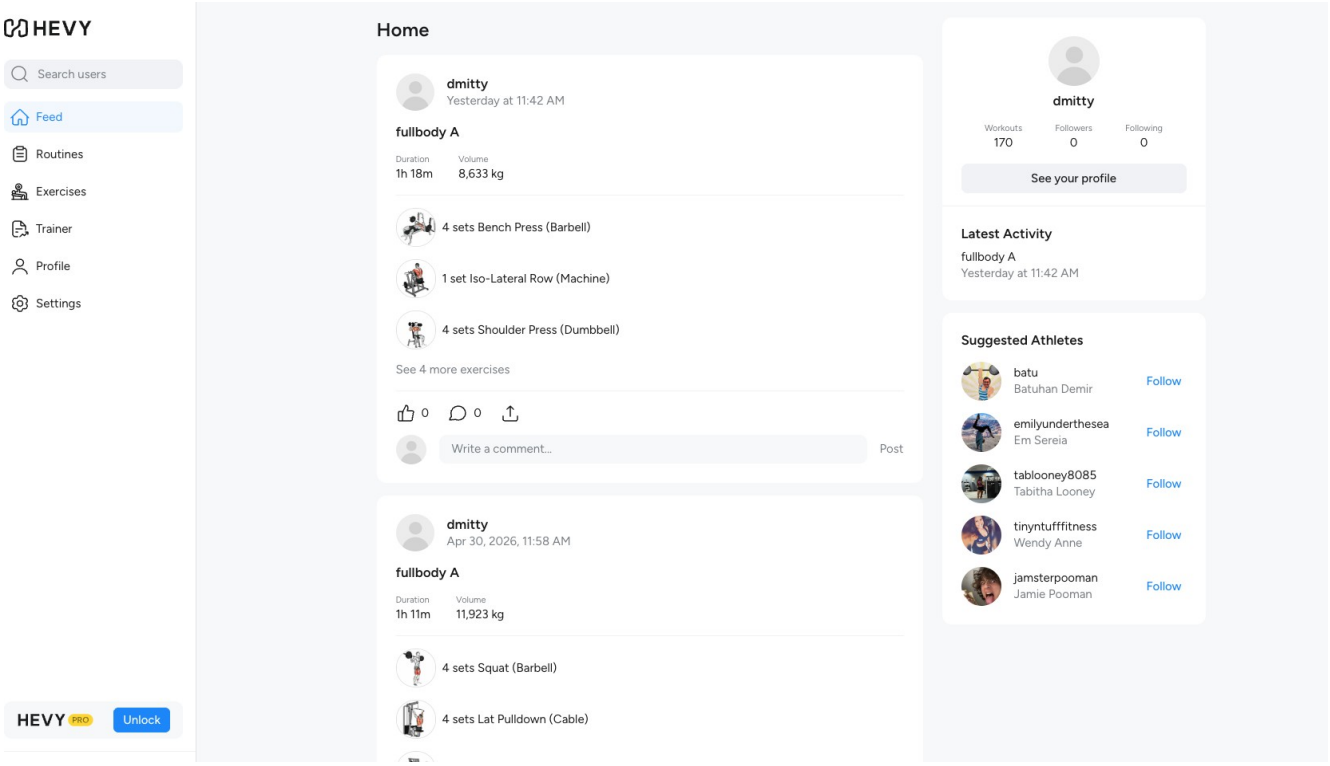


Рис. 1.1 Головна сторінка з соціальною стрічкою

Окремою сильною стороною Hevy, на відміну від багатьох конкурентів, є наявність повноцінного вебінтерфейсу для аналізу статистики на великому екрані. Це дозволяє тренерам і досвідченим спортсменам у з більшою детальністю переглядати графіки прогресу, загальний піднятий тоннаж, відстежувати динаміку зміни одноповторного максимуму, використовуючи персональний комп’ютер.

Водночас, попри очевидні переваги, платформа має низку суттєвих недоліків, які обмежують її ефективність у контексті системного та цілеспрямованого тренування. Насамперед йдеться про жорсткі обмеження безкоштовної версії, наприклад користувач може створити лише три власні шаблони тренувань. Для спортсменів, які займаються за програмами з високою

частотою, це фактично унеможливлює повноцінну роботу із застосунком без оформлення платної підписки.

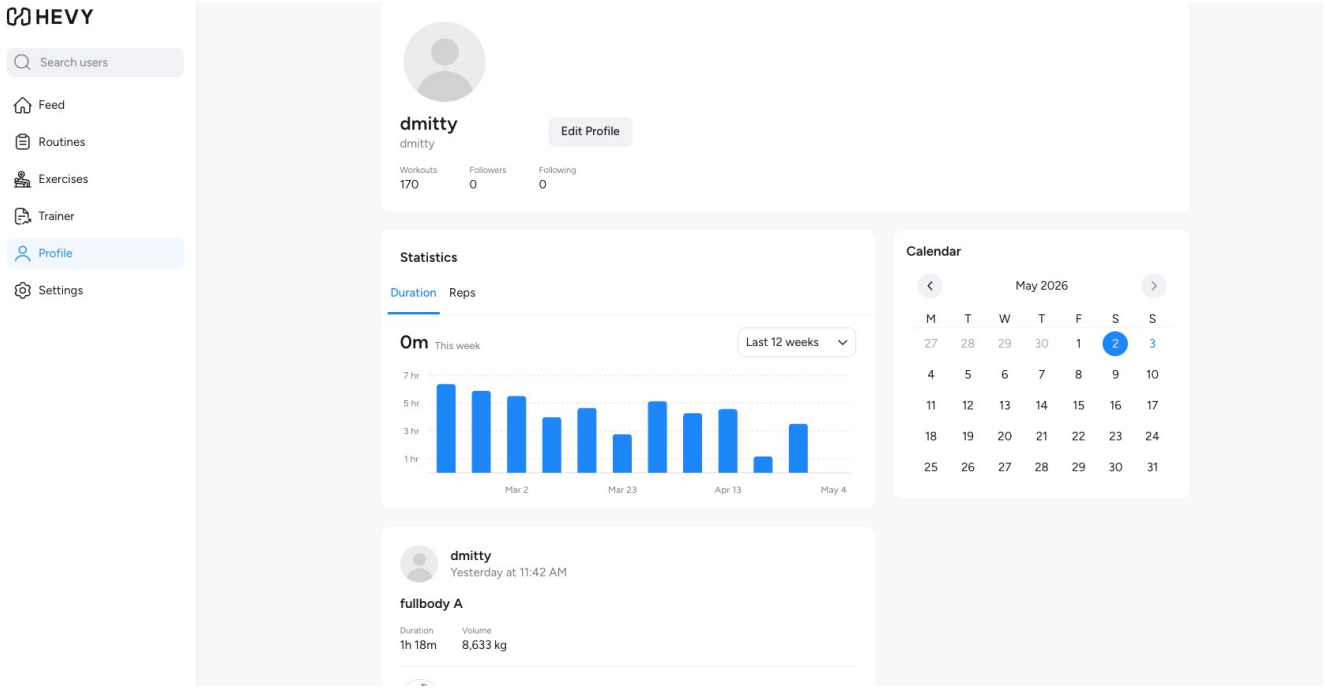


Рис. 1.2 Профіль користувача та аналітика

Окрім того, виражений акцент на соціальній складовій інколи може відволікати від самого процесу тренування, перевантажуючи інтерфейс під час тренування і знижуючи загальну зосередженість користувача.

1.3.2 Аналіз застосунку Strong

Застосунок Strong протягом тривалого часу вважався своєрідним золотим стандартом серед щоденників тренувань завдяки своєму мінімалістичному підходу. Ідея цього програмного продукту суттєво відрізняється від соціально орієнтованих рішень і базується на максимальному спрощенні взаємодії користувача з інтерфейсом у момент тренування.

Ключовою перевагою системи є акцент на швидкості введення даних та наявність спеціалізованих інструментів для пауерліфтингу, як калькулятору млинів і таймера відпочинку[2]. Процес логування побудовано таким чином, щоб

фіксація кількості повторень і ваги вимагала мінімальної кількості натискань на екран. Після збереження підходу таймер автоматично запускається, що дає змогу чітко контролювати час відновлення між робочими серіями.

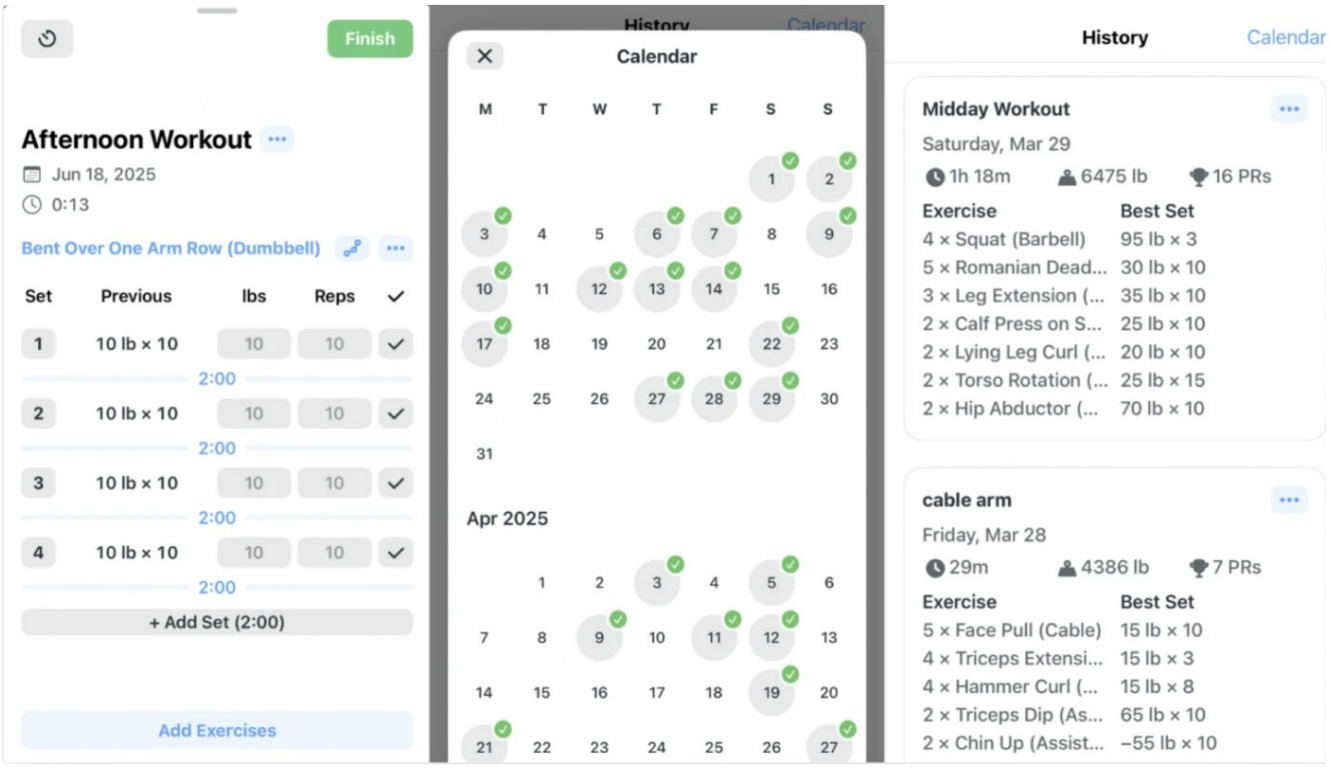


Рис. 1.3 Інтерфейс активного тренування та історії Strong

До технологічних переваг Strong варто віднести підтримку Apple Watch, завдяки чому спортсмени можуть залишати телефон у роздягальні й фіксувати тренування безпосередньо з розумного годинника.

Разом із тим застосунок має кілька критичних недоліків, пов’язаних із технічною підтримкою та подальшим розвитком продукту. Зокрема, Strong практично не отримує суттєвих функціональних оновлень впродовж останнього часу, тоді як вартість підписки залишається на рівні з ринком. Додатковою проблемою є те, що архітектура рішення є виключно мобільною: відсутній вебзастосунок, а архітектура реалізована у вигляді моноліту. У випадку спроби зберегти велике тренування за умов нестабільного інтернет-з’єднання, що є звичайною ситуацією для тренажерних залів, розташованих у підвальних

приміщеннях, можуть виникати відчутні затримки в роботі інтерфейсу. Розроблюваний застосунок має уникнути подібних проблем завдяки використанню асинхронної передачі подій через Kafka.

1.3.3 Аналіз застосунку FitNotes

Застосунок FitNotes займає окрему нішу на ринку і є особливо популярним серед досвідчених атлетів, які цінують простоту та відсутність зайвих функцій. Це рішення можна вважати класичним прикладом локального трекера, що забезпечує максимально швидкий відгук інтерфейсу незалежно від наявності підключення до мережі Інтернет.

До основних переваг FitNotes належать повністю безкоштовний базовий функціонал, відсутність реклами та будь-яких соціальних елементів, а також зручний календар для перегляду історії тренувань. Користувач має змогу створювати власні вправи, аналізувати тренувальний об’єм і при цьому не стикається з підписками чи обмеженнями. Програма використовує легку локальну базу даних, завдяки чому стабільно працює навіть на технічно застарілих пристроях.

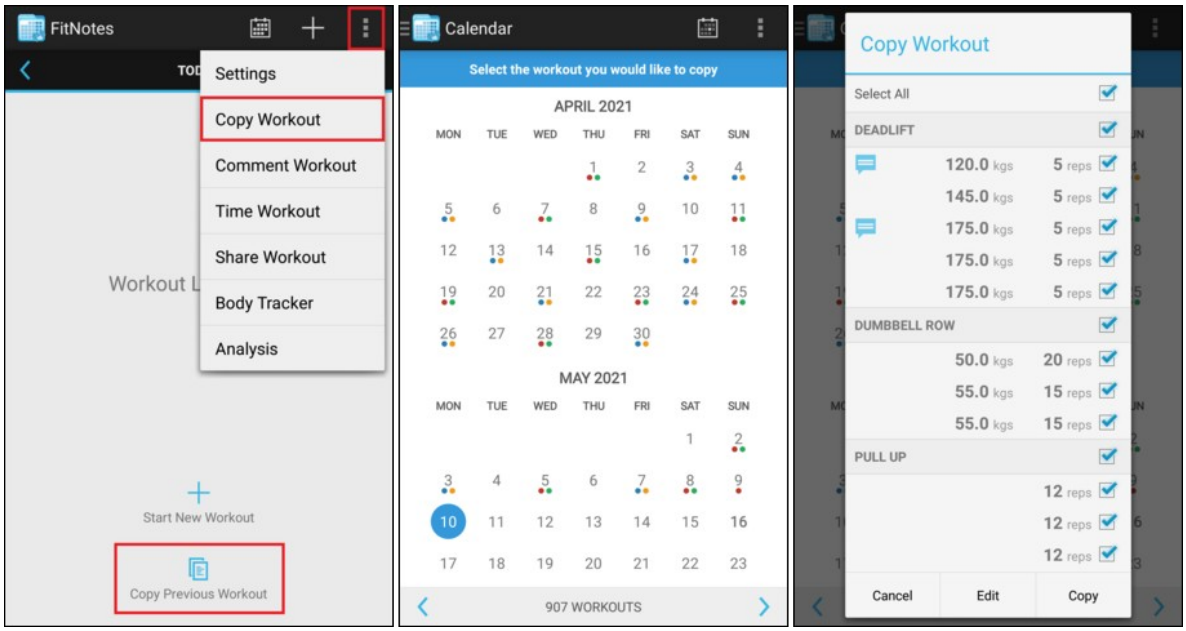


Рис. 1.4 Менеджмент тренувань по календарю FitNotes

Водночас, попри високу швидкість і безкоштовність, локальна архітектура застосунку дає низку обмежень для сучасного користувача. Серед ключових недоліків можна виділити застарілий дизайн інтерфейсу, відсутність версії для браузеру, а також відсутність синхронізації. Для збереження даних при зміні пристрою користувачеві необхідно самостійно налаштовувати резервне копіювання через Google Drive. Подібний підхід і відсутність автоматичної синхронізації роблять FitNotes менш привабливим для спортсмена, який звик до сучасних безшовних хмарних екосистем.

1.3.4 Аналіз застосунку Jefit

Застосунок Jefit є одним із найстаріших і водночас найфункціональніших гравців на ринку фітнес-трекерів. Його ключова перевага полягає в наявності великої вбудованої бібліотеки вправ, доповненої детальними анатомічними схемами м’язів та інструкціями щодо правильного виконання. Завдяки цьому Jefit є потужним інструментом для планування тренувального процесу.

Окрім розширеної бази вправ, застосунок надає користувачам доступ до готових тренувальних програм, а також до деталізованої аналітики розподілу навантаження[3]. Також в наявності підтримка синхронізації, що дає змогу зберігати дані на сервері та переглядати статистику як з мобільного пристрою, так і через повноцінний вебінтерфейс.

Але прагнення розробників об’єднати в одному продукті якомога більше функцій призвело до явних недоліків із UX та UI. Інтерфейс Jefit виглядає трохи перевантаженим і може бути складним для сприйняття, особливо для новачків. Створення власного шаблону тренувань або швидке логування підходу часто вимагає додаткових дій через не зовсім інтуітивне меню, що суперечить принципу використання основних функцій за мінімум кліків.

Безкоштовна версія містить велику кількість нав’язливої банерної та повноекранної реклами, яка може з’являтися як під час роботи з щоденником, так і

в перервах між підходами. Це негативно позначається на досвіді використання та склоняє користувача змінити застосунок по першій можливості.

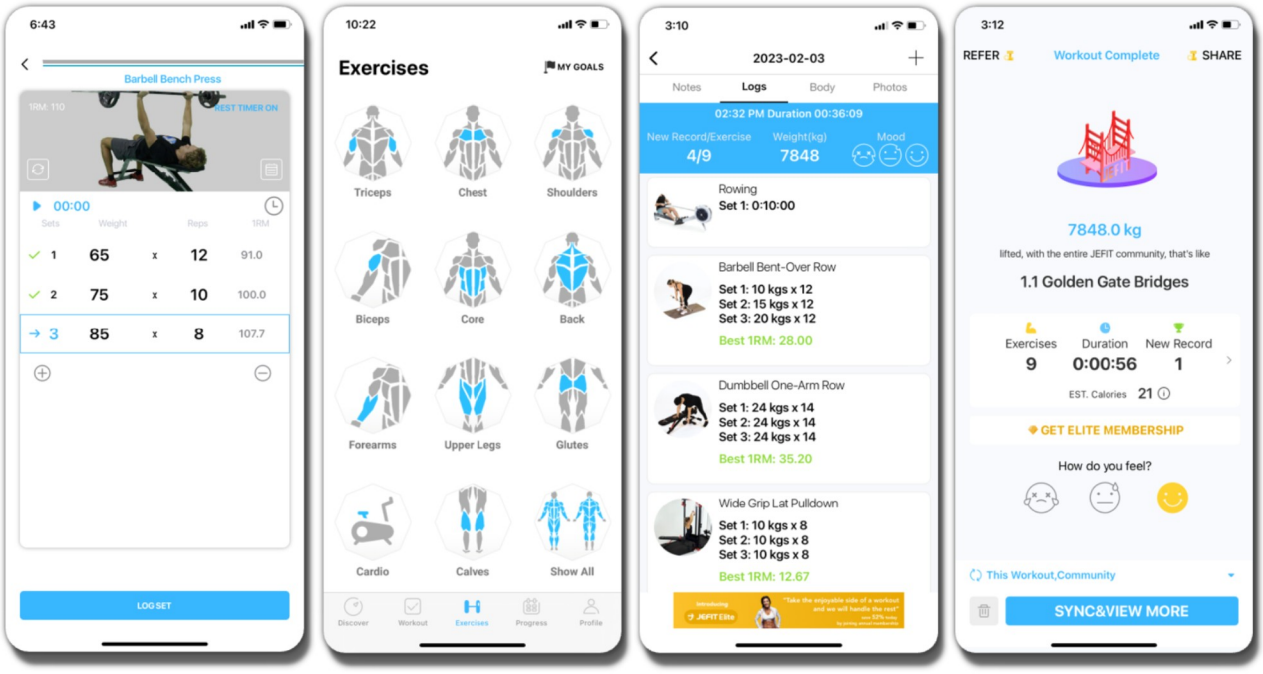


Рис. 1.5 Функціональний інтерфейс JeFit

Таблиця 1.1

Порівняльна таблиця аналогічних програмних засобів

Критерії порівняння	Hevy	Strong	Jefit	FitNotes	Розроблюване ПЗ
Синхронізація та доступність даних	+	+	+	-	+
Сучасний інтерфейс (UX/UI)	+	+	+/-	-	+
Повний безкоштовний функціонал	-	-	+	+	+
Відсутність реклами та соціальних функцій	-	+	-	+	+

Продовження таблиці 1.1

Порівняльна таблиця аналогічних програмних засобів

Критерії порівняння	Hevy	Strong	Jefit	FitNotes	Розроблюване ПЗ
Повноцінна система аналітики	+	-	+	-	+
Асинхронність	+	-	-	-	+

1.4 Визначення проблем та обґрунтування розробки власної системи

Аналіз наявних аналогічних рішень для моніторингу тренувального процесу дає підстави казати, що ринок дуже перенасичений комерційними продуктами. Однак більшість із них не повною мірою задовольняє ані архітектурні рішення, ані вимоги користувачів, які займаються системним силовим тренінгом. На основі порівняння аналогів було виявлено низку критичних проблем, що перешкоджають ефективному використанню цих систем.

Ключові проблеми:

– Штучні обмеження базового функціоналу або paywalls. Більшість сучасних хмарних застосунків суттєво обмежують користувачів у створенні шаблонів тренувань. Для побудови повноцінного тренувального циклу спортсмен фактично змушений оформлювати платну підписку. Це суперечить базовій сценарію використання цифрового щоденника тренувань та знижує доступність інструменту для звичайного кола користувачів.

– Соціальне перевантаження та відволікаючі фактори. Намагання розробників збільшити час перебування користувача в застосунку веде до створення соціальних стрічок, алгоритмів рекомендацій і рекламних блоків. У контексті високоінтенсивного тренування користувач насамперед

потребує інструменту з мінімальними відволікаючими факторами та швидким введенням даних, а не повноцінної соціальної мережі. Надлишок соціальних блоків у інтерфейсі відволікає від виконання підходів і негативно впливає на концентрацію.

– Застарілий інтерфейс окремих програм. Частина представлених на ринку інструментів, попри їхню стабільність, візуально виглядає як продукти минулого десятиліття. Відсутність сучасних UX патернів, недостатня увага до відгуків та дрібних деталей роблять взаємодію з такими системами менш інтуїтивною. Це не відповідає очікуванням звичайних користувачів, які очікують швидкий, зручний та привабливий інтерфейс.

– Архітектурні недоліки моноліту. Використання монолітної архітектури в подібних проєктах створює значні обмеження та труднощі як під час розширення функціональності, так і в процесі масштабування. Висока зв’язаність компонентів призводить до того, що зміни в одному модулі можуть спричиняти непередбачувані помилки в іншому. Окрім того, збільшується час відповіді програми при виконанні складних запитів. Масштабувати таку систему частково неможливо, навіть якщо навантаження зростає лише на сервіс аналітики, доводиться збільшувати весь моноліт, що не є ефективним з огляду на сучасний розглад вартості обладнання.

Ці проблеми обґрунтовують необхідності розробки власного вебзастосунку. Головна ідея проєкту полягає у створенні програми, яка поєднує спортивну аналітику зі швидкодією та безперебійністю локальних застосунків.

Для вирішення проблеми монолітної архітектури та надійної обробки даних застосунок базуватиметься на мікросервісній архітектурі з подійно орієнтованою взаємодією компонентів. Замість монолітного сервера, який синхронно обробляє кожен запит, основна логіка буде розподілена між кількома сервісами:

– Основний тренувальний сервіс відповідатиме виключно за швидке збереження поточних підходів у реляційну базу даних PostgreSQL. Він повертатиме клієнту успішну відповідь одразу після запиту.

– Аналітичний сервіс працюватиме асинхронно. Інформацію про завершення тренування він отримуватиме через брокер повідомлень Kafka. Що дає змогу виконувати складні обчислення у фоновому режимі й зберігати результати в NoSQL базу MongoDB, створену для швидкої побудови графіків та пошук історичних даних.

Вибір вебформату дає можливість використовувати застосунок з будь-якого пристрою, незалежно від операційної системи, за умови доступу до інтернету, що розширює потенційну аудиторію.

Крім того, розробка власної системи дає змогу повністю відмовитися від соціальних функцій та реклами на користь чистої функціональності, а також безкоштовно надати користувачам необмежений доступ до створення кастомних шаблонів вправ і тренувальних програм.

1.5 Визначення функціональних та нефункціональних вимог до застосунку

Функціональні вимоги зосереджені на реалізації повного життєвого циклу взаємодії користувача з тренувальним процесом.

– Система має забезпечувати безпечну реєстрацію, автентифікацію та авторизацію, що гарантують конфіденційність персональних даних.

– Користувач повинен мати повний контроль над власним профілем.

– Базовим елементом архітектури виступає каталог вправ, у якому інформацію груповано за цільовими м’язовими групами. Система має надавати можливість не лише використовувати заздалегідь підготовлені пресети, а й формувати персоналізовані шаблони програм тренувань.

– У процесі виконання тренування застосунок повинен забезпечувати повноцінне логування: фіксацію робочої ваги та кількості повторень під час тренування. Отримані дані одразу передаються в відповідний сервіс і конвертуються в аналітичні показники, такі як загальний тоннаж сесії та розрахунковий одноповторний максимум, які візуалізуються у вигляді графіків прогресу.

Нефункціональні вимоги задають якісну базу роботи застосунку та її надійність.

– Ключовим показником продуктивності є швидкість відгуку: час обробки типового REST-запиту на сервері не повинен перевищувати 200 мс, а повне завантаження сторінки активного тренування має відбуватися протягом 1 секунди.

– З позиції UX інтерфейс має залишатися максимально доступним: логування нового підходу повинно виконуватися не більш ніж у два кліки, а створення нової програми «з нуля» – займати в користувача до 2 хвилин без необхідності звертатися до документації.

– Окремий блок вимог стосується надійності та подальшого розширення системи. Використання мікросервісної архітектури на базі Java та Spring забезпечує високу масштабованість, даючи змогу інтегрувати нові типи сервісів або розширювати конкретний модуль без глобального рефакторингу існуючої бази.

– Робота з даними має спиратися на принципи цілісності ACID, що гарантує збереження кожного логу за будь-яких обставин.

Після визначення всіх функціональних і нефункціональних вимог, і порівняння продукту з аналогами, можна з впевненістю переходити до проєктування і практичної реалізації застосунку.

2 ЗАСОБИ РЕАЛІЗАЦІЇ ПРОГРАМНОГО ЗАСТОСУНКУ

2.1 Вибір середовища розробки та системи контролю версій

Основним інструментом для реалізації серверної логіки мовою Java обрано середовище розробки IntelliJ IDEA. Вибір цього IDE зумовлений його повноцінною інтеграцією з екосистемою Spring Boot і вбудованою підтримкою інструментів для збирання проєкту Maven і Gradle. Завдяки вбудованому аналізу контексту IntelliJ IDEA допомагає автоматизувати значну частину рутинних операцій, наприклад генерацію ідентичного коду, перевірку Bean компонентів та вбудоване управління сервісами для мікросервісної архітектури. Одна з головних переваг цього середовища є якісно вбудований debugger та профайлера, що є критично важливим під час пошуку і виправлення помилок у коді.

Поряд із важким і профільним IDE, для роботи з клієнтською частиною на базі React і Tailwind CSS, а також для швидкого редагування більшості конфігураційних файлів, таких як YAML і JSON, було використано Visual Studio Code. Легка архітектура цього редактора та велика кількість розширень дає можливість швидко і ефективно працювати з файлами, і створений особливо для роботи з фронтенд частиною.

Окремим інструментом, що суттєво підвищив продуктивність розробки, став термінальний інтерфейс. CLI забезпечує швидкий доступ до всіх файлів та інструментів — Neovim, Maven, AWS CLI та Docker.

Значна частина роботи з інфраструктурою велася через термінал із Neovim, Maven та Docker CLI. Такий підхід дає повний доступ до всіх параметрів інструментів – наприклад, команда «docker-compose up --build» разом із конфігурацією docker-compose.yml дозволяла одним кроком підіймати весь стек: PostgreSQL, MongoDB, Redis, Kafka та обидва мікросервіси.

Основою для стабільної розробки та підтримки цілісності коду виступає система контролю версій Git. Вона використовується не лише як сховище коду, а й як інструмент реалізації стратегії паралельної розробки. Для ефективного керування змінами в рамках проєкту GymTracker було розглянуто дві основні стратегії розгалуження, вибір між якими залежить від інтенсивності розробки і філософії команди:

– GitFlow – передбачає чітку послідовність гілок (Main, Develop, Feature, Release), забезпечуючи високу стабільність релізів і добре підходить для великих систем із фіксованим графіком оновлень

– GitHub Flow – базується на Agile філософії, де використовується одна стабільна головна гілка та короткоживучих гілок для окремих функціональностей. Такий підхід сприяє реалізації принципів безперервної доставки CD і дозволяє швидко реагувати на зміну вимог користувачів і впроваджувати новий функціонал.

Оскільки проєкт розроблявся однією людиною, а також термін змін залежив від успіху написання програми, було обрано спрощена версія GitFlow у якій використовувалася основна гілка для готового продукту, гілка розробки як центр міграції всіх нових функцій перед релізом, а також окремі малі тимчасові гілки для нового функціоналу і виправлення багів.

Як хмарну платформу для хостингу репозиторію і управління обрано GitHub. Цей вибір надає передові функції для менеджменту проєкту, інструменти для проведення code review, у межах яких кожен pull request аналізується на відповідність вимогам перед злиттям із головною гілкою. Крім того, екосистема GitHub надає можливість інтегрувати автоматизовані перевірки при використанні GitHub Actions, які запускають тести при кожному пуші. Це буде гарантувати, що нові зміни в одному мікросервісі не порушують логіку роботи інших частинах.

2.2 Серверний стек на базі Java та Spring Boot 3

Java і Spring Boot вважаються стандартом індустрії, але були обрані через конкретні практичні переваги. По-перше, Spring Data JPA суттєво спрощує роботу з PostgreSQL: замість написання сотень однотипних SQL-запитів достатньо описати репозиторій через інтерфейс. По-друге, Spring Security з мінімальною конфігурацією надає фільтри для перевірки JWT – ми просто підключили їх до ланцюжка, а не писали з нуля. Java 21 зі своїми Records і Stream API дозволяє писати лаконічний код для трансформацій даних між шарами.

Java, як основна мова розробки, забезпечує сувору типізацію та передбачуваність поведінки, що є надзвичайно важливим при обробці даних користувачів. Використання більш сучасних версій JDK, зокрема функціональних можливостей Stream API та Records, дозволяє писати більш лаконічний, але виразний код для обробки даних у системі, одночасно зменшуючи кількість однакових конструкцій і мінімізуючи ризик помилок на етапі збирання.

Використання Spring Boot 3 є головним фактором при розробці завдяки концепції «Convention over Configuration» [10]. Фреймворк дає змогу розгорнути готовий до продакшну мікросервіс за лічені хвилини, спираючись на стартові залежності для інтеграції з базами даних, систем безпеки та брокерами повідомлень. У рамках GymTracker це значить, що розробник може оперативно налаштувати і редагувати REST API для клієнтських застосунків, не витрачаючи велику кількість часу на конфігурування серверів додатків чи роботу зі складними XML-файлами для конфігурацій.

Важливою перевагою екосистеми Spring є інтеграція з Hibernate через Spring Data JPA. Це забезпечує безшовну взаємодію з PostgreSQL, де кожна вправа, сет або антропометричний показник пов’язується з відповідною сутністю у коді [9]. Що дає можливість реалізовувати складні запити до бази даних на основі рівня абстракції, що значно спрощує використання баз даних у системі. А

вбудовані механізми валідації даних гарантують, що в систему не потраплятимуть некоректні значення.

Переходячи до безпеки, Spring Boot забезпечує високий рівень безпеки з коробки. Завдяки модулю Spring Security реалізується модель обмеження доступу, за якої користувачі бачать лише власні дані та шаблони тренувань, а службові операції залишаються ізольованими.

2.3 Технології мікросервісної взаємодії та брокер повідомлень Kafka

Перехід від монолітної структури до мікросервісної архітектури в проєкті вимагає особливого ставлення до того, як саме компоненти системи спілкуються між собою. Якщо в класичних вебзастосунках і монолітах домінує синхронна взаємодія через REST, коли один сервіс очікує відповіді від іншого, то для програмного забезпечення з високим навантаженням і важкими розрахунками такий підхід швидко показує свої недоліки. Саме тому основою взаємодії між сервісами обрано подієво орієнтовану архітектуру (Event-Driven Architecture), використовуючи Apache Kafka [14].

Kafka використовується не просто як черга повідомлень, а як розподілений журнал подій з високою пропускною здатністю [6]. У контексті GymTracker її можна умовно порівняти з центральною нервовою системою: будь-яка дія користувача – завершення підходу, оновлення ваги тіла чи зміна програми – створює подію, яка потрапляє до відповідного топіка. Основний сервіс тренувань у цьому випадку лише публікує подію і відразу повертає відповідь клієнту, не чекаючи, поки аналітичний модуль перерахує статистику або підготує повідомлення про новий рекорд.

Чому саме Kafka, а не рішення на кшталт RabbitMQ? У нашому випадку це усвідомлений вибір, який спирається на кілька критично важливих факторів:

– Зберігання подій і можливість повторного відтворення. Kafka зберігає повідомлення на диску протягом заданого часу, а не просто передає їх далі по

черзі[13]. Завдяки цьому нові мікросервіси зможуть перечитати історію тренувань користувача за останній місяць, просто підписавшись на вже існуючий топік, що сильно спрощує додавання нового функціоналу.

– Горизонтальне масштабування. Механізм порцій (partitions) дозволяє розподіляти навантаження між кількома екземплярами одного й того самого сервісу[12]. В результаті це означає, що навіть якщо ввечері різко зростає кількість активних користувачів, система продовжує обробляти дані без помітних затримок, просто розкидаючи події між різними частинами.

– Гарантії доставки. Kafka зберігає повідомлення у реплікованому журналі, тому навіть у разі відмови окремого брокера або тимчасового відключення споживача дані не втрачаються.

Використання Kafka фактично перетворює бізнес-логіку на безперервний потік подій. Наприклад, подія WorkoutCompleted може одночасно запустити кілька незалежних процесів: оновлення даних у PostgreSQL, скидання кешу в Redis для миттєвого відображення нових результатів та запуск аналітичного методу в окремій порції для побудови графіків прогресу. Система залишається досить гнучкою: щоб додати нову можливість, достатньо підключити ще одного споживача до потрібного топіка, не чіпаючи вже працюючі модулі.

2.4 Системи управління базами даних: PostgreSQL та MongoDB

PostgreSQL обрано для зберігання основних транзакційних даних – інформації про користувачів, їхні сесії, шаблони та права доступу. Ключова причина: ці сутності мають чіткі зв'язки між собою, і будь-яка неузгодженість – наприклад, підхід без прив'язки до сесії або сесія без власника – є помилкою, яку система не повинна допускати. PostgreSQL із його ACID-гарантіями виключає подібні ситуації на рівні бази [15]. В нашому випадку такі домени як: User, Workout, Template та Exercise має жорсткі зв’язки, які реалізуються саме завдяки реляційним моделям. Це дає змогу формувати складні SQL-запити й швидко

отримувати звіти про тренування, сесії, шаблони завдяки індексам та оптимізованим запитам.

Паралельно з цим, для підсистем, де на перший план виходять гнучкість структури та висока швидкість запису, використовується MongoDB. У аналітичній частині фітнес-застосунку типів даних та способів їх аналізу більше, ніж може здатися на перший погляд: одна аналітик потребує фіксації ваги та загального навантаження, інша – спирається на час, а при розширенні також можливі показники пульсу. Спроба вмістити всі ці данні в одну реляційну базу призводить до появи великої кількості порожніх стовпців або надмірно складних зв’язків, що ускладнює підтримку й уповільнює роботу. Документ-орієнтована модель MongoDB вирішує цю проблему, кожен запис про певну активність – це окремий BSON-документ із власним набором атрибутів [16]. Це дає змогу користувачам отримувати різноманітні типи схем та аналітики без необхідності чекати зміни схеми бази даних.

MongoDB також логічно застосувати в аналітичних модулях для горизонтального масштабування та високої пропускної здатності на запис. Оскільки Kafka генерує потік подій про дії користувачів, документна база природно підходить для накопичення таких різноманітних даних. З основних переваг:

– Динамічна схема. Нові метрики можна додавати до документів без зупинки бази та тривалих міграцій.

– Висока доступність. Механізми реплікації та шардингу дозволяють базі залишатися стабільною навіть за умов зростаючого обсягу даних.

– Відома модель даних. Оскільки клієнтська частина на React працює з JavaScript, отримання JSON документів із MongoDB практично не потребує додаткових перетворень.

Поєднання PostgreSQL і MongoDB працюють в одному проєкті але для різних задач. PostgreSQL відповідає за основу системи [17]. MongoDB, у свою

чергу, забезпечує гнучке зберігання тренувальних журналів, подій і додаткових метрик, не обмежуючи користувача жорсткими рамками.

2.5 Інструменти реалізації безпеки Spring Security, JWT, Redis

Центральним механізмом захисту в проєкті виступає Spring Security, це фреймворк, що працює на основі ланцюжка фільтрів Security Filter Chain [11]. У нашому випадку кожен вхідний запит до API послідовно проходить через низку контрольно-пропускних пунктів, де перевіряється його валідність. Головним елементом цього ланцюжка є кастомний фільтр JwtAuthenticationFilter. Його задача – перехопити запит, витягнути токен із заголовка Authorization, перевірити підпис і термін дії, і лише після цього пропустити запит далі до контролерів. Це відокремлює логіку безпеки від бізнес-логіки тренувань, роблячи код чистішим.

Для ідентифікації користувача використовується стандарт JWT. На відміну від класичних сесій на стороні сервера, JWT дає змогу зберігати всю ключову інформацію безпосередньо в середині токену: ідентифікатор користувача, його ролі, строк дії тощо. Це добре підходить для мікросервісної архітектури: будь-який сервіс може самостійно перевірити токен, маючи лише спільний секретний ключ, не звертаючись кожного разу до уявної центральної бази даних [12]. У результаті зменшуються затримки при обробці запитів і система масштабується майже лінійно. Однак виникає інша проблема: як відкликати токен, якщо користувач вийшов із системи або його пристрій опинився в ненадійних руках? До моменту закінчення строку дії токен формально залишається валідним, і сервер не знає, що його потрібно заблокувати.

Для вирішення цього питання в проєкті використовується Redis. Redis це - нереляційне сховище типу ключ–значення, що працює в оперативній пам’яті. У застосунку Redis виконує роль чорного списку токенів [18]. Коли користувач ініціює вихід із системи, ідентифікатор його JWT заноситься до Redis із часом життя, який дорівнює залишковому строку дії токена. Під час кожної перевірки

JwtAuthenticationFilter додатково звертається до Redis, щоб переконатися, що токен не заблокований. Що зберігає переваги авторизації на основі JWT, але водночас отримуємо контроль над активними сесіями.

Також для був реалізований механізм оновлення токену. Замість одного довгоживучого токена система видає короткий токен доступу, та окремий refresh-токен, який зберігається в базі. Навіть якщо токен доступу буде перехоплено, час для його використання буде обмеженим, а сама сесія зможе оновлюватися у фоновому режимі, без помітних переривань для користувача під час використання.

2.6 Засоби розробки клієнтської частини React, Tailwind CSS

Реалізація клієнтської частини програми базується на сучасному фронтенд-стеку, в основі якого знаходиться бібліотека React. Вибір цього інструменту зумовлений його компонентно-орієнтованою архітектурою. Кожен елемент інтерфейсу – від картки окремої вправи до інтерактивного таймера відпочинку чи блоку підсумкової статистики – реалізований як автономний компонент із власним станом та внутрішньою логікою. Завдяки подібній модульній архітектурі інтерфейс залишається добре контрольованим і легко розширюваним.

Одну з важливих ролей відіграє механізм віртуального DOM у React. Під час тренування користувач постійно змінює вагу, кількість повторень, додає або видаляє підходи. У такому випадку повне перезавантаження сторінки було б неприйнятним. React дозволяє оновлення лише тими елементами, де дійсно змінився стан. Наприклад, коли спортсмен фіксує новий підхід, оновлюється тільки відповідний рядок і пов’язані з ним підсумкові значення, а решта інтерфейсу залишається незмінною.

Окремою задачею стала робота зі складними вкладеними структурами даних де кожне тренування складається з набору вправ, а всередині кожної вправи зберігаються масиви сетів з різними параметрами. Щоб створити подібний

інтерфейс, у проєкті поєднано локальне керування через React Hooks – useContext, що дозволяє чітко розділити локальний інтерфейсний стан і серверні дані.

Зв’язок фронтенду з бекендом реалізовано через бібліотеку Axios. На рівні перехоплювачів налаштовано автоматичне додавання JWT-токена до кожного запиту, а також обробку ситуацій, коли строк його дії вичерпується. У таких випадках Axios на задньому плані виконує запит на оновлення сесії за допомогою refresh-токена.

Візуальна частина побудована на основі Tailwind CSS. На відміну від класичних підходів зі групою кастомних CSS-файлів, Tailwind дає більше інструментів, стилі задаються безпосередньо в розмітці за допомогою невеликих класів. Механізм сбросу стилів під час збірки гарантує, що в продакшн потрапляє тільки той код, який реально використовується.

Для роботи з формами, яких досить багато , використовується бібліотека React Hook Form. Вона мінімізує кількість зайвих перерендерів при кожному натисканні клавіші. У результаті складні форми з великою кількістю полів працюють плавно. Валідація даних відбувається безпосередньо під час введення.

Найважливішим є перетворення масивів числових значень із PostgreSQL та MongoDB на результат для моніторингу прогресу. За допомогою бібліотеки Recharts показники об’єму навантаження, інтенсивності та розрахункового одноповторного максимуму відображаються у вигляді інтерактивних графіків.

Обраний стек – React, Spring Boot 3, Kafka, PostgreSQL, MongoDB – активно застосовується у комерційній розробці, що підтверджується щорічними звітами JetBrains та Stack Overflow Developer Survey.

3 ПРОЄКТУВАННЯ СИСТЕМИ МОНІТОРИНГУ АКТИВНОСТІ

3.1 Архітектура мікросервісів та потоків даних

Проєктування архітектури GymTracker спирається на ідею декомпозиції функціонала на окремі сервіси, кожен із яких відповідає за свою бізнес-область. Яка відокремлює щоденну логіку ведення тренувань від важчих аналітичних обчислень і тим самим зберегти високу продуктивність системи навіть за умов зростання кількості користувачів, або створення нових сервісів [7].

Ядром платформи є Workout Service. Саме він реалізує основну логіку управління тренувальним процесом, створення та редагування програм, фіксацію підходів, завершення тренування тощо. У цьому ж сервісі інтегровано модуль безпеки Spring Security. Сервіс відповідає за аутентифікацію користувачів через JWT-токени, взаємодію з Redis для контролю цих токенів, а також роботу з реляційною базою даних PostgreSQL, де зберігаються програми тренувань і результати виконаних сесій.

Окремим незалежним модулем виступає Statistics Service, який спеціалізується на роботі з великими обсягами даних і побудові аналітики. На відміну від основного сервісу, що працює з визначеною реляційною схемою, сервіс використовує документну базу MongoDB. Що дозволяє гнучко моделювати різні типи тренувальної активності, швидко будувати зведені звіти та не навантажувати основну транзакційну базу додатковими аналітичними запитами.

У системі чітко розділено два типи взаємодії. Для операцій, де користувач очікує миттєвий результат, таких як: реєстрація, оновлення профілю, створення нової вправи, використовується синхронний обмін через HTTP/REST. Натомість усе, що пов’язано з агрегацією даних, винесено в асинхронний потік подій, реалізований за допомогою Kafka.

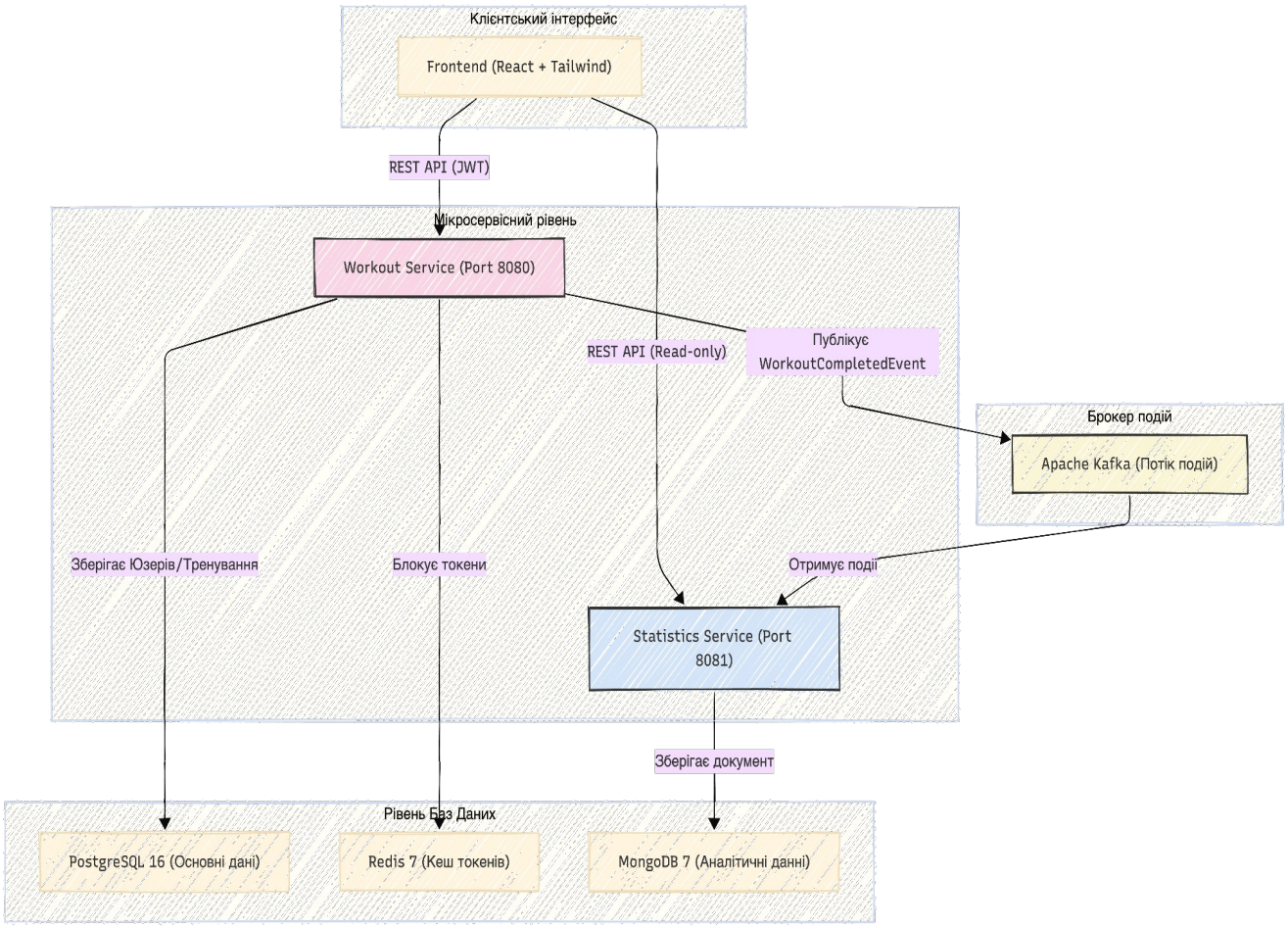


Рис. 3.1 Архітектура застосунку

Коли користувач завершує тренування, Workout Service формує подію WorkoutCompletedEvent і публікує її у відповідний топік Kafka. Далі запускається окремий ланцюжок обробки в Statistics Service, який відпрацьовує у фоновому режимі, не блокуючи основний HTTP-потік. На цьому етапі виконуються такі задачі:

– Перерахунок глобальної статистики користувача, загальний тоннаж, кількість завершених занять, середня інтенсивність.

– Фіксація нових персональних рекордів і розрахунок прогнозованого одноповторного максимуму.

– Оновлення показників активності – наприклад, кількість безперервних днів тренувань, які виступають додатковим джерелом мотивації.

На практиці слабка зв'язаність між сервісами вже виявила себе під час тестування: коли Statistics Service штучно відключали, Workout Service продовжував зберігати підходи без жодних помилок – події просто накопичувалися в Kafka і були оброблені після відновлення. Це вирішує реальну проблему коли тренажерні зали розташовані в підвалах з нестабільним інтернетом, і будь-яка аналітична затримка не повинна блокувати введення даних.

3.1.1 Обґрунтування розподілу системи на мікросервіси

Під час проєктування архітектури системи одним із початкових завдань було визначення оптимального способу розподілу функціональності між окремими компонентами. Після аналізу предметної галузі було визначенно, що моніторинг фізичної активності поєднує у собі кілька принципово різних типів функціональності: швидкі транзакційні операції логування тренувань, аналітичну обробку значних обсягів даних, а також побудову статистики та графіків. Якщо реалізувати всі ці процеси у межах єдиного монолітного застосунку це призвело б до високої зв’язаності компонентів, ускладнення масштабування та зниження швидкодії системи.

Основою архітектури став Workout Service — центральний сервіс, відповідальний за обробку транзакційних операцій у реальному часі. В його функціонал входить: управління тренуваннями, вправами, підходами, шаблонами програм, а також взаємодія з користувацьким профілем. Цей сервіс постійно працює з користувачем під час активного тренування. У таких обставинах важливими є мінімальна затримка відповіді сервера, швидке збереження даних та стабільність інтерфейсу. Через це Workout Service виконує лише легкі транзакційні операції і не містить складних аналітичних обчислень. І саме через це, найбільш підходящим варіантом була база даних PostgreSQL з її гарантіями до цілісності даних і швидку роботу порівнянно з ішими аналогам.

Окремо від основного тренувального сервісу було виділено Statistics Service, який відповідає за побудову статистики, агрегацію даних, розрахунок

тренувального тоннажу, визначення одноповторного максимуму та аналіз прогресу користувача. На відміну від Workout Service, цей модуль виконує більш важкі обчислення, які потребують додаткового часу та не є критичними до миттєвого виконання. Якщо б подібні обчислення виконувалися в основному сервісі, це б призвело до блокування операцій і зменшення швидкодії що суперечить вимогам.

Виділення аналітичного сервісу в окремий мікросервіс дозволяє створити асинхронний потік обробки даних. Після завершення тренування Workout Service публікує відповідну подію через брокер повідомлень Kafka, не очікуючи завершення подальших обчислень. Після чого Statistics Service асинхронно ловить події та виконує розрахунки у фоновому режимі.

Для подібного сервісу найкращим варіантом була база MongoDB, оскільки статистичні дані мають значно менш жорстку структуру і можуть змінюватися залежно від типу тренувань або нових метрик, які додаються до системи. Документоорієнтована модель дозволяє зберігати денормалізовані дані, які ідеально пілходять для швидкого читання, побудови графіків і формування статистики яка базується на історичних даних. Це також спрощує подальше розширення функціональності без необхідності міграції реляційної схеми.

Додатковою перевагою мікросервісного підходу є незалежне масштабування компонентів системи. Навантаження на основний та аналітичний сервіси має різний вигляд: основний сервіс генерує велику кількість коротких запитів, тоді як аналітичний модуль виконує менш часті, але значно важчі обчислення. Розподіл системи на окремі сервіси дає можливість масштабувати лише той модуль, який вимагає додаткових ресурсів у конкретний момент часу.

Також така архітектура забезпечує кращу ізоляцію у системі. Навіть у випадку тимчасових перебоїв аналітичного сервісу користувачі зможуть продовжувати логування тренувань і роботу з основним функціоналом системи.

3.1.2 Реляційна модель для Workout Service

Спроєктована реляційна модель на базі PostgreSQL є основою системи й виконує роль основного джерела для всіх транзакційних даних застосунку. Вибір цієї СУБД пов’язаний із дотриманням принципів ACID, які забезпечуючи цілісність інформації. Застосування UUID як первинних ключів по всій схемі додає системі безпеки через непередбачуваність ідентифікаторів. Головна таблиця USERS об’єднує облікові дані з базовими біологічними показниками, прив’язка до таблиці REFRESH_TOKENS реалізує механізм керування сесіями.

Логіка організована у вигляді трирівневої структури, яка розкладає кожне тренування на окремі атомарні дії. Таблиця WORKOUT_SESSIONS фіксує загальний контекст тренування – час, користувача, тип сесії, – тоді як WORKOUT_EXERCISES та EXERCISE_SETS деталізують послідовність виконаних вправ і підходів. У таблиці EXERCISES: за допомогою поля is_custom і посилання на автора запису відрізняються базові вправи від користувацьких. Це дозволяє користувачу вільно персоналізувати свій набір вправ, не змішуючи з загальним каталогом.

Тренування реалізовані через блок таблиць WORKOUT_TEMPLATES та TEMPLATE_EXERCISES, які виступають шаблонами для майбутніх тренувань. У шаблонах зберігаються базові значення підходів і повторень, окремо від фактичних результатів, зафіксованих у сесіях. Поле based_on у сесіях дає змогу зв’язати конкретне тренування з шаблоном і порівнювати заплановане навантаження з реально виконаним.

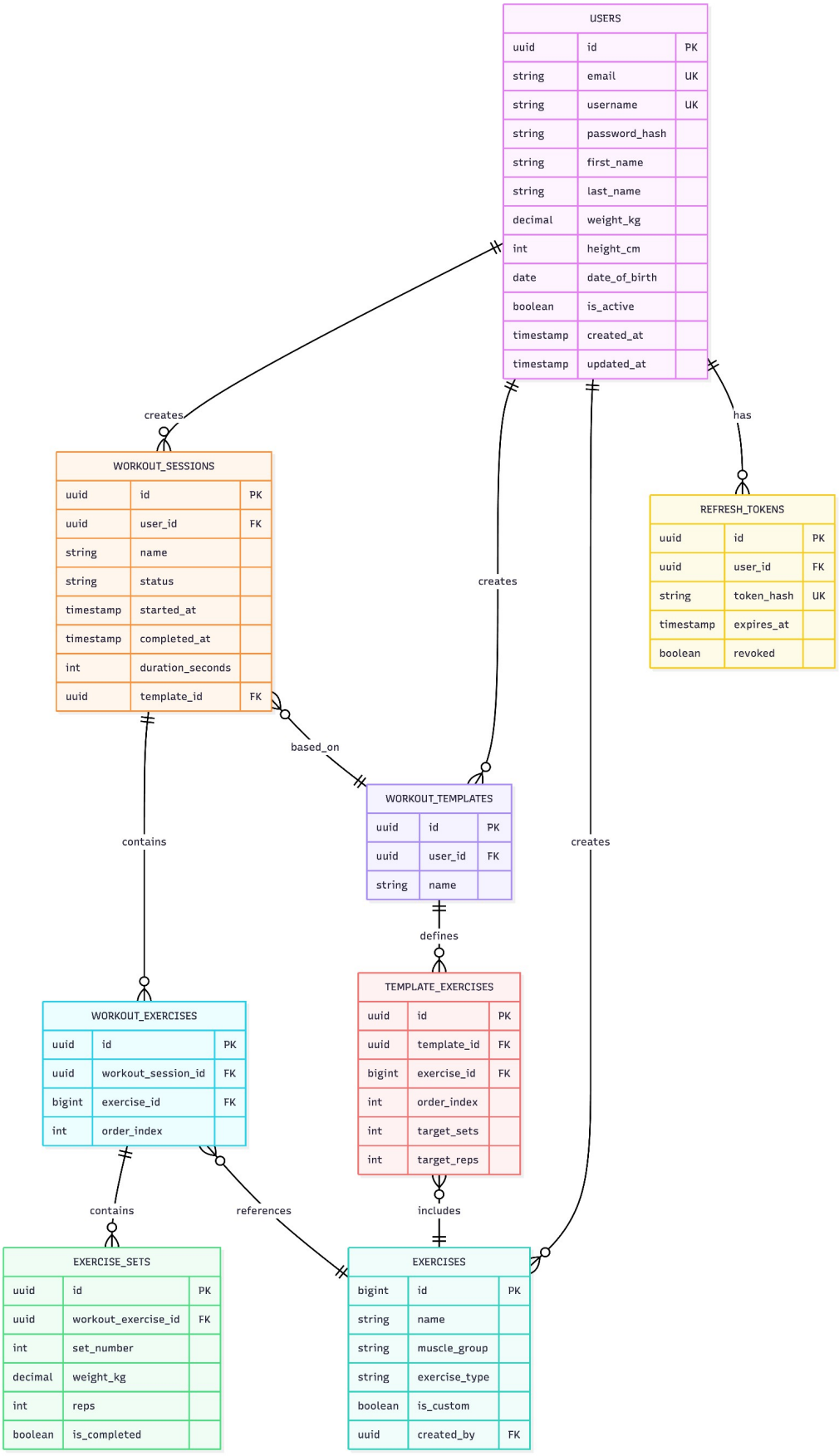


Рис. 3.2 Реляційна модель workout-service

3.1.3 Документоорієнтована модель для Statistics Service

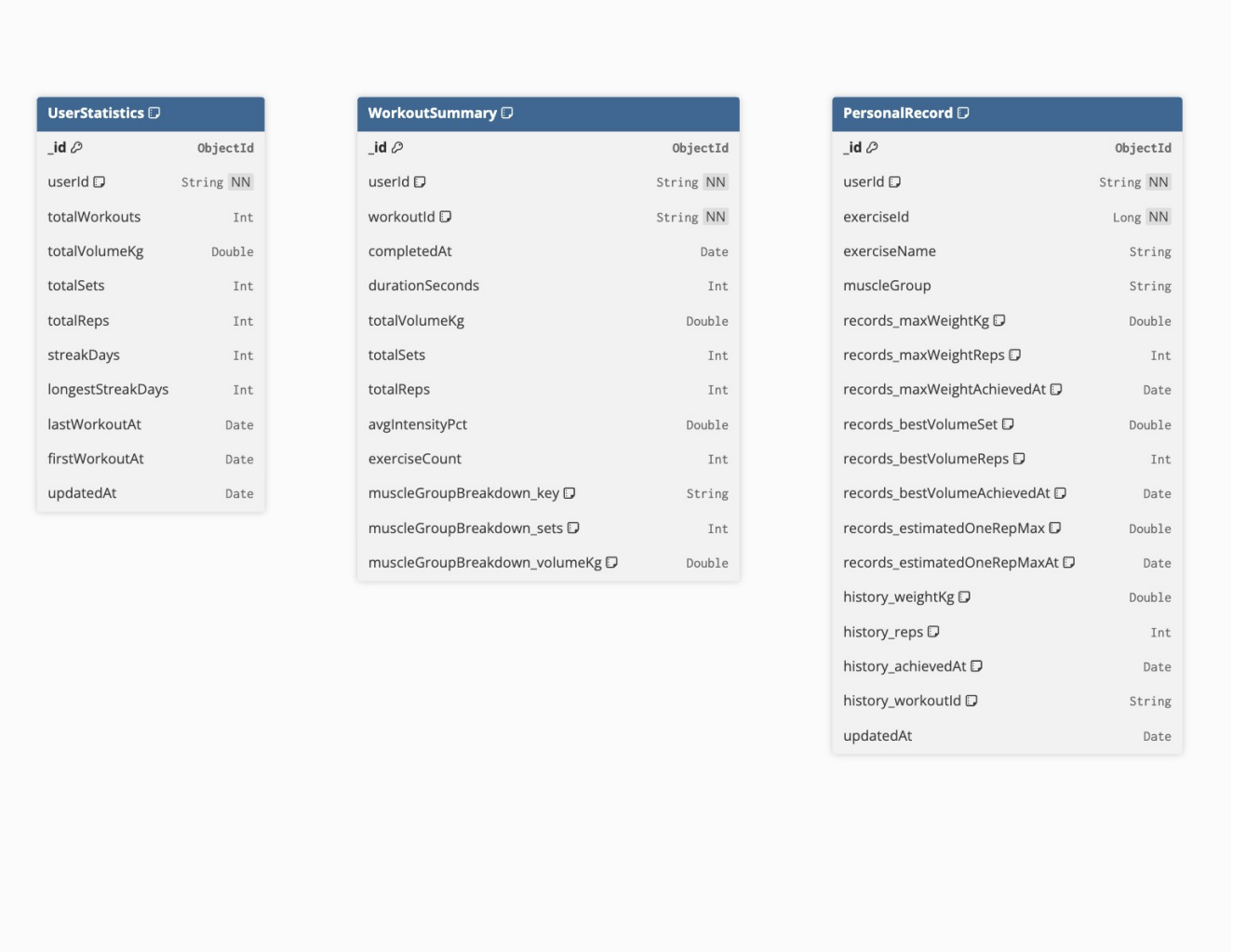


Рис. 3.3 Документна модель statistic-service

На відміну від транзакційного ядра, модуль аналітики та статистики використовує MongoDB як гнучке й швидке сховище даних. Вибір документної моделі пов’язаний із способом використання аналітичної інформації: її структура змінюється залежно від типу активності, де силове тренування, кардіо чи функціональний комплекс можуть містити різні набори показників. Формат BSON дає можливість зберігати складні об’єкти з вкладеними масивами рекордів, історією досягнень і додатковими метриками в рамках одного документа. Завдяки цьому при побудові дашбордів не потрібно виконувати важкі об’єднання таблиць, як у реляційній моделі.

Кажучи про Statistics Service колекції де USER_STATISTICS та PERSONAL_RECORDS спроєктовані як денормалізовані сховища, оптимізовані під постійне читання. Дані надходять асинхронно через брокер. Для клієнтської частини це означає, що складні графіки та підсумкові показники можна отримати в один запит без додаткової обробки на стороні браузера.

Ще однією важливою перевагою MongoDB є можливість розширення структури документів без зупинки бази й тривалих міграцій. Що ща потреби дає простір для розвитку.

3.2 Проєктування сценаріїв взаємодії користувача із системою

У цьому розділі описано, як конкретні дії спортсмена – старт тренування, фіксація підходу, натискання «Завершити» – перетворюються на послідовність запитів, публікацій у Kafka та фонових обчислень. Діаграми нижче показують не абстрактну архітектуру, а реальний маршрут даних крізь систему.

3.2.1 Діаграма потоків використання

Діаграма варіантів використання задає межі системи та описує ролі користувача. У філософії застосунку ми відмовились від перевантажених меню та не важливому функціоналі, зосереджуючись на тих функціях, які дійсно цінні для спортсмена:

– Управління тренувальним контекстом: створення власних вправ, формування та редагування шаблонів програм.

– Оперативне логування: фіксація підходів у реальному часі з миттєвим перерахунком об’єму та інтенсивності тренування.

– Ретроспективний аналіз: перегляд персональних рекордів, історії занять і динаміки об’єму навантаження за обрані періоди.

Кожен варіант використання є окремою частиною взаємодії, у межах якої система відповідає за валідацію вхідних даних і їх надійне збереження.

Наприклад, сценарій логування підходу включає не лише запис результатів у базу даних, а й перевірку прав доступу до поточної сесії через модуль аутентифікації. Лише після підтвердження, що користувач дійсно є власником сесії, Workout Service фіксує зміну, оновлює проміжні підсумки й, за потреби, генерує подію для аналітичного модуля.

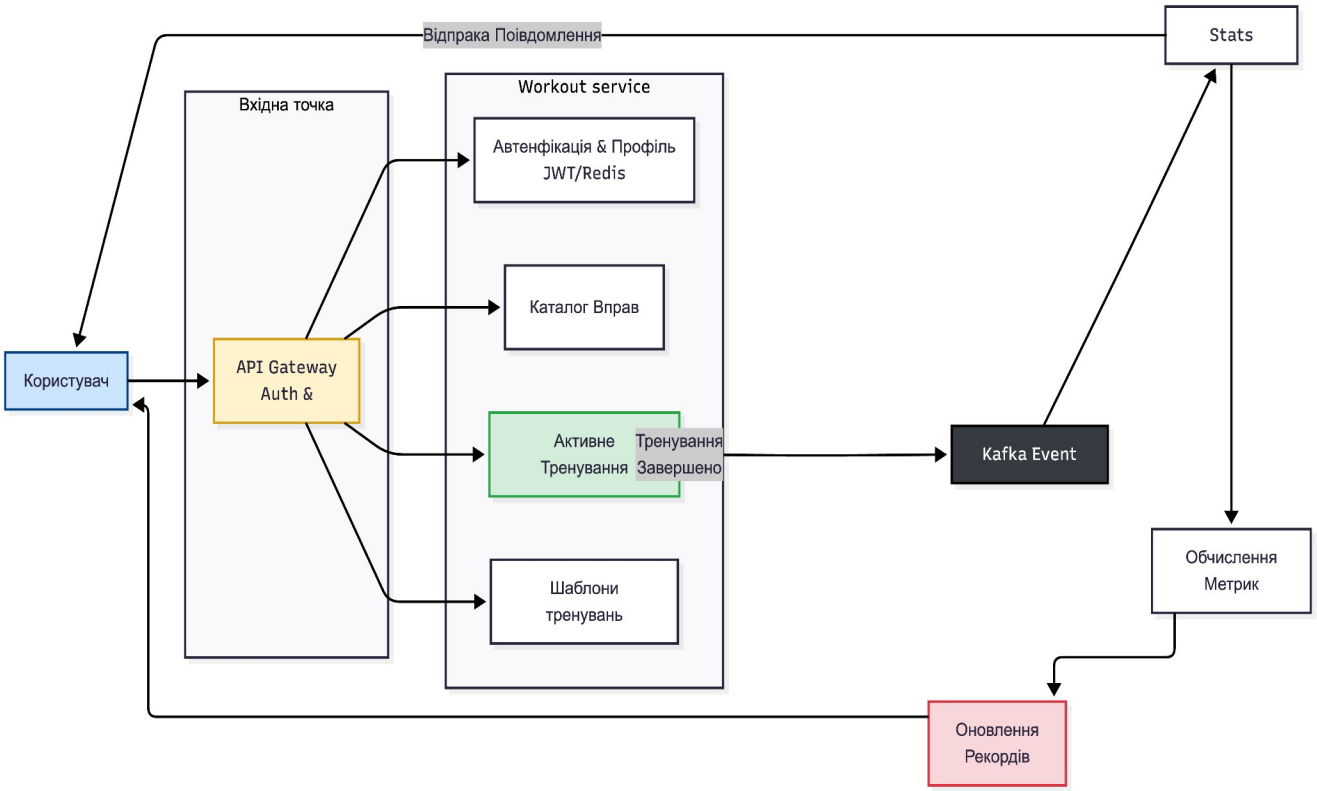


Рис. 3.4 Діаграма потоків використання

3.2.2 Діаграма послідовності асинхронної обробки статистики

Найцікавіший аспект роботи системи проявляється в асинхронній обробці подій. Діаграма послідовності демонструє повний шлях події – від натискання кнопки «Завершити тренування» до моменту, коли користувач бачить оновлені графіки прогресу.

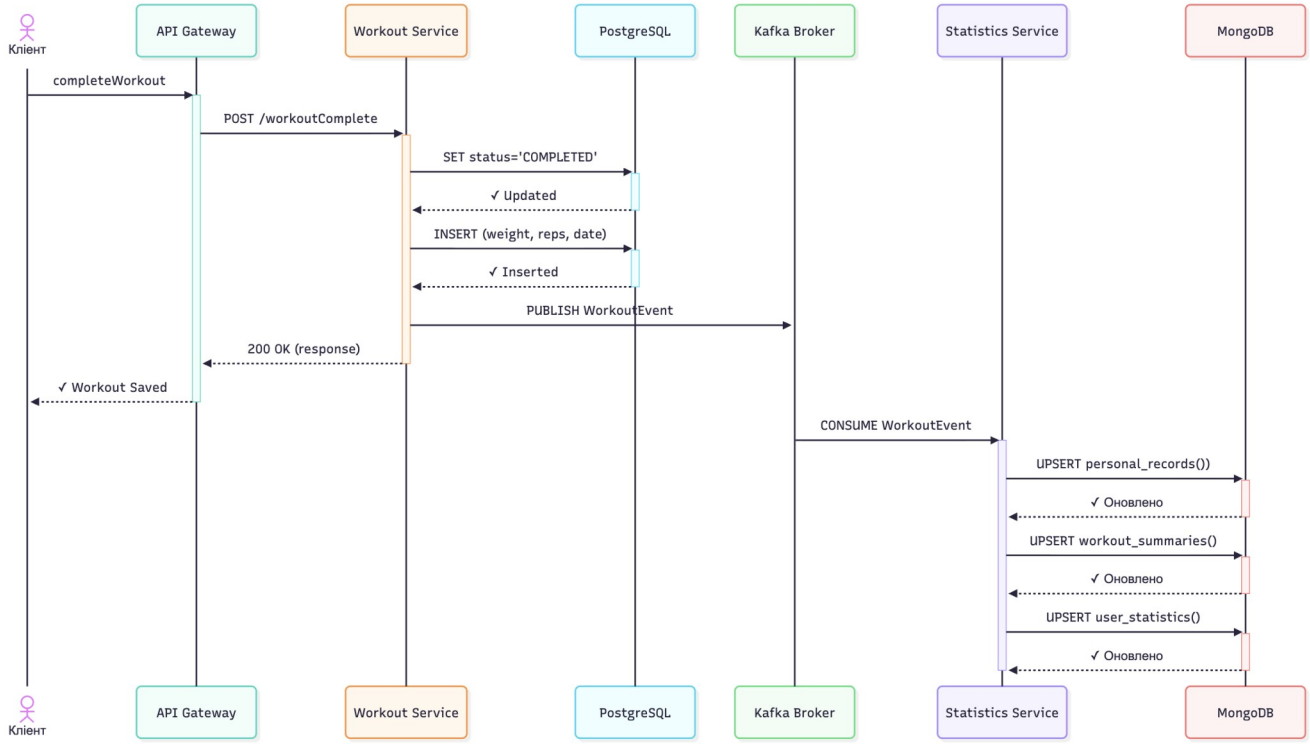


Рис. 3.5 Діаграма послідовності

У цьому сценарії frontend надсилає запит до Workout Service, який позначає сесію як завершену та фіксує відповідний стан у PostgreSQL. На цьому взаємодія з точки зору користувача майже закінчується: він одразу отримує підтвердження, що тренування збережено. Однак усередині системи процес тільки починається. Замість того щоб змушувати клієнт чекати на перерахунок складних метрик, Workout Service публікує подію у відповідний топік.

Statistics Service, який виступає підписником цього топіка, отримує повідомлення й запускає фонові обчислення:

– аналізує всі підходи тренування на наявність нових персональних рекордів;

– агрегує загальний об’єм навантаження за сесію (тоннаж) та пов’язані з ним показники інтенсивності;

– оновлює документну модель у MongoDB, щоб підготувати дані для побудови графіків і зведених панелей.

Розподіл обов'язків між сервісами вирішує практичну проблему де користувач бачить підтвердження про збереження тренування одразу, а важкі

обчислення – рекорди, тоннаж, розподіл по м'язах – відбуваються потім, у фоні, без жодного зв'язку з часом відповіді інтерфейсу.

3.3 Структурна діаграма класів доменного рівня

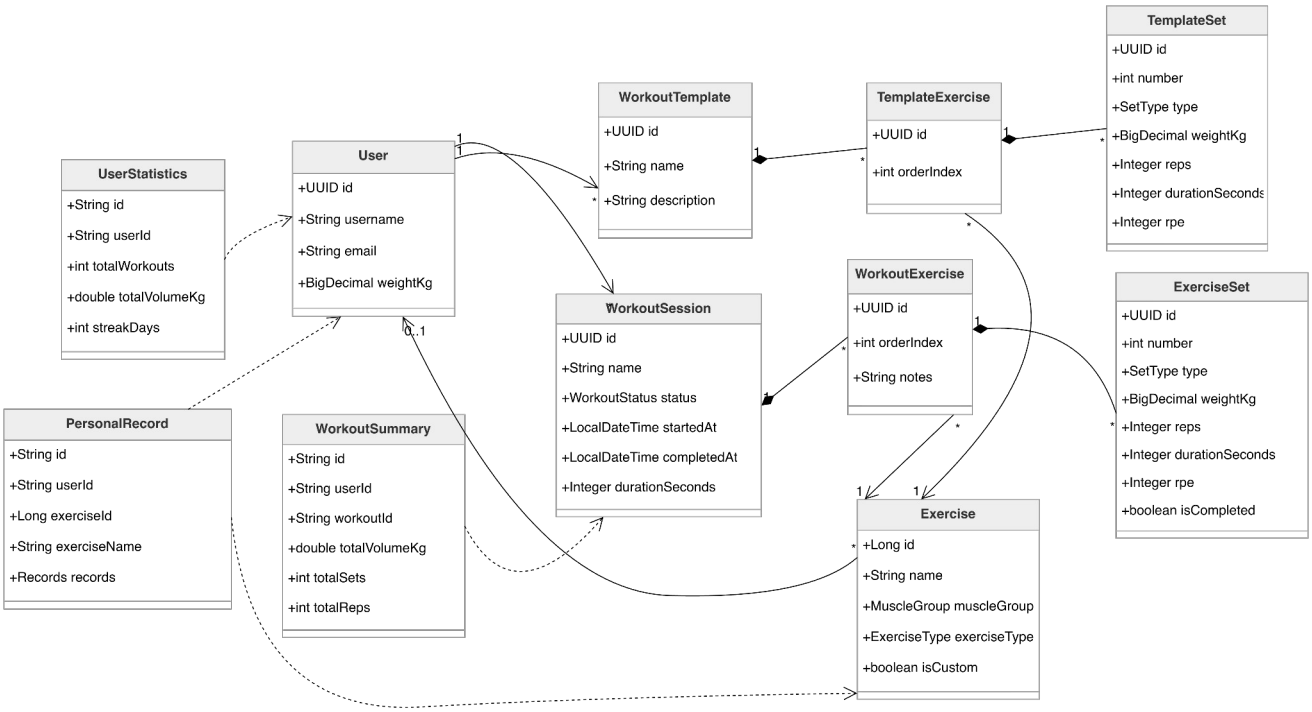


Рис. 3.6 Діаграма класів

Діаграма класів показує, як доменні об'єкти пов'язані між собою в пам'яті програми. На відміну від ER-діаграми, тут важливо не де саме зберігаються дані, а хто від кого залежить: WorkoutSession містить WorkoutExercise, а той – масив ExerciseSet. Такий погляд допомагає зрозуміти, які об'єкти мають сенс лише в контексті батьківського.

Клас WorkoutTemplate пов’язаний із вправами через проміжну сутність TemplateExercise. Ця побудова дозволяє зберігати планові показники окремо від фактичних результатів конкретної сесії. Завдяки цьому система може ефективно порівнювати заплановане навантаження з реально виконаним і будувати більш змістовну аналітику.

Діаграма також відображає принцип інкапсуляції даних користувача, усі ключові об’єкти – тренувальні сесії, шаблони програм, кастомні вправи – мають жорстку прив’язку до моделі User. Це дає ізоляцію інформації в багатокористувацькому середовищі.

При використанні композиції у зв’язках між WorkoutSession, WorkoutExercise та ExerciseSet це показує нам, що ці класи утворюють єдиний агрегат. Лог окремої вправи або підходу не має сенсу поза контекстом конкретної сесії, тому їх життєвий цикл повністю залежить від батьківського об’єкта.

3.4 Механізм забезпечення безпеки та відкликання токенів через Redis

JWT-авторизація вирішує питання масштабованості: кожен мікросервіс може перевірити токен самостійно, без звернення до центральної бази сесій. Але за цю зручність доводиться платити: стандартним способом заблокувати користувача до закінчення часу дії токена просто не існує. Якщо хтось заволодів токеном, він залишається чинним до його природного закінчення – зазвичай 15–30 хвилин. Для застосунку, що зберігає персональні тренувальні дані, це неприйнятний ризик, тому знадобилось додаткове рішення. У данному проєкті це питання вирішується за рахунок інтеграції Redis як шару оперативної пам’яті для динамічного керування доступом.

Основою механізму відкликання токенів є динамічний «чорний список» у Redis. Коли користувач ініціює вихід із системи, застосунок не обмежується видаленням токена на клієнті, а виконує список дій на бекенді:

– Ідентифікація. Із JWT витягується його унікальний ідентифікатор jti або хеш підпису токена. В нашому випадку ми порівнюємо хеш токену.

– Розрахунок часу життя. На основі поля закінчення дії обчислюється, скільки часу токен ще залишатиметься валідним.

– Запис в Redis. У Redis створюється запис із ключем, пов’язаним із цим ідентифікатором, і таймером життя, що дорівнює залишковому часу дії токена.

У результаті токен використовує місце в пам’яті Redis тільки протягом того періоду, коли ним теоретично міг би скористатися зловмисник. Після завершення цього інтервалу Redis автоматично видаляє запис, не потребуючи додаткового втручання.

Щоб зробити цю схему прозорою для користувача, перевірка токенів інтегрована безпосередньо в ланцюжок фільтрів. Для кожного вхідного запиту сервіс виконує два послідовні кроки:

– Статична валідація. Перевіряється цифровий підпис JWT та строк його дії – це відбувається локально й не потребує зовнішніх запитів.

– Динамічна перевірка. Виконується швидкий запит до Redis на наявність запису в «чорному списку» для поточного ідентифікатора токена.

Оскільки Redis працює в оперативній пам’яті, така перевірка займає мілісекунди й практично не впливає на загальний час обробки запиту [18]. Якщо токен знаходиться в «чорному списку», фільтр безпеки одразу перериває обробку й повертає статус 401 Unauthorized, навіть якщо формально строк дії токена ще не сплинув.

4 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ЗАСТОСУНКУ

4.1 Організація структури проєкту та налаштування мікросервісів

Реалізація програмного застосунку розпочалася з побудови модульної структури, що є основою для мікросервісної архітектури. Замість створення кількох окремих репозиторіїв, прийнято рішення використовувати монорепозиторій із чітким розподілом відповідальності між модулями. Це дає змогу підтримувати спільну кодову базу для спільних об’єктів та допоміжних утиліт, одночасно зберігаючи можливість незалежної збірки й деплою кожного мікросервісу. Центральним елементом виступає головний файл конфігурації збірки, який описує спільні залежності та координує взаємодію між сервісом тренувань, модулем аналітики й інфраструктурними компонентами.

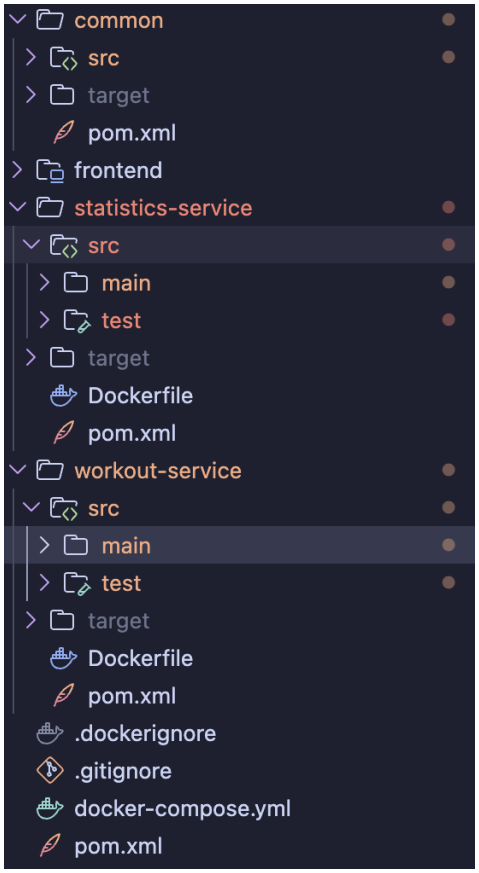


Рис. 4.1 Дерево проєкту

4.2 Реалізація транзакційної логіки та управління шаблонами тренувань

Серце бекенд-частини програми полягає не просто в збереженні рядків і записів у базу даних, а в тому, щоб гарантувати постійну транзакційну цілісність усіх дій під час виконання тренувань. У процесі тренування користувач постійно змінює вагу, кількість повторень, додає або видаляє підходи, а система повинна в будь-який момент мати консистентну картину того, що саме відбувається. Важливим моментом є натискання кнопки Завершити тренування: саме тоді застосунок фіксує фінальний стан усіх сутностей, пов’язаних із цією сесією. У цей момент запускається комплексний каскад операцій: система записує час завершення, розраховує загальну тривалість тренування, оновлює статуси вправ та підходів, підраховує проміжні підсумки і готує дані, які пізніше будуть використані методами для аналітики та сервісом статистики.

Щоб жоден із цих кроків не пройшов наполовину, на рівні сервісного шару Spring Boot використовується анотація @Transactional [9]. Вона об’єднує всі дії завершення тренування в одну логічну транзакцію: або зберігаються всі зміни, або не зберігається нічого. Якщо, наприклад, під час запису останнього підходу, оновлення статусу сесії або формування агрегованих значень відбудеться помилка (конфлікт доступу до бази, збій мережі, некоректні дані), ORM-механізм ініціює повний відкат транзакції. Це запобігає появі не цілісних тренувань, у яких частина підходів збережена, а частина – ні, або ж сесія значиться завершеною, але не має повної історії навантажень. Для користувача це питання довіри: втрата навіть одного підходу в рекордному сеті набагато болючіша за тимчасову технічну помилку, тому гарантія цілісності даних тут має першочергове значення.

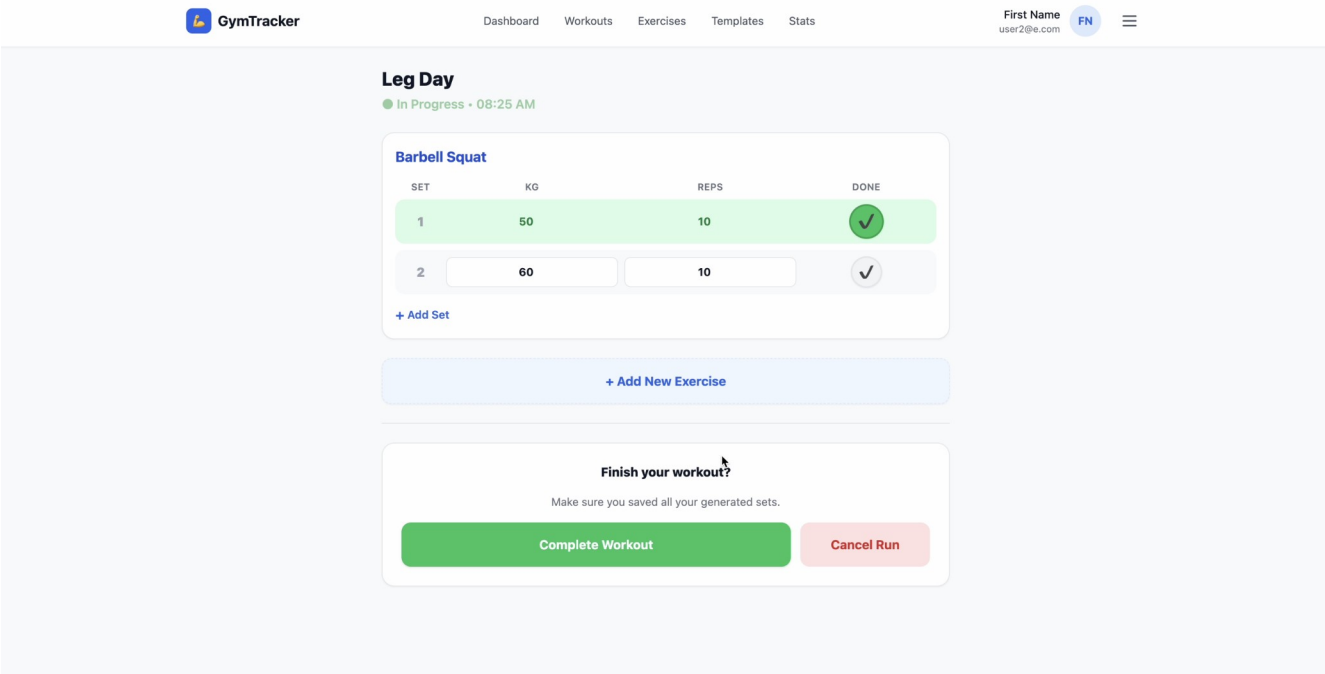


Рис. 4.2 Сторінка активного тренування

Окрему складність архітектури становить механізм управління шаблонами. Шаблон у застосунку – це не просто копія минулого тренування, а гнучке креслення, на основі якого створюється нова сесія. Користувач один раз налаштовує послідовність вправ, цільові ваги та кількість повторень, після чого цей шаблон можна багаторазово використовувати як стартову точку. Під час ініціалізації тренування за шаблоном система зчитує пов’язані з ним сутності TemplateExercise і на їх основі створює нові об’єкти WorkoutExercise та початкові ExerciseSet. План зберігається окремо від фактичних результатів, але водночас користувач отримує постійний стандарт для чергового тренування.

При цьому важливо витримати баланс між підказками та чистотою нових записів. З одного боку, користувач очікує, що програма підставить останні робочі ваги й орієнтовну кількість повторень, щоб не вводити всі значення вручну перед кожним підходом. З іншого – нове тренування не має перетворюватися на одноманітну копію минулого: користувач повинен мати можливість у будь-який момент відхилитися від плану, змінити вагу, додати вправу поза шаблоном чи видалити заплановану. Для цього використовується динамічне керування колекціями в межах однієї транзакції Hibernate: під час сесії користувач вільно

змінює набір вправ і підходів, а при збереженні тренування в базу потрапляє вже фактичний склад тренування – без ризику пошкодити оригінальний шаблон.

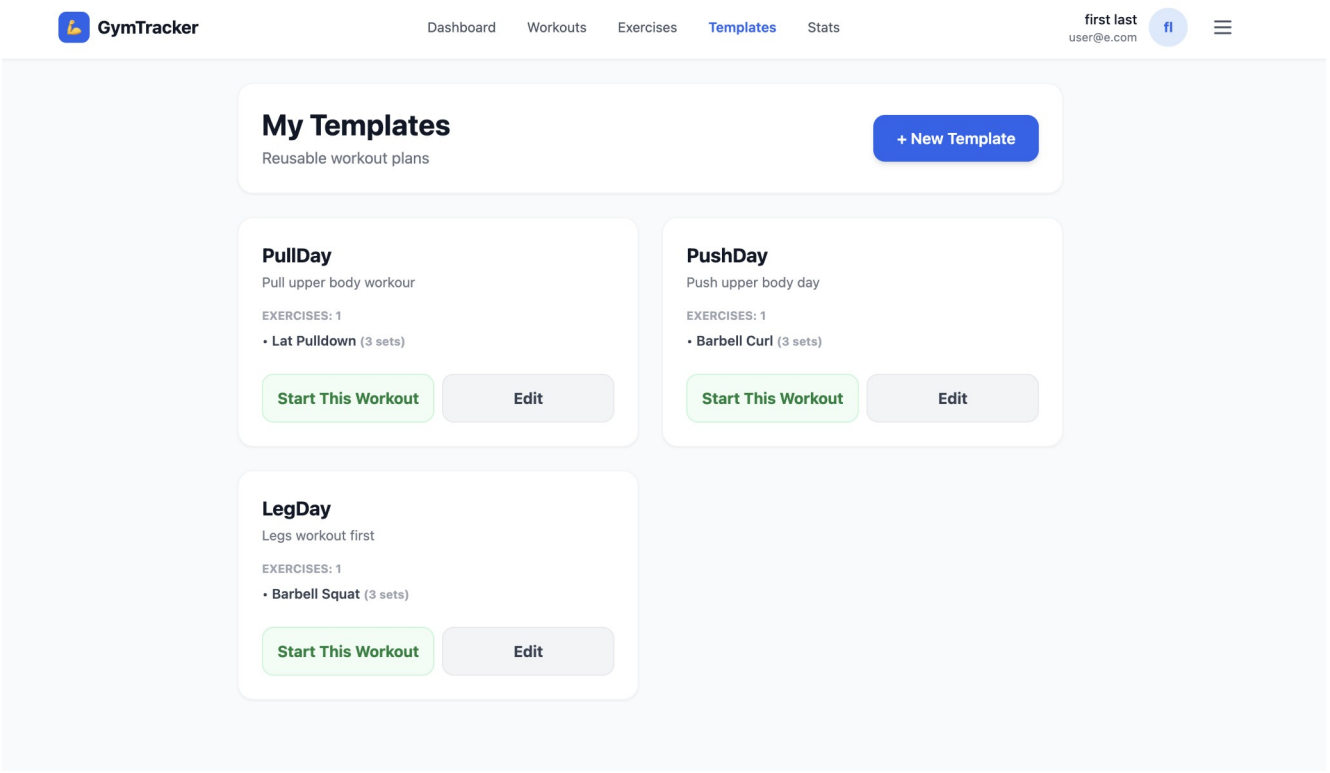


Рис. 4.3 Сторінка шаблонів тренування

Для оптимізації роботи з базою даних PostgreSQL ми використовуємо стратегію каскадного збереження [15]. Оскільки зв’язок між сесією, вправами та підходами є композиційним, JPA налаштовано так, щоб під час збереження головного об’єкта WorkoutSession автоматично оновлювався все дерево пов’язаних моделей. Це дозволяє уникнути десятків окремих SQL-запитів, замінюючи їх кількома груповими операціями або batch query. Цей підхід підвищує швидкість роботи застосунку, особливо в умовах слабкого мобільного інтернету, коли важливо мінімізувати час утримання з’єднання з сервером. Окрім суто технічних переваг, це забезпечує безперебійність і постійність у досвіді користувача: програма просто працює, тоді як складні процеси валідації та синхронізації відбуваються у фоні.

Щодо основник аспектів:

– Спеціальні handler-и перехоплюють помилки валідації і повертають зрозумілі повідомлення користувачу замість технічних помилок і кодів.

– Чіткий перехід станів з IN_PROGRESS → COMPLETED дозволяє уникнути випадкового редагування вже завершених тренувань, зберігаючи гарну валідаці і цілісність даних.

– При створенні сесії з шаблону система інтелектуально підставляє значення останніх робочих ваг, що економить час користувача в залі.

4.3 Реалізація асинхронного обробника подій та розрахунку рекордів

Реалізація асинхронного обробника подій стала відповіддю на одину із головних вимог до проєкту: як перетворити звичайні цифри виконаних підходів на корисну аналітику, не сповільнюючи при цьому роботу основного інтерфейсу. Учасником цього процесу є мікросервіс статистики, який у певному сенсі виконує роль аналітика. Поки користувач завершує сесію й відпочиває після останнього підходу, система у фоновому режимі завершує цикл розрахунків, не змушуючи його чекати на результати.

Уся взаємодія між сервісом логування тренувань і модулем аналітики організована через брокер повідомлень Kafka. Workout Service, завершуючи сесію, не рахує складні метрики самостійно, а публікує подію про завершення тренування у відповідний топік. Далі в гру вступає Statistics Service, який підписаний на ці події й обробляє їх незалежно від основного синхронного потоку. Це дозволяє чітко помітити дві задачі, а саме: швидке збереження необроблених даних і більш важкий аналіз. Навіть якщо одночасно відбуваються тисячі тренувань, інтерфейс залишається швидким, а розрахунки рекордів і агрегованих показників виконуються паралельно, без блокування користувацьких запитів.

Коли подія про завершення тренування потрапляє до асинхронного обробника, першим кроком запускається алгоритм розрахунку сумарного об’єму. Сервіс не обмежується збереженням ваги та кількості повторень, він проходить по всіх виконаних підходах, множить робочу вагу на кількість повторень і отримує тоннаж як по кожній вправі окремо, так і по сесії в цілому. Таким чином, фізична втома спортсмена перетворюється на конкретну числову метрику, яка відображає навантаження на м’язи і центральну нервову систему. Кажучи про реалізацію це виражається у денормалізації, де уже розрахований об’єм одразу записується до документа користувача в MongoDB. Завдяки цьому фронтенд може в один запит отримати готові суми без необхідності щоразу перераховувати сотні історичних підходів, що особливо важливо для швидкого відображення статистики на мобільних пристроях.

Окремим елементом функціоналу є розрахунок прогнозованого одноповторного максимуму (1RM). Для більшості користувачів вихід на свій максимум із реальною граничною вагою є некорисним і з точки зору безпеки, і з точки зору тренувального процесу. Тому застосунок бере на себе обчислення за формулою Еплі, сервіс аналізує робочі підходи і на основі ваги та кількості повторень обчислює теоретичний максимум сили. У коді реалізовано порівняння цього значення з уже наявним рекордом у базі. Якщо новий розрахунок виявляється вищим, система фіксує оновлений персональний рекорд і передає цю інформацію клієнтській частині для візуалізації.

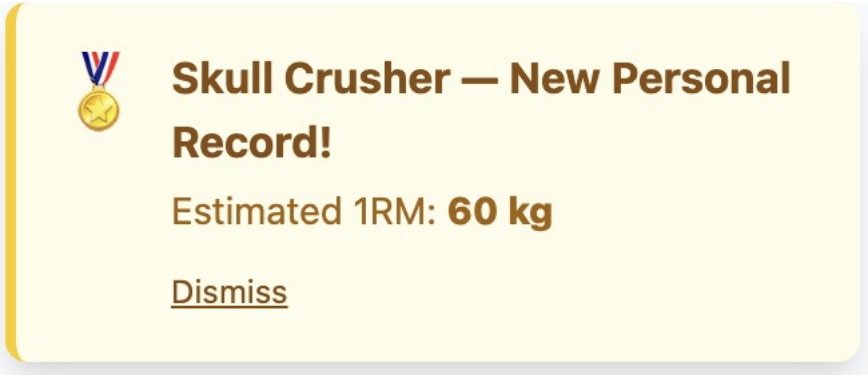


Рис. 4.4 Візуалізація персонального рекорду під час тренування

Користувач не просто бачить черговий набір цифр, а отримує значний заряд мотивації, та візуалізації прогресу, система підсвічує нові рекорди, дає зрозуміти, що прогрес реальний, і тим самим підтримує мотивацію продовжувати тренування [21].

Останім етапом життя процесу є формування аналітичного звіту за м’язовими групами. Сервіс групує підходи за цільовими м’язами, обчислює об’єм і відносну частку навантаження на кожну групу та формує структурований документ із підсумками тренування. Усі основні показники: сумарний об’єм, оновлений 1RM, розподіл навантаження по м’язових групах – зберігаються в MongoDB як єдиний документ.

4.4 Розробка інтерфейсу користувача та візуалізація статистики

Розробка інтерфейсу користувача є етапом, де мікросервісна логіка та математичні розрахунки перетворилися у візуальні форми. Головним завданням було створення фронтенду, яке б не перевантажувало спортмена під час тренування, але водночас надавало вичерпну аналітику для ретроспективного аналізу. Відповідно до сформованих раніше вимог, інтерфейс має поєднувати швидкість відгуку з повноцінною функціональністю. Використання React у поєднанні з Tailwind дозволило реалізувати концепцію подібного швидкого інтерфейсу, де кожен основний елемент керування знаходиться на відстані одного кліку. Ергономіка є одною з головних характеристик для frontend - великі кнопки для введення ваги, контрастні шрифти та адаптивність під мобільні пристрої є частиною основник вимог при розробці.

Використовувся Vite як інструмента збірки, що дозволило значно пришвидшити процес розробки та гарячого перезавантаження, який дозволяє оновлювати зміни у коді без повного перезавантаження сторінки.

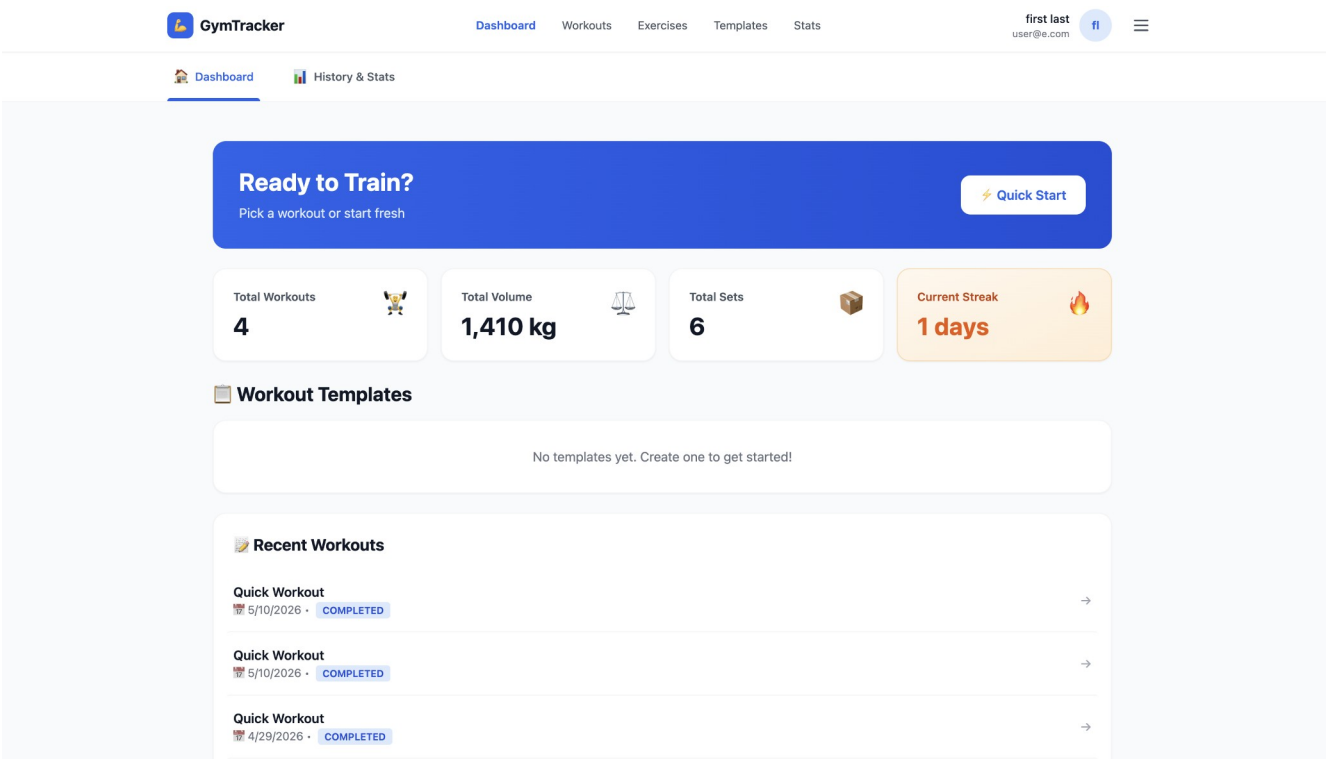


Рис. 4.5 Головна сторінка програми

Для відображення загального аналітичного висновку за м’язовими групами розроблено окремий екран статистики. Замість сирих числових даних інтерфейс презентує підсумки у вигляді структурованих інформаційних карток. Усі ключові показники – сумарний тренувальний об’єм, оновлений 1RM та відсотковий розподіл навантаження по цільових м'язах – візуалізуються на зручному дашборді, дозволяючи користувачеві миттєво оцінити ефективність виконаної роботи.

В основі роботи зі Statistics Service лежить ідея управління серверним станом за допомогою TanStack Query. Яка дозволяє не просто отримувати денормалізовані JSON-об'єкти, а й оперативно кешувати їх на стороні клієнта, автоматично синхронізуючи дані у фоновому режимі. В кінцевому результаті це забезпечує безперебійну роботу дашборду, особливо коли спортсмен бачить актуальний приріст інтенсивності та аналіз миттєво.

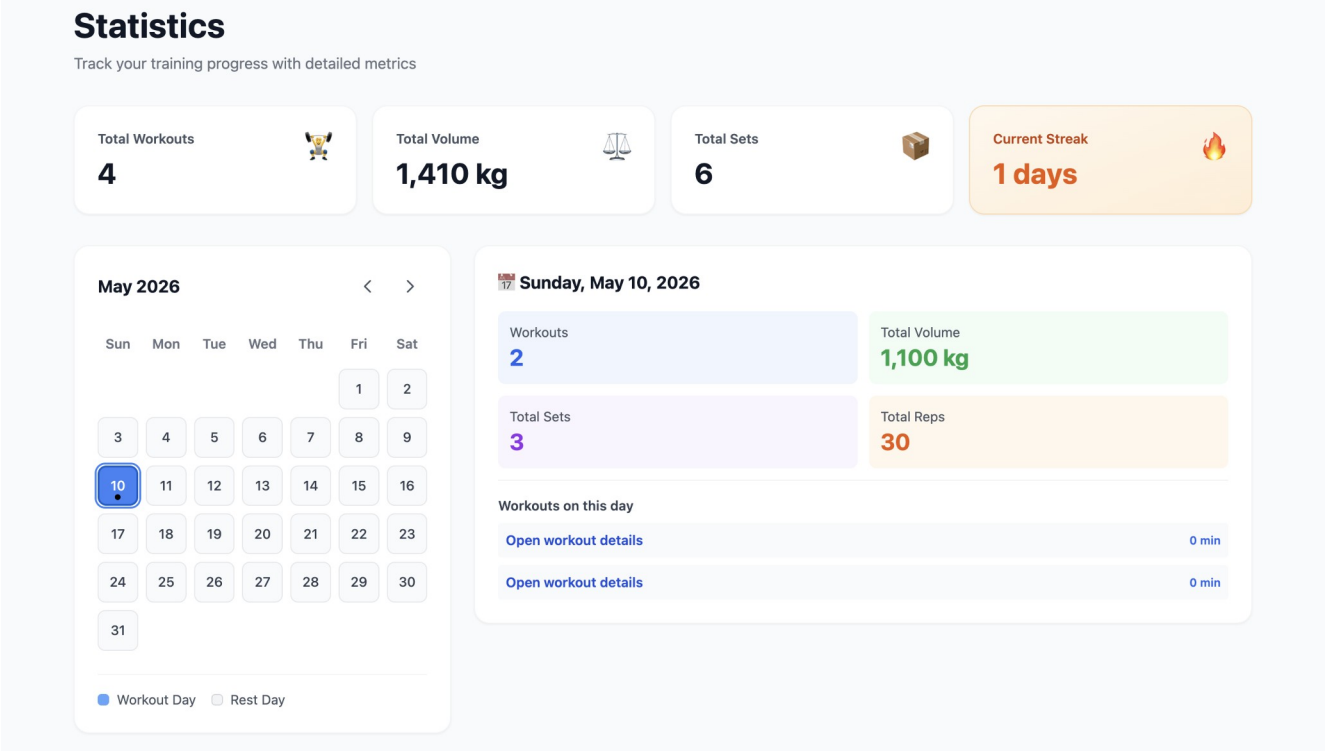


Рис. 4.6 Статистика користувача

Найбільш технологічно насиченою частиною інтерфейсу є блок візуалізації прогресу. Для відображення динаміки одноповторного максимуму та об’єму тренувань використано бібліотеку Recharts, яка дає змогу будувати плавні, лінійні графіки й комбіновані діаграми. Кожна точка на графіку є інтерактивною, при наведенні або натисканні користувач бачить деталі конкретної сесії, яка спричинила зростання показників або встановлення нового рекорду. Система автоматично фільтрує дані, дозволяючи перемикатися між оглядом по окремій вправі та зведеною статистикою по цілій м’язовій групі.

Це дає змогу спортсмену помічати періоди застою, коли прогрес у рекордах або тоннажу застряє на одному місці, і вчасно відкоригувати тренувальну програму. Важливо, що всі висновки ґрунтуються не на суб’єктивних відчуттях і записах на папері, а на реальних періодичних даних, які система збирає й обробляє.

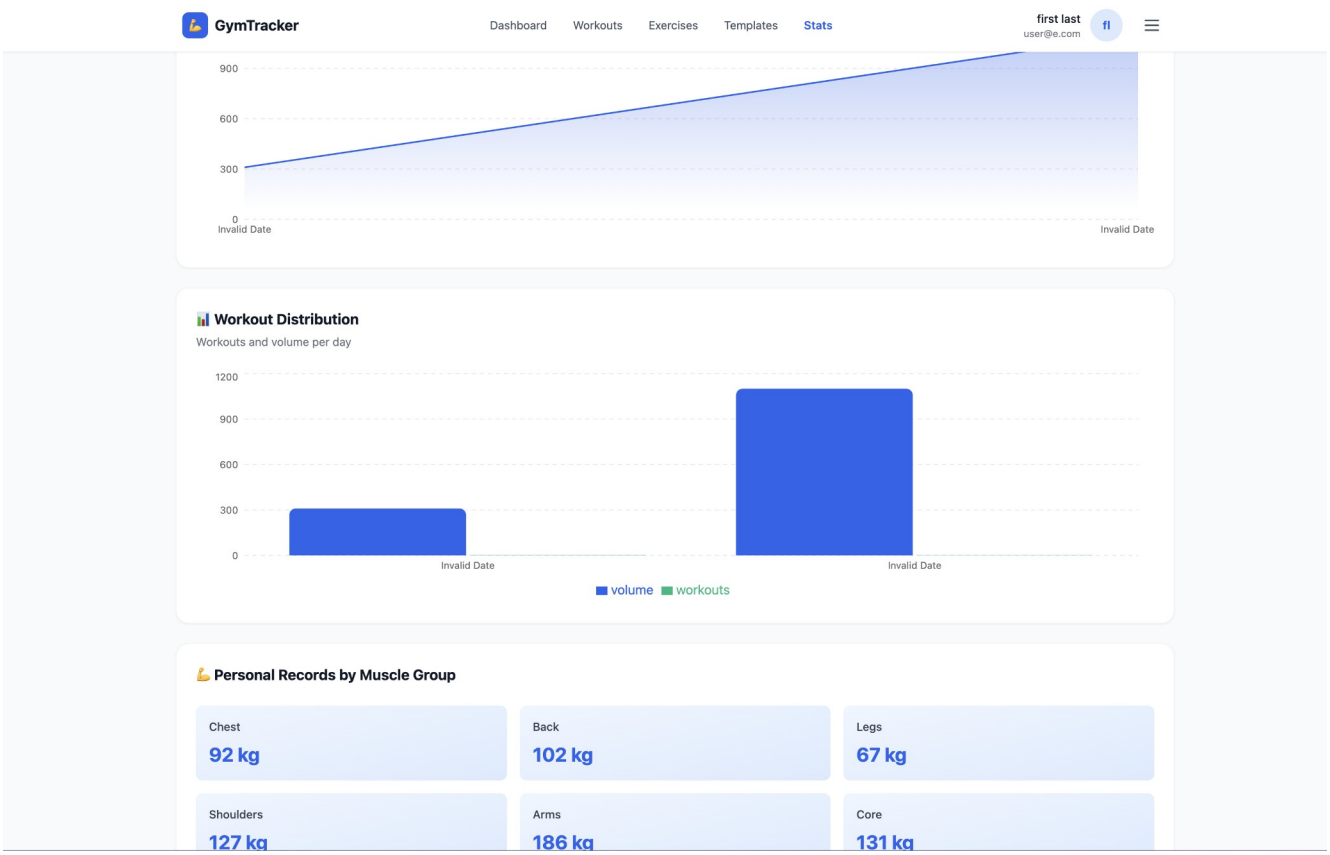


Рис. 4.7 Графік та рекордні показники користувачам

Важливим елементом є інтерактивний каталог вправ, який об’єднує в собі як системні пресети, так і вправи, створені самим користувачем. Завдяки компонентній архітектурі фронтенду пошук і фільтрація працюють у режимі реального часу.Кожна картка вправи містить короткий опис, основні технічні рекомендації та швидкий доступ до історії останніх підходів, тож користувачу не потрібно гортати старі записи в пошуках інформації про робочі ваги.

Безпека программи починається з інтерфейсу користувача. Архітектура фронтенд-частини спроектована за принципом суворого розділеня відповідальності між модулями. Кожен запит проходить через централізований шар Axios-перехоплювачів, які автоматично інтегрують JWT-токени та обробляють логіку відкликання сесій, роблячи механізм безпеки абсолютно прозорим для кінцевих компонентів.

4.5 Тестування програмного забезпечення

Тестування мікросервісної системи складніше, ніж тестування моноліту – помилка в одноіму сервісі може проявитись в зовсім іншому місці. Якщо Statistics Service отримає некоректну подію з Kafka і запише неправильний рекорд, користувач просто побачить хибні дані в профілі – і не зрозуміє чому. Саме тому процес тестування тут не зводиться до запуску одного набору unit-тестів наприкінці розробки. Щоб мінімізувати такі ризики, процес тестування було вбудовано в кожен етап розробки, не чекаючи самого кінця, тож більшість дефектів вдається виявляти ще на ранніх стадіях розробки [19].

Основні цілі тестування:

– Підтвердження математичної точності алгоритмів розрахунку в аналітичному сервісі.

– Валідація цілісності даних при збереженні складних об’єктів у PostgreSQL.

– Перевірка реакції системи до некоректних або граничних вхідних даних, які не мають існувати в системі.

– Коректна взаємодії мікросервісів через брокер повідомлень, перевірка роботи з обох сторін циклу.

4.5.1 Модульне тестування бізнес-логіки (JUnit 5, Mockito)

Перший рівень контролю якості реалізовано за допомогою модульних тестів використовуючи JUnit 5 та бібліотеку Mockito. Ідея цього етапу доволі проста, відокремити бізнес-логіку від усього, що залежить від зовнішнього середовища – баз даних, мережі, брокера повідомлень – і перевірити саме поведінку сервісних методів. Для сервісу тренувань це означає детальне тестування сервісного блоку, де за допомогою моків ми імітували роботу репозиторіїв та допоміжних компонентів. Що дозволило,перевірити, що сесію неможливо завершити без

жодного виконаного підходу, а також що перед будь-яким оновленням даних коректно перевіряються права доступу користувача.

Важливою вимогою було тестування математичних моделей у сервісі статистики. Використовуючи параметризовані тести (@ParameterizedTest), формули проводили через десятки різних комбінацій ваги та повторень, щоб переконатися у відсутності помилок округлення й адекватній роботі з великими значеннями. Mockito при цьому дозволив протестувати асинхронні слухачі подій Kafka в ізоляції, у тестах ми використовували штучні події й перевіряли лише те, що сервіс викликає потрібні методи розрахунку й оновлення статистики.

Підходи і інсрументи щоб досягти достатнього рівня покриття коду:

– AssertJ для читабельних перевірок. Це дозволило писати умови у формі, близькій до технічного завдання (наприклад, assertThat(workout).hasStatus(COMPLETED)), завдяки чому тести легше читати й підтримувати.

– Тестування винятків. Окремі сценарії перевіряли, що при спробі додати підхід до вже закритого тренування система коректно викидає CustomBusinessException, а не втрачає помилку.

– Керування життєвим циклом тестів. Використовувалася анотація @BeforeEach для ініціалізації тестових об’єктів і гарантувало, що кожен тест стартує з нуля, не залежить від результатів попередніх запусків. Що особливо важливо для стабільної роботи CI-процесу, де тести виконуються автоматично при кожному білдові.

Для контролю рівня покриття коду використовувався інструмент JaCoCo, який є стандартним рішенням для аналізу тестового покриття у Java-проєктах. Він інтегрується у процес збірки Maven/Gradle та дозволяє автоматично визначати, які частини коду були виконані під час запуску тестів. Інструмент формує детальні звіти щодо покриття класів, методів, умов та рядків коду, що дає можливість оцінити візуально відсоток покриття бізнес логіки по кожному классу застосунку.

За результатами аналізу JaCoCo, покриття бізнес-логіки застосунку перевищує 80%, що фактично свідчить про високий рівень тестового покриття основних функціональних частин системи. Найбільшу увагу було приділено сервісному рівню застосунку та рівню іструментів, оскільки саме там зосереджена ключова логіка роботи застосунку.

4.5.2 Інтеграційне тестування інфраструктури (Testcontainers)

Перехід від модульних тестів до інтеграційних став ключовим моментом , коли окремі фрагменти коду почали працювати разом із реальною інфраструктурою. Для мікросервісної архітектури це завжди значить особливий підхід: потрібно перевірити взаємодію з PostgreSQL, Redis і Kafka, але при цьому не перетворювати локальне середовище розробника на купу із десятків постійно запущених сервісів [19]. Вирішено це було за допомогою технології Testcontainers, яка перетворює інтеграційні тести на керовану оркестрацію тимчасових Docker-контейнерів, що існують лише протягом виконання тестів.

Testcontainers дозволили відмовитися від не зовсім професійної практики тестування на in-memory базах даних (на кшталт H2), які часто відрізняються від баз даних по типу PostgreSQL, як за синтаксисом SQL, так і за підтримкою специфічних можливостей. У нашому випадку це було важливо, зокрема, для роботи з JSONB-полями та нетривіальними індексами. Під час запуску інтеграційних тестів Spring Boot разом із Testcontainers автоматично підіймає новий екземпляр PostgreSQL, на який завантажуються всі схеми. Це дає впевненість, що запити, які успішно пройшли тестування, працюватимуть так само й у продуктовому середовищі. Фактично ми отримуємо інфраструктуру як код, і розробнику не потрібно додатково налаштовувати середовище, оскільки тести самі створюють потрібні контейнери й самі ж їх зупиняють після завершення.

Головним нюансом стало тестування асинхронної взаємодії через Kafka, яка вважається однією з найскладніших ділянок для перевірки в розподілених

системах. Завдяки модулю testcontainers-kafka, ми змогли запустити повноцінний сервіс спеціально для тестів. Це дало змогу реалізувати сценарії, максимально близькі до end-to-end, ду сервіс тренувань завершує сесію, публікує подію в реальний топік Kafka, а тестовий споживач очікує на надходження цього повідомлення і перевіряє його структуру та вміст. При допомозі бібліотеки Awaitility це дозволило коректно працювати з асинхронністю, тест дає системі кілька секунд на доставку події, після чого перевіряє результат.

В плані безпеки важливим напрямом інтеграційного тестування стало підтвердження коректної роботи механізмів пов'язаних з Redis. Сценарії перевірявся на практиці за списком, у тесті ініціюється вихід користувача із системи, токен додається до «чорного списку» в контейнері Redis, а наступний запит до API з тим самим токеном має бути відхилений фільтром безпеки зі статусом 401. Spring Boot для цього спрощює процес конфігурації завдяки анотації @ServiceConnection, яка автоматично підтягує параметри підключення до контейнерів, позбавляючи необхідності вручну прописувати порти й URL-адреси.

```
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
[INFO] --- failsafe:3.2.5:verify (default) @ statistics-service ---
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary for GymTracker Parent 1.0.0-SNAPSHOT:
[INFO]
[INFO] GymTracker Parent ................................. SUCCESS [  0.040 s]
[INFO] GymTracker Common ................................. SUCCESS [  0.795 s]
[INFO] GymTracker Workout Service ........................ SUCCESS [ 19.466 s]
[INFO] GymTracker Statistics Service ..................... SUCCESS [  7.512 s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  27.948 s
[INFO] Finished at: 2026-05-11T09:58:28+03:00
```

Рис. 4.8 Результат інтеграційних тестів

4.5.3 Тестування негативних сценаріїв та продуктивності

Справжня надійність усіх систем проявляється не тоді, коли все працює за ідеальним сценарієм, і один раз, а коли з'являються некоректні данні або відмови

окремих сервісів інфраструктури. Головною метою подібних тестувань є намагатися зламати інфрастуктуру як роблять це білі хакери [19]. Окремий блок тестів був спрямований на перевірку реакції системи на некоректні вхідні параметри: спроби зберегти від’ємну вагу обтяження, нульову кількість повторень чи дату початку тренування в майбутньому. Spring має велику перевагу по частині централізованого перехоплення помилок за допомогою анотації @ControllerAdvice, застосунко у всіх випадках повертав зрозуміле повідомлення про помилку з відповідним кодом, не допускаючи запису некоректних даних у базу.

Безпека є основою а тому тестування токенів стало важливим етапом. Тестувалися ситуації з підробленими JWT-токенами, токенами з простроченим строком дії, а також спроби змінити ідентифікатор користувача в URL-запитах. Ціль була проста - впевнитися, що користувач А ні за яких умов не зможе отримати доступ до тренувальних шаблонів або історії користувача Б. Інтеграційні тести показали, що зв’язка Spring Security та динамічної перевірки токенів у Redis миттєво блокує подібні запити. Додатково було змодельовано сценарій втрати зв’язку з брокером повідомлень Kafka: сервіс тренувань продовжував працювати в штатному режимі, зберігаючи всі ключові операції локально й гарантуючи, що користувач зможе завершити тренування навіть за тимчасових збоїв на стороні аналітичного модуля.

Ще один напрям тестування стосувався продуктивності обробки подій. Оскільки Kafka виконує роль центральної магістралі для даних, важливо було оцінити її пропускну здатність і затримку під час розрахунку рекордів та агрегованих показників. За допомогою інструментів для навантаження тестів побудовано сценарій максимального навантаження, коли сотні користувачів майже одночасно натискають кнопку завершення сесії.

Результати тестування показали, що завдяки неблокуючій філософії моделі обробки та можливості горизонтального масштабування Kafka середня затримка розрахунку аналітики не перевищувала 150–200 мс навіть за великої

інтенсивності близько 500 подій на секунду. Для користувача це означає, що оновлені рекорди та графіки з’являються в профілі практично одразу після завершення тренування.

Додатково перевірявся механізм перегріву у випадках, коли MongoDB не встигала обробляти запис великого обсягу даних, Kafka тимчасово утримувала події в черзі, не перевантажуючи оперативну пам’ять мікросервісів.

4.6 Документування REST API за специфікацією OpenAPI

Одна з відмінностей мікросервісної системи від монолітів яка створює додаткову складність звучить так: чим більше незалежних сервісів, тим складніше утримувати актуальне знання про існуючі ендпоінти та формати їхніх відповідей. В нашому контексті, де функціонують незалежні бекенд-сервіси з власним потоком авторизації, будь-яка невідповідність між очікуваннями фронтенду та реальністю сервера може стати критичною проблмою. Через це OpenAPI було обрано не просто як додаток, а як фундаментальний інструмент, навколо якого будувалася взаємодія між частинами системи з першого дня.

Для автоматичної генерації специфікацій у кожен із мікросервісів інтегровано бібліотеку springdoc-openapi-starter-webmvc-ui. Після підключення залежності кожен сервіс автоматично публікує два ресурси: JSON-документ за адресою /v3/api-docs та інтерактивний інтерфейс Swagger UI. Специфікація створюється динамічно під час старту застосунку, Spring сканує всі анотовані контролери й формує документацію.

Swagger UI став першим кордоном перевірки нових ендпоінтів ще до етапу автоматизованого тестування. Завдяки глобальній реєстрації схеми безпеки для JWT, інтерфейс відображає кнопку «Authorize», яка дозволяє один раз ввести токен і автоматично підставляти його в заголовок Authorization кожного запиту. Це виявилося дуже зручним та важливим при мануальному тестувані механізмів безпеки та перевірці чорного списку в Redis: після виклику логауту повторний

запит через Swagger UI повертав 401 Unauthorized, що значило що система працює як і спроектовано.

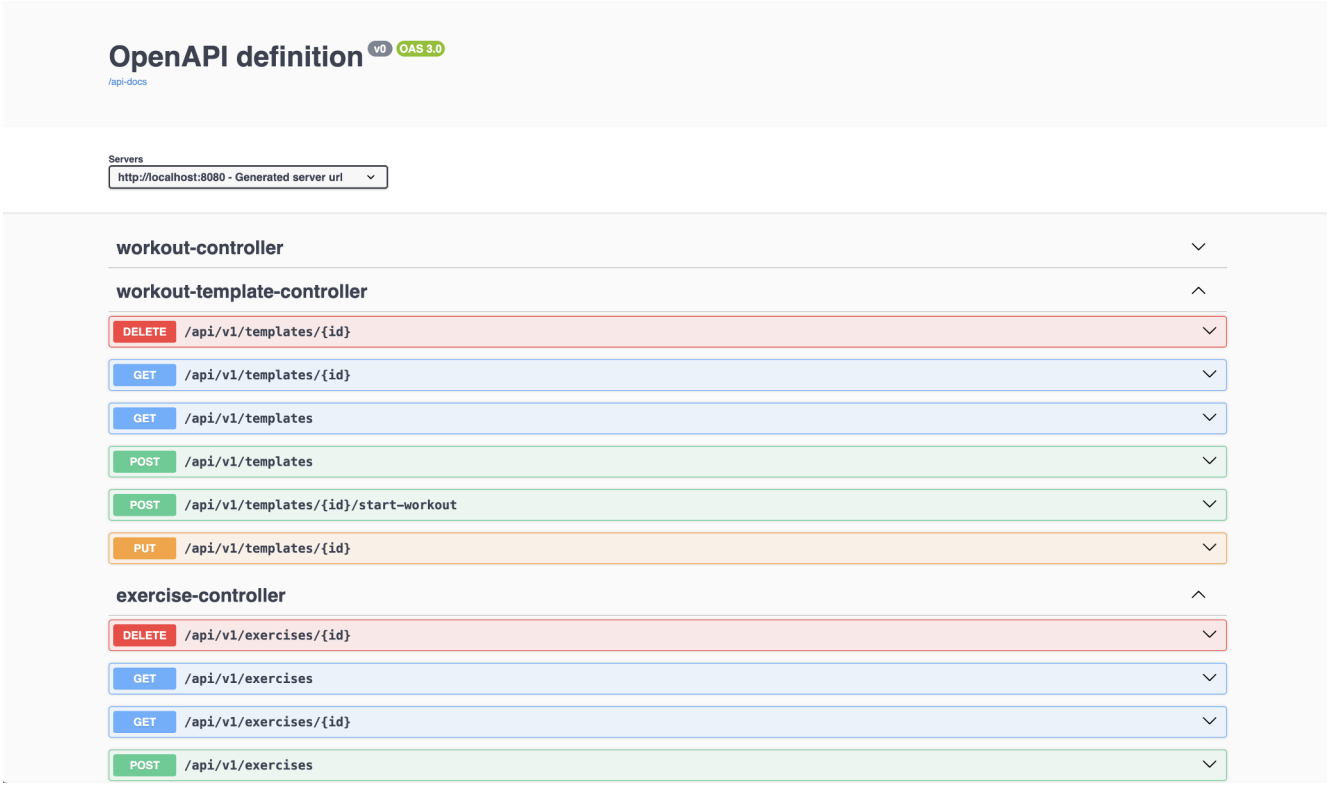


Рис. 4.9 Інтерфейс SwaggerUI

Найбільш відчутний практичний вплив OpenAPI полягає в його ролі як основного і єдиного джерела правди для клієнтської частини. JSON-документ специфікації використовувався безпосередньо при побудові механізму HTTP-клієнтів у застосунку, що дозволило одразу уникнути цілого списку проблем, пов'язаних із розбіжністю форматів даних. У фінальному вигляді проекту інтерфейс для Swagger умовно буде відключатися через конфігурацію профілів Spring для забезпечення безпеки, але в процесі розробки він залишається контрактом і важливим інструментом, який скорочує час налагодження та гарантує стабільну взаємодію між серверною частиною застосунку та інтерфейсом користувача.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи було розроблено розподілену програмну систему GymTracker, призначену для ефективного моніторингу тренувальної активності та статистичного аналізу спортивних досягнень. Розроблена система перетворює мануальний процес фіксації вправ на структурований, і надає атлету об’єктивну аналітику прогресу в майже реальному часі.

У ході роботи над проєктом були розв’язані такі ключові завдання:

1. Досліджено предметну область та ринок існуючих рішень. Аналіз популярних щоденників тренувань показав значний розрив між перевантаженими і неповноцінними інструментами фіксації й реальним інструментом спортсмена. Було виявлено, що більшість застосунків мають або застарілий інтерфейс, або обмежені можливості щодо глибокого аналізу силових показників.

2. Спроєктовано архітектуру системи на базі мікросервісного підходу. Данна структура дозволила чітко відокремити основний функціонал системи від аналітичних обчислень. PostgreSQL ідеально підходить під лінійні і структуровані задачі основного сервісу, тоді як MongoDB відповідає за швидкий денормалізований доступ до аналітики.

3. Реалізовано асинхронну систему взаємодії через Kafka. Побудовано подієво орієнтований проєкт, у межах якого розрахунок метрик виконується у фоновому режимі. Це зберегло високу швидкість відгуку клієнтської частини та забезпечило відмовостійкість системи в періоди пікових навантажень.

4. Впроваджено багаторівневу систему безпеки. На базі Spring Security та JWT реалізовано захищений доступ до REST API. Особливістю рішення стала інтеграція Redis для миттєвого відкликання токенів.

5. Розроблено зручний інтерфейс користувача. Використання React та Tailwind CSS дало змогу створити мінімалістичний і швидкий інтерфейс,

орієнтований на функціональність, а інтерактивні графіки забезпечують візуалізацію прогресу.

6. Проведено комплексне тестування та оцінку продуктивності. Завдяки використанню Testcontainers було створене оточення, максимально наближене до реального з PostgreSQL, Kafka та Redis які запускалися в тимчасових Docker-контейнерах в процесі тестування.

Таким чином, було успішно реалізовано всі поставлені завдання: від проєктування мікросервісної архітектури й налаштування асинхронної обробки подій до розробки інтерфейсу та тестування системи. Отримані результати підтверджують, що розроблена система повністю готова до практичного використання та підходить для подальшого масштабування і розширення функціоналу.

Робота пройшла апробацію. За її результатами було опубліковано наступні тези доповідей:

1. Лисенко Д.В., Калинюк Б.С. Визначення вимог до застосунку для моніторингу тренувальної активності. Матеріали II Всеукраїнської науково-технічної конференції «Виклики та рішення в програмній інженерії», 26 листопада 2025 року, Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. – К.: ДУІКТ, 2025. – С. 165-167.

2. Лисенко Д.В. Забезпечення інформаційної безпеки REST API у веб-застосунку для ведення тренувального щоденника. Матеріали Всеукраїнської науково-практичної конференції «Цифрова трансформація кібербезпеки», 29 квітня 2026 року, Київ, ДУІКТ. Збірник тез. – С. 48-49.

ПЕРЕЛІК ПОСИЛАНЬ

1. Hevy Workout Tracker & Planner. Офіційний сайт. URL: https://www.hevyapp.com/ (дата звернення: 02.03.2026).

2. Strong Workout Tracker. Офіційний сайт. URL: https://www.strong.app/ (дата звернення: 02.03.2026).

3. Jefit. Офіційний сайт. URL: https://www.jefit.com/ (дата звернення: 02.03.2026).

4. Pioske J. S., Arnett J. E., Ortega D. G., Roberts T. D., Schmidt R. J., Housh T. J. Cross-Validation of Original and Modified Equations for Estimating Bench Press One-Repetition Maximum from Repetitions to Failure in Recreationally Active Men. American Journal of Sports Science and Medicine. 2025. Vol. 13(1). P. 1–8. URL: http://pubs.sciepub.com/ajssm/13/1/1/index.html (дата звернення: 15.03.2026).

5. Marston K. J., Peiffer J. J., Newton M. J., Scott B. R. A comparison of traditional and novel metrics to quantify resistance training. *Scientific Reports.* 2017. Vol. 7. Article 5606. URL: https://www.nature.com/articles/s41598-017-05953-2 (дата звернення: 15.03.2026).

6. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud / M. Villamizar et al. 10th Computing Colombian Conference (10CCC). IEEE, 2015. P. 583–590. URL: https://ieeexplore.ieee.org/document/7333476 (дата звернення: 23.03 .2026).

7. Newman S. Building Microservices: Designing Fine-Grained Systems. 2nd ed. O'Reilly Media, 2021. 612 p. (дата звернення: 23.03.2026).

8. Kleppmann M. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. Sebastopol: O'Reilly Media, 2017. 616 p. (дата звернення: 25.03.2026).

9. Spring Boot Reference Documentation. Spring. URL: https://docs.spring.io/spring-boot/docs/current/reference/html/ (дата звернення: 01.04.2026).

10. Walls C. Spring in Action. 6th ed. Shelter Island: Manning Publications, 2022. 520 p. (дата звернення: 01.04.2026).

11. Spring Security Reference. Spring. URL: https://docs.spring.io/spring-security/reference/index.html (дата звернення: 02.04.2026).

12. Shingala K. JSON Web Token (JWT) based client authentication in Message Queuing Telemetry Transport (MQTT). *arXiv preprint arXiv:1903.02895*. 2019. DOI: 10.48550/arXiv.1903.02895. URL: https://arxiv.org/abs/1903.02895 (дата звернення: 02.04.2026).

13. Apache Kafka Documentation. Apache Software Foundation. URL: https://kafka.apache.org/documentation/ (дата звернення: 04.04.2026).

14. Narkhede N., Shapira G., Palino T. Kafka: The Definitive Guide. 2nd ed. Sebastopol: O'Reilly Media, 2021. 347 p. URL: https://www.oreilly.com/library/view/kafka-the-definitive/9781492043072/ (дата звернення: 04.04.2026).

15. PostgreSQL: The World's Most Advanced Open Source Relational Database. URL: https://www.postgresql.org/docs/ (дата звернення: 06.04.2026).

16. MongoDB Documentation. MongoDB. URL: https://www.mongodb.com/docs/ (дата звернення: 06.04.2026).

17. Barman U., Joshi K. Polyglot Persistence - Usage and Challenges. *URF Publishers*. 2025. URL: https://urfpublishers.com/article/view/polyglot-persistence-usage-and-challenges (дата звернення: 06.04.2026).

18. Redis: The open source, in-memory data store. URL: https://redis.io/docs/ (дата звернення: 06.04.2026).

19. Implementation Approach of Unit and Integration Testing Method Based on Recent Advancements in Functional Software Testing / Z. Y. J. Tan et al. *Journal of System and Management Sciences.* 2022. Vol. 12(4). P. 85–100. DOI: 10.33168/JSMS.2022.0406. URL: https://www.aasmr.org/jsms/Vol12/JSMS%20august%202022/Vol.12.No.04.06.pdf (дата звернення: 15.04.2026).

20. Bsdurrek P., Meng M. Simply Effective? Simplified User Interfaces in Software Tutorials. *Journal of Technical Writing and Communication.* 2024. Vol. 55(3). P. 223–246. DOI: 10.1177/00472816241262221. URL: https://www.researchgate.net/publication/382531554_Simply_Effective_Simplified_User_Interfaces_in_Software_Tutorials (дата звернення: 16.04.2026).

21. Zhou J., Lin Y., Wang W. Improving User Experience (UX) in fitness apps: A study on long-term motivational strategies to support health goals. *Lecture Notes in Computer Science.* 2025. Vol. 15792. URL: https://ro.ecu.edu.au/ecuworks2022-2026/6543/ (дата звернення: 16.04.2026).

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розробка вебзастосунку для комплексного моніторингу та аналізу показників фізичної активності користувача

Виконав студент 4 курсу
групи ПД-43
Лисенко Дмитро Вікторович
Керівник роботи
асистент кафедри ІПЗ Калинюк Богдан Сергійович

Київ – 2026

МЕТА, ОБ’ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи - спростити моніторинг фізичної активності шляхом створення програмного забезпечення для обліку тренувань, фіксації робочих ваг та відстеження і аналізу прогресу користувача.

Об’єкт дослідження - процес збору, збереження та аналітичної обробки даних щодо фізичної активності користувача під час тренувань.

Предмет дослідження - методи, засоби та технології розробки клієнт-серверного програмного забезпечення для моніторингу тренувального процесу на основі Java та мікросервісної архітектури.

ЗАДАЧІ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1. Провести аналіз предметної галузі моніторингу фізичної активності. Виконати порівняльний аналіз існуючих програмних засобів.
2. Сформувати функціональні та нефункціональні вимоги до системи.
3. Спроєктувати мікросервісну архітектуру застосунку, структуру баз даних і механізми взаємодії між сервісами.
4. Реалізувати серверну і клієнтську частину застосунку.
5. Провести модульне та інтеграційне тестування програмного забезпечення.

3

АНАЛІЗ АНАЛОГІВ

Критерії порівняння	Hevy	Strong	Jefit	FitNotes	GymTracker
Синхронізація та доступність даних	+	+	+	-	+
Адаптивний інтерфейс (UX/UI)	+	+	+/-	-	+
Повний безкоштовний функціонал	-	-	+	+	+
Відсутність реклами та соціальних функцій	-	+	-	+	+
Аналітичний функціонал	+	-	+	-	+
Асинхронність	+	-	-	-	+

4

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні вимоги

1. Система повинна забезпечувати реєстрацію, автентифікацію користувачів.
2. Система повинна надавати каталог вправ із групуванням за м’язовими групами.
3. Користувач повинен мати можливість створювати, редагувати та видаляти власні шаблони тренувань.
4. Застосунок повинен забезпечувати ведення тренувального щоденника з фіксацією ваги, кількості повторень і підходів.
5. Система повинна автоматично обчислювати тренувальні показники, зокрема тоннаж та одноповторний максимум.
6. Застосунок повинен забезпечувати збереження історії тренувань.
7. Система повинна візуалізувати статистичні дані та прогрес.

5

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Нефункціональні вимоги

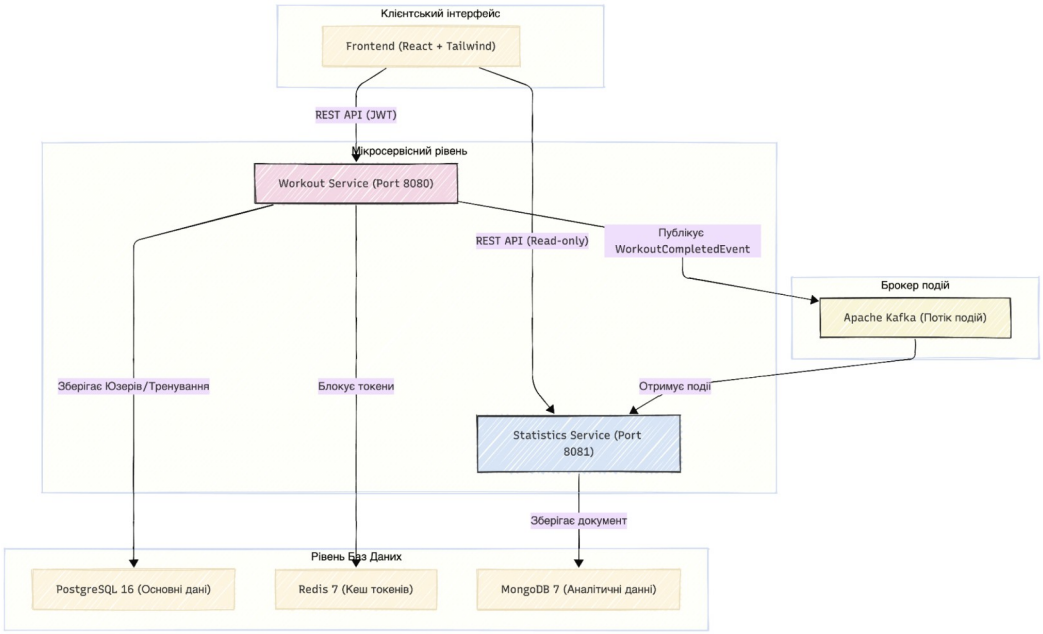
1. Час обробки типового REST-запиту не повинен перевищувати 200 мс.
2. Повне завантаження сторінки активного тренування повинно виконуватися не довше ніж за 1 секунду.
3. Система повинна забезпечувати безпечне зберігання персональних даних за допомогою BCrypt хешування.
4. Система повинна застосовувати мікросервісний підхід який забезпечує високу масштабованість і подальше розширення.
5. Основні елементи функціоналу повинні бути на відстані не більше 2 кліків.

6

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

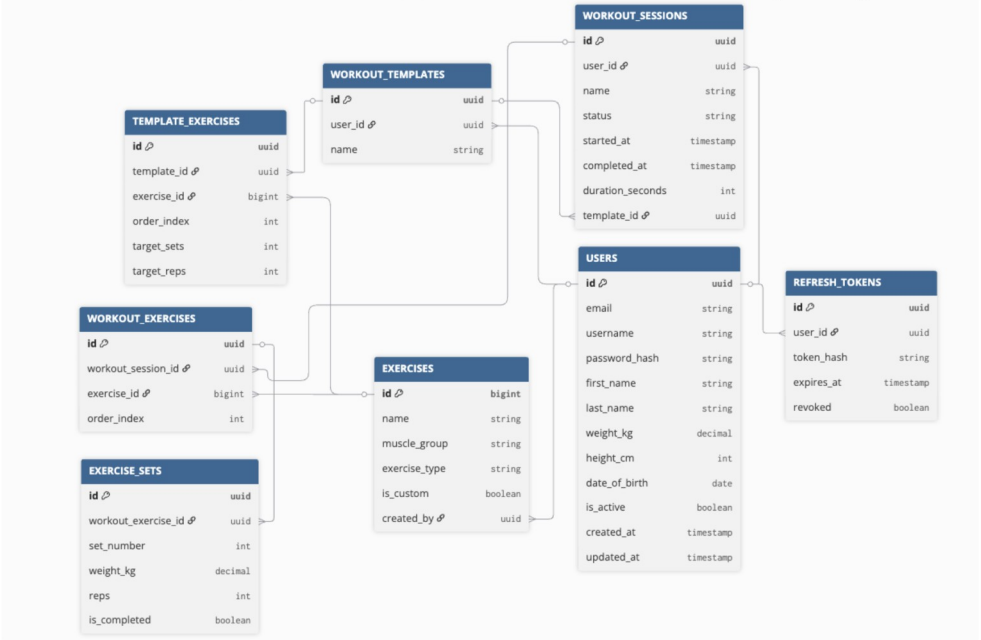
7

Діаграма архітектури застосунку



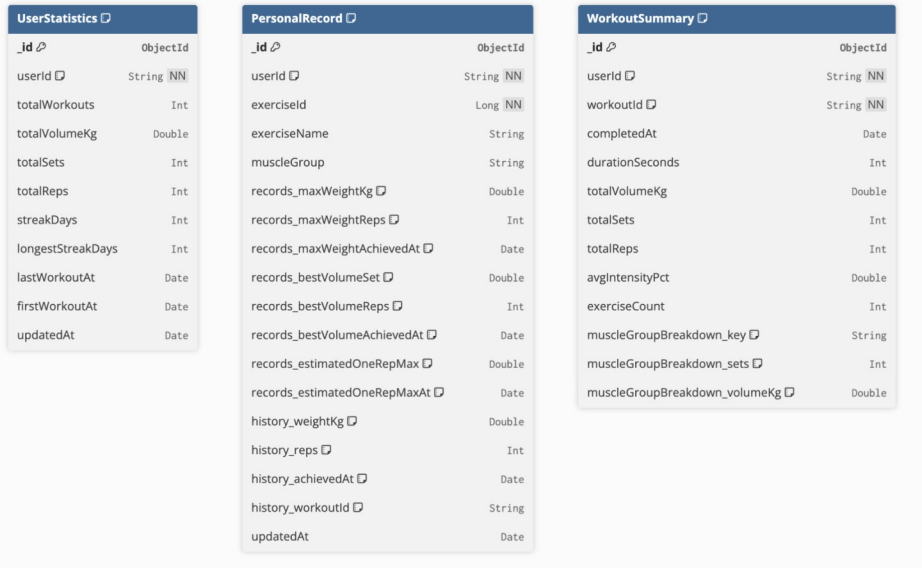
8

Реляційна модель основного сервісу



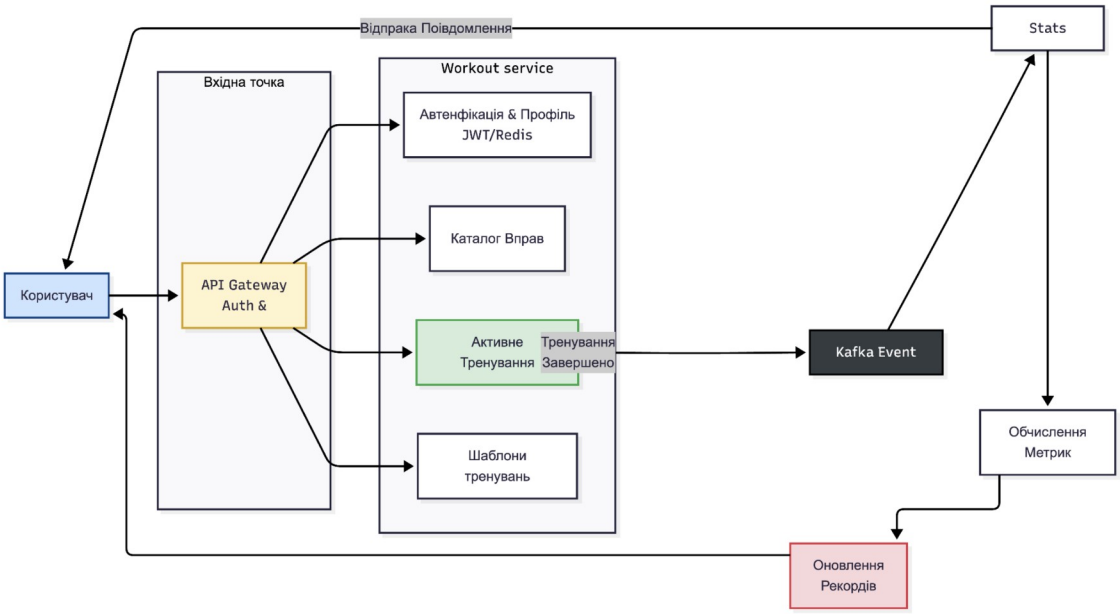
9

Документо-орієнтована модель аналітичного сервісу



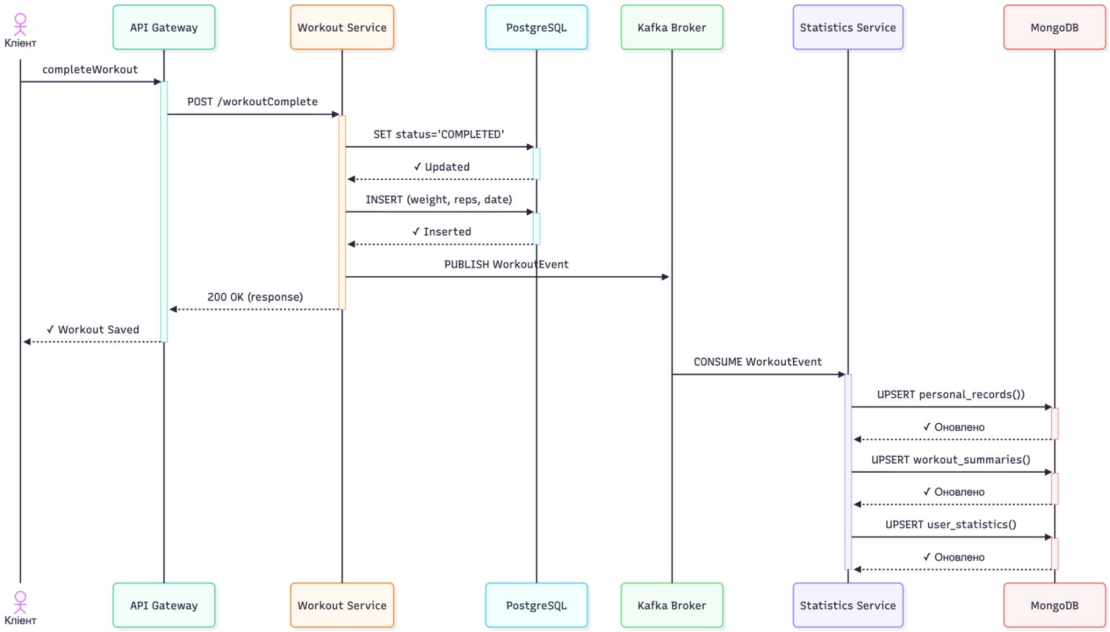
10

Діаграма основного потоку використання



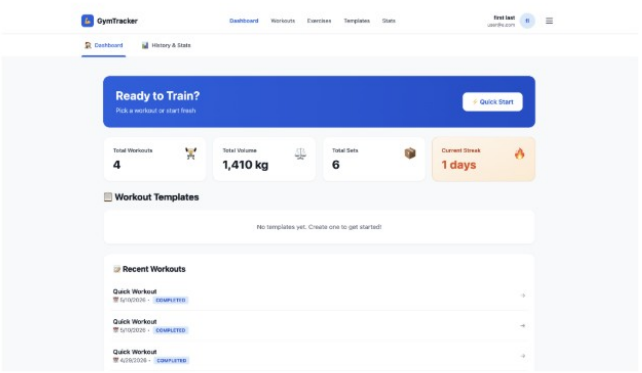
11

Діаграма послідовності

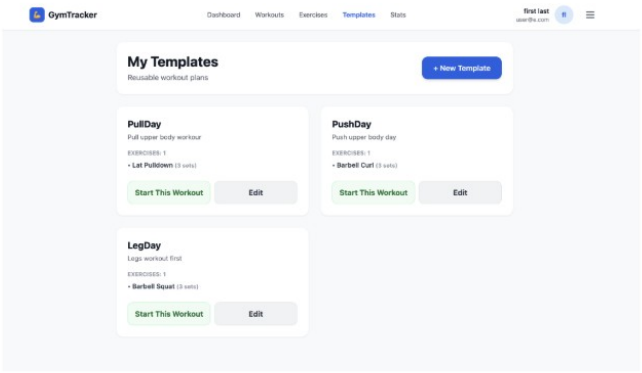


12

ЕКРАННІ ФОРМИ

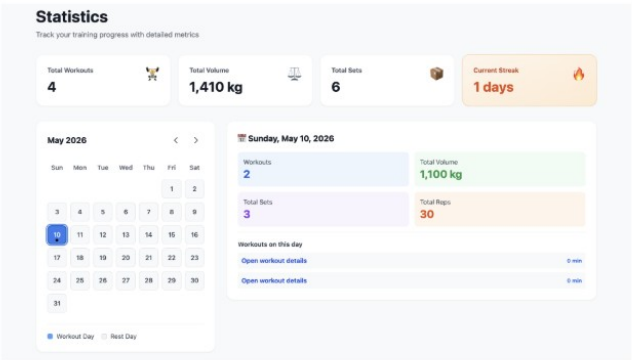


Головний екран

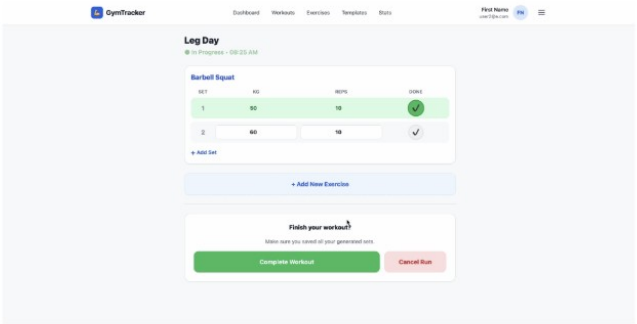


Екран шаблонів

ЕКРАННІ ФОРМИ



Екран статистики



Екран активного тренування

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

1. Лисенко Д.В., Калинюк Б.С. Визначення вимог до застосунку для моніторингу тренувальної активності. // Матеріали II Всеукраїнської науково-технічної конференції «Виклики та рішення в програмній інженерії», 26 листопада 2025 року, Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. – К.: ДУІКТ, 2025. – С. 165-167.

2. Лисенко Д.В. Забезпечення інформаційної безпеки REST API у веб-застосунку для ведення тренувального щоденника. // Матеріали Всеукраїнської науково-практичної конференції «Цифрова трансформація кібербезпеки», 29 квітня 2026 року, Київ, ДУІКТ. Збірник тез. – С. 48-49.

15

ВИСНОВКИ

1. Проведено аналіз предметної галузі та існуючих застосунків для моніторингу тренувань (Hevy, Strong, FitNotes, Jefit), визначено їхні основні недоліки та обґрунтовано необхідність розробки власної системи.
2. Сформовано функціональні та нефункціональні вимоги до програмного забезпечення, які дали основу для подальших архітектурних рішень.
3. Спроєктовано мікросервісну архітектуру системи, реалізовано структури баз даних PostgreSQL і MongoDB, а також механізм асинхронної взаємодії сервісів через Kafka, що дозволило перейти до розробки програмного застосунку.
4. Розроблено серверну частину застосунку на Java Spring Boot та клієнтський вебінтерфейс на React, що дозволило запустити застосунок і перейти до тестування.
5. Проведено модульне та інтеграційне тестування програмного забезпечення із використанням JUnit, Mockito та Testcontainers, що підтвердило коректність роботи системи та мікросервісів.

16

ДОДАТОК Б. ЛІСТИНГ ОСНОВНИХ МОДУЛІВ

```
package com.gymtracker.workout.domain.entity; import
jakarta.persistence.; import lombok.; import
org.springframework.data.annotation.CreatedDate; import
org.springframework.data.annotation.LastModifiedDate; import
org.springframework.data.jpa.domain.support.AuditingEntityListe
ner; import
org.springframework.security.core.GrantedAuthority; import
org.springframework.security.core.authority.SimpleGrantedAutho
rity; import
org.springframework.security.core.userdetails.UserDetails; import
java.math.BigDecimal; import java.time.LocalDate; import
java.time.LocalDateTime; import java.util.ArrayList; import
java.util.Collection; import java.util.List; import java.util.UUID;
@Entity
@Table(name ="users")
@EntityListeners(AuditingEntityListener.class)
@Getter @Setter
@NoArgsConstructor @AllArgsConstructor
@Builder
public class User implements UserDetails {
@Id
@GeneratedValue(strategy = GenerationType.UUID)
@Column(updatable = false, nullable = false)
private UUID id;
@Column(nullable = false, unique = true, length = 255)
 private String email;
 @Column(nullable = false, unique = true, length = 50)
 private String username;
@Column(name = "password_hash", nullable = false, length =
255)
 private String passwordHash;
@Column(name = "first_name", nullable = false, length =
100) private String firstName;
@Column(name = "last_name", nullable = false, length =
100) private String lastName;
@Column(name = "weight_kg", precision = 5, scale = 2)
private BigDecimal weightKg;
@Column(name = "height_cm")
private Integer heightCm;
@Column(name = "date_of_birth")
private LocalDate dateOfBirth;
@Column(name = "is_active", nullable = false)
@Builder.Default
private boolean isActive = true;
 @CreatedDate @Column(name = "created_at", nullable = false,
updatable = false)
private LocalDateTime createdAt;
 @LastModifiedDate @Column(name = "updated_at", nullable =
false)
private LocalDateTime updatedAt;
@OneToMany(mappedBy = "user", cascade = CascadeType.ALL,
orphanRemoval = true)
@Builder.Default private List workoutSessions = new
ArrayList<>();
@Override public Collection<? extends GrantedAuthority>
getAuthorities() {
return List.of(new SimpleGrantedAuthority("ROLE_USER")); }
@Override public String getPassword() { return passwordHash; }
@Override public String getUsername() { return email; }
@Override public boolean isAccountNonExpired() { return
isActive; }
@Override public boolean isAccountNonLocked() { return
isActive; }
@Override public boolean isCredentialsNonExpired() { return
isActive; }
@Override public boolean isEnabled() { return isActive; } }
```

```
package com.gymtracker.workout.domain.entity;
import jakarta.persistence.*;
import lombok.*;
import org.springframework.data.annotation.CreatedDate;
import
org.springframework.data.jpa.domain.support.AuditingEntityListe
ner;
import java.time.LocalDateTime;
import java.util.UUID;
@Entity
@Table(name = "refresh_tokens")
@EntityListeners(AuditingEntityListener.class)
@Getter @Setter @NoArgsConstructor
@AllArgsConstructor @Builder
public class RefreshToken {
@Id @GeneratedValue(strategy = GenerationType.UUID)
@Column(updatable = false, nullable = false)
private UUID id;
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "user_id", nullable = false)
private User user;
@Column(name = "token_hash", nullable = false, unique = true,
length = 255)
private String tokenHash;
@Column(name = "expires_at", nullable = false)
private LocalDateTime expiresAt;
@Column(nullable = false)
@Builder.Default
private boolean revoked = false;
@CreatedDate
@Column(name = "created_at", nullable = false, updatable =
false)
private LocalDateTime createdAt;}
```

```
package com.gymtracker.workout.service;
import
com.gymtracker.common.exception.DuplicateResourceException;
import
com.gymtracker.common.exception.ResourceNotFoundException;
import com.gymtracker.common.exception.ValidationException;
import com.gymtracker.workout.domain.entity.RefreshToken;
import com.gymtracker.workout.domain.entity.User;
import com.gymtracker.workout.dto.request.LoginRequest;
import com.gymtracker.workout.dto.request.RefreshRequest;
import com.gymtracker.workout.dto.request.RegisterRequest;
```

```
import com.gymtracker.workout.dto.response.AuthResponse;
import com.gymtracker.workout.mapper.UserMapper;
import
com.gymtracker.workout.repository.RefreshTokenRepository;
import com.gymtracker.workout.repository.UserRepository;
import com.gymtracker.workout.security.JwtService;
import com.gymtracker.workout.security.TokenBlacklistService;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Value;
import
org.springframework.security.authentication.AuthenticationManag
er;
import
org.springframework.security.authentication.UsernamePasswordA
uthenticationToken;
import
org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.time.LocalDateTime;
import java.util.Date;
import java.util.HexFormat;
import java.util.UUID;
@Slf4j
@Service
@RequiredArgsConstructor
@Transactional(readOnly = true)
public class AuthService {
private final UserRepository userRepository;
private final RefreshTokenRepository refreshTokenRepository;
private final PasswordEncoder passwordEncoder;
private final AuthenticationManager authenticationManager;
private final JwtService jwtService;
private final TokenBlacklistService tokenBlacklistService;
private final UserMapper userMapper;
@Value("${app.jwt.access-token-expiration}")
private long accessTokenExpirationMs;
@Value("${app.jwt.refresh-token-expiration}")
private long refreshTokenExpirationMs;
@Transactional
public AuthResponse register(RegisterRequest request) {
if (userRepository.existsByEmail(request.email())) {
throw new DuplicateResourceException("Email is already
registered: " + request.email());}
if (userRepository.existsByUsername(request.username())) {
throw new DuplicateResourceException("Username is already
taken: " + request.username());}
User user = User.builder().email(request.email())
.username(request.username())
.passwordHash(passwordEncoder.encode(request.password()))
.firstName(request.firstName())
.lastName(request.lastName())
.isActive(true).build();
user = userRepository.save(user);
log.info("Registered new user: email={}", user.getEmail());
return buildAuthResponse(user);}

@Transactional

public AuthResponse login(LoginRequest request
{authenticationManager.authenticate(
new UsernamePasswordAuthenticationToken(request.email(),
request.password()));
User user = userRepository.findByEmail(request.email())
.orElseThrow(() -> new ResourceNotFoundException("User",
request.email()));
log.info("User logged in: email={}", user.getEmail());
return buildAuthResponse(user);
}
@Transactional
public AuthResponse refresh(RefreshRequest request) {
String hash = hashToken(request.refreshToken());
RefreshToken stored =
refreshTokenRepository.findByTokenHash(hash)
.orElseThrow(() -> new ValidationException("Refresh token is
invalid or not found"));
if (stored.isRevoked()) {
throw new ValidationException("Refresh token has been
revoked");}
if (stored.getExpiresAt().isBefore(LocalDateTime.now())) {
throw new ValidationException("Refresh token has expired");}
stored.setRevoked(true);
refreshTokenRepository.save(stored);
User user = stored.getUser();
log.info("Token rotated for user: email={}", user.getEmail());
return buildAuthResponse(user);
}
@Transactional
public void logout(String accessToken, UUID userId) {
refreshTokenRepository.findByUserIdAndRevokedFalse(userId)
.forEach(token -> token.setRevoked(true));
String jti = jwtService.extractJti(accessToken);
Date expiry = jwtService.extractExpiration(accessToken);
long remainingTtlMs = expiry.getTime() -
System.currentTimeMillis();
tokenBlacklistService.blacklistToken(jti, remainingTtlMs);
log.info("User logged out: userId={}, jti={}", userId, jti);}
private AuthResponse buildAuthResponse(User user) {
String accessToken = jwtService.generateAccessToken(user);
String rawRefreshToken = jwtService.generateRefreshToken();
RefreshToken refreshToken = RefreshToken.builder().user(user)
.tokenHash(hashToken(rawRefreshToken))
.expiresAt(LocalDateTime.now().plusSeconds(refreshTokenExpi
ationMs / 1000)).revoked(false).build();
refreshTokenRepository.save(refreshToken);
return new AuthResponse(
accessToken,
rawRefreshToken,
"Bearer",
accessTokenExpirationMs / 1000,
userMapper.toSummaryResponse(user)
);}
private String hashToken(String token) {
```

```
try {
MessageDigest digest = MessageDigest.getInstance("SHA-256");
byte[] hash =
digest.digest(token.getBytes(StandardCharsets.UTF_8));

return HexFormat.of().formatHex(hash);}
 catch (NoSuchAlgorithmException e) {
throw new IllegalStateException("SHA-256 algorithm not
available", e);}}}

package com.gymtracker.workout.mapper;
import com.gymtracker.workout.domain.entity.ExerciseSet;
import com.gymtracker.workout.domain.entity.WorkoutExercise;
import com.gymtracker.workout.domain.entity.WorkoutSession;
import com.gymtracker.workout.dto.request.CreateSetRequest;
import
com.gymtracker.workout.dto.request.CreateWorkoutRequest;
import
com.gymtracker.workout.dto.response.ExerciseSetResponse;
import
com.gymtracker.workout.dto.response.WorkoutExerciseResponse;
import
com.gymtracker.workout.dto.response.WorkoutSessionResponse;
import
com.gymtracker.workout.dto.response.WorkoutSummaryRespons
e;
import org.mapstruct.Mapper;
import org.mapstruct.Mapping;
@Mapper(componentModel = "spring")
public interface WorkoutMapper {
WorkoutSessionResponse toResponse(WorkoutSession session);
WorkoutSummaryResponse
toSummaryResponse(WorkoutSession session);

@Mapping(target = "exerciseId", source = "exercise.id")
@Mapping(target = "exerciseName", source = "exercise.name")
@Mapping(target = "muscleGroup", source
="exercise.muscleGroup")
WorkoutExerciseResponse toExerciseResponse(WorkoutExercise
workoutExercise);
@Mapping(target = "setNumber", source = "number")
@Mapping(target = "setType", source = "type")
@Mapping(target = "isCompleted", expression =
"java(exerciseSet.isCompleted())")
ExerciseSetResponse toSetResponse(ExerciseSet exerciseSet);
@Mapping(target = "id", ignore = true)
@Mapping(target = "user", ignore = true)
@Mapping(target = "status", ignore = true)
@Mapping(target = "startedAt", ignore = true)
@Mapping(target = "completedAt", ignore = true)
@Mapping(target = "durationSeconds", ignore = true)
@Mapping(target = "exercises", ignore = true)
@Mapping(target = "templateId", ignore = true)
@Mapping(target = "createdAt", ignore = true)
@Mapping(target = "updatedAt", ignore = true)
WorkoutSession toEntity(CreateWorkoutRequest request);
@Mapping(target = "id", ignore = true)
@Mapping(target = "workoutExercise", ignore = true)
@Mapping(target = "isCompleted", ignore = true)
@Mapping(target = "createdAt", ignore = true)
@Mapping(target = "number", source = "setNumber")
@Mapping(target = "type", source = "setType")
ExerciseSet toSetEntity(CreateSetRequest request);
}

@Slf4j
@Component
@RequiredArgsConstructor
public class WorkoutEventConsumer {
private final StatisticsService statisticsService;
@KafkaListener(topics = "${app.kafka.topics.workout-
completed}", groupId = "${spring.kafka.consumer.group-id}")
public void
consumeWorkoutCompleted(WorkoutCompletedEvent event) {
log.info("Received WorkoutCompletedEvent for workoutId: {}",
event.workoutId());
try {
statisticsService.processWorkoutCompleted(event);
} catch (Exception e) {
log.error("Failed to process WorkoutCompletedEvent for
workoutId: {}. Skipping. Error: {}", event.workoutId(),
e.getMessage(), e);}}
}

package com.gymtracker.workout.security;
import org.junit.jupiter.api.BeforeEach;
 import org.junit.jupiter.api.Test;
 import org.junit.jupiter.api.extension.ExtendWith;
 import org.mockito.junit.jupiter.MockitoExtension;
import org.springframework.security.core.userdetails.User;
 import
org.springframework.security.core.userdetails.UserDetails;
 import java.util.Collections; import java.util.Date;
 import static org.assertj.core.api.Assertions.assertThat;
 @ExtendWith(MockitoExtension.class)
class JwtServiceTest {
 private JwtService jwtService; private final String testSecret =
"TestSecretKeyThatIsAtLeast32BytesLongForHS256Algorithm";
private final long testExpiration = 900000;
 @BeforeEach
 void setUp()
{ jwtService = new JwtService(testSecret, testExpiration); }
 @Test void
generateAccessToken_givenValidUser_returnsValidTokenWithCl
aims() { UserDetails user = new User("test@example.com",
"dummy", Collections.emptyList());
String token = jwtService.generateAccessToken(user);
 assertThat(token).isNotBlank(); assertThat(jwtService.extractEma
il(token)).isEqualTo("test@example.com");
assertThat(jwtService.extractJti(token)).isNotBlank();
assertThat(jwtService.extractExpiration(token)).isAfter(new
Date()); }
@Test void
validateToken_givenValidTokenAndMatchingUser_returnsTrue()
{ UserDetails user = new User("valid@example.com", "dummy",
Collections.emptyList());
 String token = jwtService.generateAccessToken(user); boolean
isValid = jwtService.validateToken(token, user);
```

```
 assertThat(isValid).isTrue(); }
 @Test void validateToken_givenMismatchedUser_returnsFalse()
{ UserDetails user1 = new User("user1@example.com",
"dummy", Collections.emptyList());
UserDetails user2 = new User("user2@example.com", "dummy",
Collections.emptyList());
 String token = jwtService.generateAccessToken(user1);
 boolean isValid = jwtService.validateToken(token, user2);
assertThat(isValid).isFalse(); }
 @Test void validateToken_givenTamperedToken_returnsFalse()
{ UserDetails user = new User("tampered@example.com",
"dummy", Collections.emptyList());
String token = jwtService.generateAccessToken(user);
String tamperedToken = token.replace("ey", "xx");
boolean isValid = jwtService.validateToken(tamperedToken,
user); assertThat(isValid).isFalse(); }
@Test void validateToken_givenExpiredToken_returnsFalse()
throws InterruptedException {
JwtService shortLivedService = new JwtService(testSecret,
1); UserDetails user = new User("expired@example.com",
"dummy",
Collections.emptyList()); String token =
shortLivedService.generateAccessToken(user);
Thread.sleep(10); boolean isValid =
shortLivedServiced.validateToken(token,
user); assertThat(isValid).isFalse(); }
@Test void generateRefreshToken_returnsPlainUuidString()
{ String token = jwtService.generateRefreshToken();
assertThat(token).isNotBlank();
assertThat(token.split("-")).hasSize(5); } }
```

```
package com.gymtracker.workout.repository;import
com.gymtracker.workout.domain.entity.Exercise;
import com.gymtracker.workout.domain.enums.MuscleGroup;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import java.util.Optional;
import java.util.UUID;

public interface ExerciseRepository extends
JpaRepository<Exercise, Long> {
Page<Exercise>
findAllByMuscleGroupAndIsCustomFalse(MuscleGroup
muscleGroup, Pageable pageable);
Page<Exercise> findAllByCreatedByIdOrIsCustomFalse(UUID
userId, Pageable pageable);
@Query("""
SELECT e FROM Exercise e
WHERE e.id = :id
AND (e.isCustom = false OR e.createdBy.id = :userId)
""")
Optional<Exercise> findByIdAccessibleTo(@Param("id") Long
id, @Param("userId") UUID userId);
@Query("""
SELECT e FROM Exercise e
WHERE (e.isCustom = false OR e.createdBy.id = :userId)
AND (:muscleGroup IS NULL OR e.muscleGroup
= :muscleGroup)
AND (:search IS NULL OR LOWER(e.name) LIKE
LOWER(CONCAT('%', :search, '%')))
ORDER BY e.isCustom ASC, e.name ASC
""")
Page<Exercise> findAccessibleExercises(
@Param("userId") UUID userId,
@Param("muscleGroup") MuscleGroup muscleGroup,
@Param("search") String search,
Pageable pageable);}
```

```
package com.gymtracker.workout.event;
import com.gymtracker.common.event.WorkoutCompletedEvent;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.stereotype.Component;
@Slf4j
@Component
@RequiredArgsConstructor
public class WorkoutEventPublisher {
private final KafkaTemplate<String, WorkoutCompletedEvent>
kafkaTemplate;
@Value("${app.kafka.topics.workout-completed:workout-
completed-events}")
private String workoutCompletedTopic;
public void publishWorkoutCompleted(WorkoutCompletedEvent
event) {
kafkaTemplate.send(workoutCompletedTopic,
event.userId().toString(), event).thenAccept(result -> log.info(
"Published WorkoutCompletedEvent: workoutId={}, userId={},
partition={}, offset={}",
event.workoutId(), event.userId(),
result.getRecordMetadata().partition(),
result.getRecordMetadata().offset()))
.exceptionally(ex -> {
log.error("Failed to publish WorkoutCompletedEvent
workoutId={}: {}",
event.workoutId(), ex.getMessage(), ex);
return null;});}
```