

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

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка застосунку для агрегації та відображення
корпоративного розкладу»

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

Андрій КУГАЇВСЬКИЙ

(nidnuc)

Виконав:

здобувач вищої освіти групи ПД-41

Андрій КУГАЇВСЬКИЙ

Керівник:

Віталій ЗАЛИВА

доктор філософії (PhD)

Рецензент:

**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**

Кугаївському Андрію Олександровичу

1. Тема кваліфікаційної роботи: «Розробка застосунку для агрегації та відображення корпоративного розкладу»

керівник кваліфікаційної роботи доктор філософії (PhD) Віталій ЗАЛИВА,
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: наукові публікації, технічна документація та відкриті джерела щодо розробки розподілених систем на основі мікросервісної архітектури, документація до фреймворків NestJS та React, специфікація роботи з СУБД PostgreSQL та Prisma ORM, опис побудови REST API з механізмами кешування (KeyDB), асинхронної комунікації через RabbitMQ та Bull, документація Telegram Bot API та Telegram Mini Apps, методи валідації даних ініціалізації Mini App через криптографічну перевірку підпису, приклади реалізації розподілених систем для забезпечення доступу до навчальної інформації.

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

1. Аналіз предметної галузі організації доступу до розкладу навчальних занять та визначення вимог до програмного забезпечення.
2. Огляд та аналіз засобів реалізації розподіленої системи для агрегації та відображення розкладу.
3. Проєктування мікросервісної архітектури застосунку для агрегації та відображення корпоративного розкладу.
4. Реалізація та тестування функціональності застосунку.

5. Перелік графічного матеріалу: *презентація*

1. Аналіз аналогів.
2. Вимоги до програмного забезпечення.
3. Програмні засоби реалізації.
4. Дизайн високого рівня.
5. ERD-діаграма бази даних.
6. Діаграма користувацького шляху.
7. Діаграма прецедентів.
8. Діаграма потоку даних.
9. Діаграма послідовності.
10. Екранні форми.
11. 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 - 04.03.2026	
2	Аналіз та дослідження існуючих аналогів	05.03 - 13.03.2026	
3	Аналіз предметної галузі організації доступу до розкладу та визначення вимог до ПЗ	14.03 - 19.03.2026	
4	Огляд та аналіз засобів реалізації розподіленої системи агрегації розкладу	20.03 - 28.03.2026	
5	Проєктування мікросервісної архітектури застосунку	29.03 - 13.04.2026	
6	Реалізація та тестування функціональності застосунку	14.04 - 28.04.2026	

7	Оформлення роботи: вступ, висновки, реферат	29.04 - 11.05.2026	
8	Розробка демонстраційних матеріалів	12.05 - 18.05.2026	
9	Попередній захист роботи	19.05 - 22.05.2026	
10	Подання кваліфікаційної роботи	23.05 - 29.05.2026	

Здобувач вищої освіти

(підпис)

Андрій КУГАЇВСЬКИЙ

Керівник
кваліфікаційної роботи

(підпис)

Віталій ЗАЛИВА

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 76 стор., 1 табл., 12 рис., 27 джерел.

Мета роботи - підвищення доступності та оперативності отримання розкладу занять для користувачів за рахунок розробки розподіленої системи з серверною агрегацією даних, кешуванням та інтеграцією з платформою Telegram.

Об'єкт дослідження - процес організації доступу студентів та викладачів до розкладу навчальних занять в умовах нестабільної доступності офіційних інформаційних систем університету.

Предмет дослідження - методи, моделі та програмні засоби розробки розподілених систем для агрегації, кешування та відображення даних розкладу через платформу Telegram.

Короткий зміст роботи: У роботі проаналізовано предметну область організації доступу до розкладу занять у закладах вищої освіти та проведено порівняльний огляд існуючих рішень - веб-системи МКР, офіційного мобільного додатку МКР та стороннього Android-додатку Skedy. Обґрунтовано необхідність створення альтернативного рішення із серверною агрегацією даних. Реалізовано мікросервісну систему, що складається з чотирьох компонентів: REST API на базі NestJS із адаптером Fastify, Telegram-бот на основі фреймворку grammY, клієнтський інтерфейс Telegram Mini App із використанням React та фоновий сервіс Loader для агрегації даних. Реляційне зберігання даних забезпечено СУБД PostgreSQL із Prisma ORM. Для кешування застосовано сховище KeyDB за стратегією cache-aside з адаптивним TTL. Асинхронну обробку завдань організовано за допомогою брокера повідомлень RabbitMQ та бібліотеки Bull із обмеженням швидкості відправки повідомлень. Реалізовано механізм щоденних сповіщень користувачів та систему виявлення змін у розкладі на основі бібліотеки deep-object-diff. Автентифікацію у Telegram Mini App забезпечено

криптографічною верифікацією ТМА-підпису. Для моніторингу системи інтегровано Sentry та стек Elasticsearch + Kibana. Розгортання виконано за допомогою Docker та Docker Compose. Система знаходиться у продуктивній експлуатації та обслуговує близько 1000 активних користувачів щоденно.

Сферою використання застосунку є система, призначена для доступу до розкладу навчальних занять у закладах вищої освіти (ЗВО).

КЛЮЧОВІ СЛОВА: BACK-END, FRONT-END, TYPESCRIPT, NODE.JS, NESTJS, TELEGRAM BOT, TELEGRAM MINI APP, REST API, POSTGRESQL, KEYDB, RABBITMQ, МІКРОСЕРВІСНА АРХІТЕКТУРА, КЕШУВАННЯ, РОЗКЛАД ЗАНЯТЬ.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ.....	10
ВСТУП.....	11
1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ	15
1.1 Організація доступу до розкладу у закладах вищої освіти та проблеми доступності	15
1.2 Аналіз існуючих програмних рішень.....	18
1.2.1 Веб-доступ до системи МКР.....	18
1.2.2 Офіційний мобільний додаток «Расписание занятий МКР»	20
1.2.3 Сторонній Android-додаток Skedy	21
1.3 Визначення вимог до застосунку.....	23
1.3.1 Функціональні вимоги.....	24
1.3.2 Нефункціональні вимоги.....	26
1.4 Інформаційна модель предметної галузі	29
2 ЗАСОБИ РЕАЛІЗАЦІЇ ЗАСТОСУНКУ	32
2.1 Середовище розробки.....	32
2.1.1 Visual Studio Code / Cursor	32
2.2 Мови програмування	33
2.2.1 TypeScript.....	33
2.2.2 SQL	34
2.3 Веб-фреймворки та бібліотеки	34
2.3.1 Node.js та NestJS.....	34
2.3.2 grammY.....	35
2.3.3 React та Vite	36
2.4 Система управління базами даних	38
2.4.1 PostgreSQL	38
2.4.2 Prisma.....	39
2.5 Кешування та сесії	39
2.5.1 KeyDB.....	39
2.6 Брокер повідомлень та черги	40
2.6.1 RabbitMQ.....	40
2.6.2 Bull	41
2.7 Моніторинг та спостережуваність.....	42
2.7.1 Sentry	42
2.7.2 Elasticsearch та Kibana.....	42
2.8 Контейнеризація та CI/CD.....	43
2.8.1 Docker	43

2.9 Система контролю версій	44
2.9.1 Git та GitHub	44
2.10 Інструменти дизайну інтерфейсу	45
2.10.1 Figma.....	45
3 ПРОЄКТУВАННЯ СИСТЕМИ.....	46
3.1 Проєктування архітектури застосунку.....	46
3.1.1 Діаграма діяльності користувача.....	49
3.1.2 Діаграма варіантів використання	51
3.2 Архітектура клієнтської частини (Telegram Mini App).....	56
3.3 Архітектура серверної частини (API + Bot + Loader).....	59
3.3.1 API Service (NestJS).....	59
3.3.2 Bot Service (grammY)	60
3.3.3 Loader Service	61
3.4 Архітектура бази даних	61
4 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ	62
4.1 Реалізація серверної частини (API)	62
4.1.1 TMA Authentication Middleware.....	62
4.1.2 Cache Interceptor	63
4.1.3 Schedule Service з логікою порівняння	63
4.1.4 Feedback Service з RabbitMQ RPC	64
4.1.5 User Service з управлінням ідентичністю	64
4.2 Реалізація клієнтської частини (Mini App).....	66
4.2.1 Потік ідентифікації	66
4.2.2 Сторінка розкладу	66
4.2.3 Сторінка налаштувань	67
4.2.4 Сторінка зворотного зв'язку.....	68
4.3 Реалізація Telegram-бота	69
4.3.1 Конфігурація Bull-черг	69
4.4 Реалізація Loader-сервісу	70
4.5 Інтеграція та взаємодія компонентів.....	72
4.6 Тестування застосунку.....	74
4.6.1 Функціональне тестування API	74
4.6.2 Тестування Telegram-бота.....	75
ВИСНОВКИ.....	76
ПЕРЕЛІК ПОСИЛАНЬ	79
ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ	81
ДОДАТОК Б. ЛІСТИНГИ ПРОГРАМНИХ МОДУЛІВ.....	91

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ПЗ - Програмне забезпечення

IDE - Integrated Development Environment

API - Application Programming Interface

REST - Representational State Transfer

SPA - Single Page Application

HTML - HyperText Markup Language

CSS - Cascading Style Sheets

JS - JavaScript

JWT - JSON Web Token

JSON - JavaScript Object Notation

SQL - Structured Query Language

ORM - Object-Relational Mapping

ВСТУП

Актуальність: починаючи з 2022 року, умови функціонування закладів вищої освіти в Україні зазнали кардинальних змін, зумовлених повномасштабним збройним вторгненням. Систематичні ракетні удари по об'єктах енергетичної інфраструктури призвели до масових відключень електроенергії, що безпосередньо вплинуло на стабільність роботи серверних потужностей університетів. Офіційна інформаційна система МКР, розгорнута на локальних серверах університету, виявилася вразливою до перебоїв електропостачання: у періоди відключень система ставала повністю недоступною, позбавляючи студентів та викладачів можливості ознайомитися з актуальним розкладом занять. Крім того, навіть у штатному режимі роботи система МКР демонструє значний час відгуку та обмежену функціональність мобільного інтерфейсу, що створює додаткові перешкоди для оперативного доступу до інформації.

Месенджер Telegram набув безпрецедентного поширення серед українських користувачів, досягнувши позначки понад 41 мільйон активних користувачів в Україні [25]. Для переважної більшості студентів Telegram є основним комунікаційним інструментом, який постійно активний на їхніх мобільних пристроях. Платформа Telegram надає розвинену екосистему для створення ботів [7] та міні-застосунків (Telegram Mini Apps [8]), що відкриває можливості для інтеграції освітніх сервісів безпосередньо у звичне комунікаційне середовище користувачів. Таким чином, розробка системи, яка агрегує дані розкладу із офіційних джерел, забезпечує їх кешування для гарантії доступності та надає зручний інтерфейс через Telegram, є актуальною задачею, що відповідає реальним потребам академічної спільноти.

Додатковим фактором актуальності є відсутність серед існуючих аналогів рішення, яке б комплексно поєднувало серверну агрегацію даних із множини джерел, інтелектуальне кешування з автоматичним виявленням змін, проактивне

сповіщення користувачів про оновлення розкладу та повноцінний графічний інтерфейс у форматі Telegram Mini App.

Об'єктом дослідження є процес організації доступу студентів та викладачів до розкладу навчальних занять в умовах нестабільної доступності офіційних інформаційних систем університету.

Предметом дослідження є методи, моделі та програмні засоби розробки розподілених систем для агрегації, кешування та відображення даних розкладу через платформу Telegram.

Метою роботи є підвищення доступності та оперативності отримання розкладу занять за рахунок розробки розподіленої мікросервісної системи з серверною агрегацією даних, кешуванням та інтеграцією з платформою Telegram.

Методи дослідження першим етапом дослідження стало ознайомлення з предметною областю організації доступу до розкладу у ЗВО та аналіз існуючих рішень (веб-система МКР, мобільний додаток МКР, сторонній додаток Skedy). Це дозволило сформулювати функціональні та нефункціональні вимоги до майбутнього застосунку. Далі було проведено огляд стеку технологій, які відповідають визначеним вимогам. Для реалізації серверної частини обрано фреймворк NestJS із адаптером Fastify та мову програмування TypeScript. Для Telegram-бота використано фреймворк grammY. Для клієнтської частини обрано бібліотеку React зі збірником Vite. PostgreSQL із Prisma ORM обрано як систему управління базами даних. KeyDB використовується як кеш-сховище за стратегією cache-aside. Асинхронну міжсервісну комунікацію реалізовано через RabbitMQ, а контрольовану доставку повідомлень - через Bull-черги. Автентифікацію Telegram Mini App реалізовано на основі криптографічної валідації initData. Архітектуру програмного забезпечення спроектовано з використанням UML-діаграм. Розгортання здійснюється через Docker та Docker Compose.

Наукова новизна роботи полягає у розробці комплексного підходу до побудови розподіленої системи агрегації розкладу занять, який поєднує стратегію кешування cache-aside із використанням KeyDB та адаптивним TTL, алгоритм виявлення змін на основі deep-object-diff із транзакційним оновленням бази даних,

механізм обмеження інтенсивності запитів через Bull-черги для дотримання лімітів Telegram Bot API, криптографічну автентифікацію ТМА з верифікацією НМАС-підпису, систему двомовної локалізації на основі Fluent та i18next та ієрархічну каскадну стратегію оновлення даних.

Практична значущість результатів підтверджується успішним впровадженням розробленої системи у реальну експлуатацію. Станом на момент написання кваліфікаційної роботи система обслуговує близько 1000 активних користувачів щоденно (DAU), що свідчить про її затребуваність серед студентів та викладачів Державного університету інформаційно-комунікаційних технологій. Розроблений Telegram-бот та Mini App забезпечують безперебійний доступ до актуального розкладу занять незалежно від стану офіційних інформаційних систем університету, що є особливо важливим в умовах нестабільного енергопостачання.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Організація доступу до розкладу у закладах вищої освіти та проблеми доступності

Одним із основних джерел інформації для вищого навчального закладу (ВНЗ) є його розклад занять. Цей розклад визначає хід навчального процесу не тільки для студентів, а й для викладачів та адміністрації. Весь навчальний процес у ЗВО від початку до кінця відбувається згідно з розкладом протягом тижнів семестру. Наразі розклад занять у навчальних закладах здебільшого доступний через централізовані інформаційні системи, найпопулярнішою з яких є система МКР [27].

У системі МКР розклади занять формуються за рівнями, що визначаються організаційною структурою університету. Починаючи з найвищого рівня, підрозділ, наприклад інститут, поділяється на відповідні факультети. У межах кожного факультету виокремлюються різні курси навчання, що утворюють нижчі рівні. Нижче за ними програми поділяються на менші групи, відомі як академічні групи. Саме в цих групах формуються та застосовуються фактичні плани занять. Студенти проходять такий шлях: підрозділ, потім факультет, далі рік навчання, і наприкінці - група. Для викладацького складу маршрут дещо змінюється. Та сама початкова одиниця веде до кафедр, а не до факультетів. Кожна кафедра включає окремих викладачів, які отримують свої розклади. Їхній ланцюжок простягається від одиниці до кафедри, зрештою досягаючи викладача. Окрема кафедра вписується у визначений структурний сегмент, об'єднуючи викладачів, які працюють в одній галузі. Звідти, щоб дізнатися, коли конкретний студент відвідує заняття, потрібно крок за кроком пройти чотири вбудовані рівні, для викладача це лише три. Рівні розгортаються поступово, формуючись залежно від того, як одиниці розподіляють обов'язки.

Одна з головних проблем, пов'язаних із централізованими онлайн-платформами, полягає в тому, наскільки вони залежать від безперебійного доступу

до фізичних серверів. Коли конфлікт в Україні розгорівся на повну силу, цю вразливість стало все важче ігнорувати. Атаки з використанням ракет і дронів часто вражають об'єкти енергетики, порушуючи електропостачання.

Електропостачання може бути відсутнім протягом періодів, що тривають від кількох годин до кількох днів. У такі моменти доступ зникає, коли університетські комп'ютери відключаються від мережі. Разом із цим зникає й система розкладу. Без оновлень стає неможливо дізнатися, які лекції відбудуться завтра. Такі деталі, як час початку, номери аудиторій або зміни порівняно з попередніми планами, не доходять до користувачів. Інформаційні прогалини розширюються, поки мережі залишаються недоступними. Це призводить до відчутних наслідків: можуть траплятися пропущені лекції, недоступні години консультацій або відсутність на іспитах, а викладацький склад стикається з труднощами в організації щоденної роботи. Хоча система МКР стикалася з періодичними перебоями або плановими оновленнями в стабільні періоди - короткочасно блокуючи доступ до розкладу - у нинішніх умовах військового стану ці перебої трапляються частіше і тривають довше. Те, що колись здавалося рідкісним, тепер відбувається дедалі частіше.

Саме так, виникає потреба у збиранні розкладів - не просто показувати веб-сторінки такими, якими вони є, а натомість повністю обробляти дані. Інформація надходить з офіційних джерел, упорядковується у єдиний формат, зберігається у приватній базі даних, а потім надається користувачам незалежно від того, чи працює джерело. На відміну від перенаправлення трафіку через дзеркало, цей метод створює ізоляцію - автономний резервуар, здатний існувати без зовнішнього введення даних. Хоча прямі трансляції залишаються доступними, оновлення часто оновлюють збережену версію, зміни в часі реєструються і відповідно досягають одержувачів. Важливе значення має й обізнаність - користувачі дізнаються, чи те, що вони бачать, надходить безпосередньо зі сховища після втрати зв'язку, чи було закешовано за мить до цього. Тривалість відповідає наміру протягом усього процесу. Актуальність залишається видимою завжди.

Не менш важливим, ніж будь-який дизайнерський вибір, є те, як розклад доходить до користувачів. Згідно з даними Київського міжнародного інституту

соціології (КМІС [25]), на перший план виходить один інструмент - месенджер Telegram [25], який регулярно охоплює понад 41 мільйон українців. Серед студентів цей простір виходить за межі обміну повідомленнями, тут створюються навчальні групи, відбувається обмін файлами, проводяться уроки та дискусії з однокурсниками та викладачами. Замість того, щоб очікувати встановлення ще одного окремого додатка, набагато природніше зустріти їх там, де вони вже знаходяться. У Telegram таку інтеграцію підтримують два шляхи - автоматизовані помічники через Bot API [7], а також Mini Apps [8] - компактні програми, що працюють безпосередньо у вікні чату, повністю оминаючи зовнішні браузері.

Сучасні системи розкладів вимагають не лише відображення інформації. На першому плані - персоналізація, кожна особа самостійно обирає, чи є вона студентом чи співробітником. Далі відбувається налаштування на основі ролей, яке визначає вибір занять для студентів та викладачів важливими є кафедри та прізвища. Вибір мови з'являється як опція поряд із сповіщеннями, що налаштовуються індивідуально. Автоматизація працює непомітно - розклади з'являються ще до світанку, надсилаються без нагадувань. Щоразу, коли в розкладі відбуваються зміни, оновлення миттєво доходять до користувачів. Не потрібно чекати, не потрібно повторно входити в систему протягом тижня. Адміністративні функції працюють за лаштунками - менеджери розсилають повідомлення, спрямовані на конкретні групи. Опитування з'являються, коли необхідна інформація, і спрямовані лише на відповідних учасників. Відгуки надходять захищеними каналами, що дозволяє відповідати, не розкриваючи себе. Кожен елемент залишається пов'язаним, але працює незалежно, коли це потрібно.

Час від часу нові дані автоматично збираються з джерел у рамках системи консолідації розкладів. Після отримання даних перед їх збереженням проводиться перевірка точності. Цілісність даних під час зберігання забезпечується за допомогою механізмів транзакційної безпеки у структурованих сховищах даних. Часті запити отримують результати з тимчасових високошвидкісних шарів, а не з основних таблиць. Доступ для споживачів здійснюється через інтерфейси, такі як кінцеві точки REST, автоматизовані месенджери або компактні додатки. Стан

системи, затримки у відповідях та журнали помилок перебувають під постійним наглядом. Коли у розкладах з'являються зміни ззовні, внутрішні версії відповідно коригуються. Кожна фаза працює з окремими інструментами, що формує особливі вимоги до структури дизайну.

З цієї причини проблеми з доступом до розкладів занять в українських університетах виходять за межі звичайних збоїв у роботі серверів. До них належать затримки в оновленні інформації, жорсткі формати, що не відповідають індивідуальним потребам, обмежені можливості отримання детальної інформації, а також мінімальні можливості для зворотного зв'язку з користувачами. Для належного вирішення таких завдань потрібен інтегрований набір мікросервісів, що об'єднує потоки даних у режимі реального часу, гнучкі канали передачі інформації, індивідуальне формування контенту та безперебійну роботу - і все це через найпопулярніший в країні месенджер.

1.2 Аналіз існуючих програмних рішень

На момент проведення дослідження ринок програмних рішень для доступу до розкладу навчальних занять у закладах вищої освіти України представлений спектром продуктів різного рівня зрілості та функціональності. Ці рішення можна класифікувати за типом: офіційні веб-системи, створені розробниками самої платформи МКР, офіційні мобільні додатки, що є клієнтськими інтерфейсами до тих самих систем а також сторонні додатки, розроблені незалежними командами для конкретних університетів або групи університетів. Кожна категорія має свої характерні переваги та обмеження, аналіз яких є необхідним для формулювання вимог до нової системи.

1.2.1 Веб-доступ до системи МКР

МКР є основним і єдиним офіційним інструментом для перегляду розкладу занять у низці українських вищих навчальних закладів. Зокрема, МКР надає користувачам можливість переглянути розклад занять для певної навчальної групи

або викладача на певний тиждень через веб-інтерфейс. Щоб сформувати розклад, необхідно послідовно вибрати кілька параметрів із випадających списків: а саме, вибрати структурний підрозділ, кафедру, курс та навчальну групу або кафедру й прізвище відповідно. Після введення всієї необхідної інформації система формує розклад у форматі HTML, у якому дні тижня представлені у вигляді стовпців, а час - у вигляді рядків.

Однією з головних переваг системи МКР є її загальна доступність - для перегляду розкладу не потрібно додаткового програмного забезпечення, оскільки від користувача вимагається лише сучасний веб-браузер, який можна використовувати на будь-якому пристрої, від ПК до планшета чи смартфона. Ще однією перевагою використання системи МКР є те, що вона є офіційним джерелом даних, що гарантує оперативність та надійність наданої інформації. Доступ до веб-версії сервісу не вимагає входу в систему, а сам сервіс безпосередньо підтримується розробниками МКР.

З іншого боку, у реалізації є низка недоліків, які впливають на корисність програми як функціонального інструменту. Найголовнішим недоліком є повна залежність програми від серверів. Наприклад, у разі відключення електроенергії, технічних проблем або планових робіт з технічного обслуговування університетських серверів користувач взагалі не зможе отримати доступ до свого розкладу. Крім того, немає можливості кешування інформації на стороні клієнта, що означає: навіть якщо користувач нещодавно переглядав свій розклад, йому все одно доведеться підключатися до сервера, щоб знову отримати цю інформацію.

Система МКР має чимало недоліків з точки зору мобільного досвіду. Веб-додаток не повністю адаптований до обмежень смартфонів. Зокрема, розклад не відображається повністю на одному екрані (його доводиться прокручувати по горизонталі), випадające меню можуть бути незручними під час вибору елементів, нарешті, загалом у ньому забагато тексту, розрахованого на перегляд на звичайному настільному комп'ютері. Крім того, немає офлайн-режиму, немає можливості збереження налаштувань з попередніх сесій (наприклад, користувачеві доведеться щоразу заново обирати свою групу або прізвище), а також немає

можливості налаштувати інтерфейс. Немає можливості надати розробникам відгук щодо будь-яких помилок або пропозицій щодо вдосконалення.

Отже, хоча веб-інтерфейс до бази даних МКР є офіційно визнаним основним джерелом інформації, він не відповідає сучасним стандартам щодо зручності користування, доступності та інформування користувачів.

1.2.2 Офіційний мобільний додаток «Расписание занятий МКР»

Офіційний додаток «Расписание занятий МКР» - це мобільний клієнт для систем Android та iOS, розроблений розробниками системи МКР. Додаток дозволяє користувачам переглядати розклади занять на мобільних пристроях. Додаток охоплює понад 15 вищих навчальних закладів України, надаючи користувачам універсальний доступ до розкладів занять різних навчальних закладів.

Однією з сильних сторін програми є її мобільний інтерфейс, розроблений спеціально для мобільних пристроїв, який дозволяє максимально ефективно використовувати сенсорне управління та невеликі розміри екрану. Список класів відображається у вертикальному форматі, що зручніше для перегляду на смартфоні, ніж горизонтальне розміщення, яке використовується у веб-версії програми. Можливість зберігати свої налаштування підвищує ефективність щоденного використання.

Найбільшою проблемою є те, що додаток не оновлювався з вересня 2023 року, що свідчить про те, що робота над ним вже деякий час не ведеться. Це важливо, оскільки з того часу відбулися значні зміни як в операційних системах Android та iOS, так і в умовах роботи магазинів додатків. Крім того, жодна з помилок, про які повідомляли користувачі, з того часу не була виправлена.

Функціонал додатка передбачає дві основні опції перегляду розкладу - список занять на конкретний день та календарний вигляд, що відображає заняття на певні дні. Спочатку користувач обирає свій навчальний заклад зі списку доступних університетів, а потім вибір критеріїв здійснюється аналогічно до веб-версії - структурний підрозділ, факультет, курс, група, кафедра та викладач. Обрані

користувачем параметри зберігаються в пам'яті додатка і не потребують повторного налаштування при кожному запуску програми.

Проте, у цього додатка є низка серйозних недоліків, які значно знижують його корисність. По-перше, цей додаток потрібно встановлювати, це на крок складніше, ніж користування веб-браузерами чи ботами в месенджерах, де все починається з простого кліка. У відгуках користувачів є багато скарг на проблеми з авторизацією та стабільністю: проблеми із завантаженням розкладу, проблеми зі зміною налаштувань та помилки під час перемикання режимів. Рейтинг цього додатка становить у середньому лише 2,7 з п'яти зірок, що є дуже низьким показником для продуктивного інструменту.

Технічно додаток не виконує жодної агрегації даних на стороні сервера, а діє як мобільний клієнт, що отримує розклад безпосередньо з серверів. Отже, додаток стає настільки ж залежним від сервера, як і веб-додаток. Додаток не надає щоденних сповіщень про розклад на наступний день, не має можливості відстежувати будь-які зміни в розкладі та не пропонує жодного зворотного зв'язку для користувачів та управління розкладом. Підтримка понад 15 університетів, що рекламується як одна з функцій, на практиці перетворюється на нестабільну роботу через відмінності в характері різних МКР та їхніх специфічних вимог.

1.2.3 Сторонній Android-додаток Skedy

Skedy - це незалежний додаток для Android, створений незалежним розробником під ніком YaCode для перегляду розкладів занять у вищих навчальних закладах, які використовують систему МКР. Аналіз файлу APK дозволяє отримати повне уявлення про технологічний стек та архітектуру додатка.

Додаток розроблено на основі сучасного стеку технологій для Android: мова програмування Kotlin версії 1.9.25, фреймворк декларативного інтерфейсу користувача Jetpack Compose версії 1.6.8, база даних Room версії 2.6.1, платформа хмарних сервісів Firebase, фреймворк ін'єкції залежностей Hilt версії 2.51.1, менеджер фонових завдань WorkManager та HTTP-клієнт OkHttp3. Цей набір технологій відповідає рекомендаціям Google щодо розробки додатків для Android.

Отже, розподіл компонентів Activity на окремі класи в додатку Skedy здійснюється на основі такого набору елементів: LoginActivity - для входу в систему та вибору університету, MainActivity - для відображення екрана розкладу, SettingsActivity - для екрана налаштувань, IntroActivity - для екрана привітання для нових користувачів, AboutActivity - для опису додатка та автора, AudiencesActivity - стосовно цікавої функції додатка, яка була

У додатку передбачено два основні види відображення розкладу: щоденний (з показом занять на конкретний день) та тижневий (з оглядом усього тижня). Вигляд тижневого розкладу базується на горизонтальній прокрутці та масштабуванні, що дозволяє користувачам швидко переглядати насичений розклад на екранах своїх смартфонів. Повідомлення надсилаються за допомогою Firebase Cloud Messaging у поєднанні з WorkManager.

Додаток підтримує дві мови інтерфейсу: українську та російську. Оновлення додатка здійснюються через вбудовану функцію оновлення, що дозволяє швидко доставляти нові версії додатка користувачам.

Використання локальної бази даних Room є однією з архітектурних переваг, користувач може переглядати будь-які раніше завантажені дані навіть без підключення до Інтернету, оскільки розклад зберігається локально. Крім того, Jetpack Compose дозволяє створювати сучасний та динамічний інтерфейс користувача, що відповідає стандарту Material Design 3. Підтримка декількох університетів розширює цільову аудиторію, оскільки додаток доступний для всіх користувачів системи МКР.

З іншого боку, Skedy також має низку недоліків. Перший з них - залежність від платформи, оскільки додаток доступний лише для користувачів з пристроями на базі Android і не може використовуватися тими, хто має мобільні телефони на базі iOS (а вони становлять значну частину аудиторії). Аналіз маніфесту додатка показує, що в ньому присутній Firebase AD_ID (ідентифікатор реклами), що означає, що ідентифікатор реклами пристрою користувача збирається.

З архітектурної точки зору, навіть якщо додаток використовує локальну базу даних Room, Skedy не забезпечує повного агрегування даних на стороні сервера.

Room лише кешує дані локально на пристрої користувача, тоді як агрегація на стороні сервера передбачає наявність спеціального серверного модуля, який агрегує незалежно нормалізовані дані з основного джерела та робить їх доступними для всіх інших клієнтів одночасно. У випадку Skedy кожен пристрій користувача окремо отримує розклад від серверів МКР, але це не вирішує проблему доступності у випадку, якщо основне джерело стає недоступним для нових запитів або нових користувачів.

У додатку немає можливості надсилати сповіщення про розклад на завтра без ручної перевірки додатка самим користувачем. В інтерфейсі додатка немає можливості зворотного зв'язку, щоб безпосередньо повідомити адміністраторів розкладу або розробників Skedy про проблеми. Також немає функції розпізнавання змін у розкладі та сповіщення користувачів про ці зміни з необхідною якістю.

Порівняльний аналіз трьох згаданих існуючих рішень - веб-додатку МКР, офіційного мобільного додатку та стороннього додатку Skedy - виявляє недоліки в кожній із цих систем, які унеможливають їх використання як повноцінного та комплексного рішення проблеми. Першим недоліком, притаманним усім трьом рішенням, є їхня залежність від серверів МКР як єдиного джерела інформації без реалізації повноцінної серверної агрегації даних з окремим сховищем. Другою проблемою цих додатків є відсутність функції автоматичного щоденного нагадування з можливістю налаштування часу доставки. Третьою проблемою додатків є їхня нездатність відстежувати зміни на стороні сервера з негайним повідомленням усіх користувачів, яких це стосується. Нарешті, додаткам бракує інтеграції з найпоширенішою в Україні комунікаційною платформою - Telegram.

1.3 Визначення вимог до застосунку

На основі аналізу предметної галузі та існуючих програмних рішень було сформульовано набір функціональних та нефункціональних вимог до системи ТТА ДУІКТ. Функціональні вимоги визначають конкретну поведінку системи, яку

очікує користувач, тоді як нефункціональні вимоги описують якісні характеристики системи в цілому.

1.3.1 Функціональні вимоги

FR-01: Ідентифікація користувача як студента або викладача з навігацією за ієрархією. Система повинна надавати можливість користувачу обрати свою роль - студент або викладач - та послідовно визначити свою позицію в організаційній ієрархії університету. Для студента це передбачає вибір структурного підрозділу, факультету, курсу та академічної групи. Для викладача - вибір структурного підрозділу, кафедри та свого прізвища зі списку викладачів кафедри. Обрані параметри зберігаються у профілі користувача та використовуються для подальшого отримання розкладу без повторного вибору.

FR-02: Перегляд розкладу за діапазон дат із обмеженням до 31 дня. Система повинна дозволяти перегляд розкладу на довільний діапазон дат, обмежений максимальною тривалістю в 31 день. Розклад відображається у структурованому вигляді з групуванням за днями, із зазначенням часу проведення, назви дисципліни, типу заняття (лекція, практична, лабораторна тощо), прізвища викладача (для студентського розкладу) або назви групи (для викладацького розкладу), номера аудиторії та будівлі.

FR-03: Серверна агрегація з резервним доступом через кеш та базу даних. Система повинна реалізувати триступеневу стратегію отримання даних: у першу чергу дані запитуються з кешу, у разі відсутності в кеші - з бази даних, і лише за відсутності даних у обох сховищах - безпосередньо з першоджерела. Це забезпечує доступність розкладу навіть при тривалій недоступності серверів МКР.

FR-04: Прозорість джерела даних через заголовок X-Data-Status. Кожна відповідь API, що містить дані розкладу, повинна супроводжуватися HTTP-заголовком X-Data-Status, значення якого вказує на джерело отриманих даних: *cached* (дані отримані з кешу), *fresh* (дані отримані безпосередньо з першоджерела або бази даних). Це дозволяє клієнтським додаткам інформувати користувача про актуальність наданих даних.

FR-05: Щоденні сповіщення за розкладом. Система повинна автоматично надсилати користувачам розклад занять на наступний навчальний день. Сповіщення надсилаються за розкладом з неділі по четвер о 15:00 за київським часом та у неділю об 15:00 з розкладом на понеділок. Користувач повинен мати можливість увімкнути або вимкнути цю функцію у своїх налаштуваннях.

FR-06: Виявлення змін у розкладі через `deer-object-diff` [23] із повідомленням через `RabbitMQ` та `Bull`. Система повинна автоматично порівнювати новозавантажений розклад із раніше збереженим за допомогою бібліотеки `deer-object-diff` та, у разі виявлення розбіжностей, формувати повідомлення про зміни та направляти його до черги повідомлень `RabbitMQ` для подальшої обробки та доставки користувачам через `Bull`.

FR-07: Не менше 8 параметрів персоналізації з синхронізацією через `RabbitMQ`. Система повинна надавати користувачу можливість налаштувати щонайменше 8 параметрів, серед яких: увімкнення/вимкнення щоденних сповіщень, сповіщення про зміни у розкладі, мова інтерфейсу, відображення номера пари, відображення часу заняття, відображення типу заняття та інші. Зміна налаштувань у `Mini App` повинна синхронізуватися з ботом через `RabbitMQ` для забезпечення консистентності.

FR-08: Зворотний зв'язок через захищений канал `RabbitMQ`. Система повинна надавати користувачам можливість надсилати зворотний зв'язок (скарги, пропозиції, повідомлення про помилки) через інтерфейс `Mini App`. Повідомлення зворотного зв'язку передаються через `durable` чергу `RabbitMQ` для гарантії доставки та обробляються ботом для сповіщення адміністраторів.

FR-09: Адміністративні оголошення та опитування з таргетуванням аудиторії. Адміністратори системи повинні мати можливість надсилати текстові оголошення та опитування обраній аудиторії: усім користувачам, лише студентам, лише викладачам або конкретним групам. Розсилка повинна здійснюватися через чергу `Bull` з обмеженням швидкості для запобігання блокуванню з боку `Telegram API`.

FR-10: Багатомовність через Fluent та i18next. Система повинна підтримувати щонайменше дві мови інтерфейсу: українську (за замовчуванням) та англійську. Бот використовує формат локалізації Fluent [22] (.ftl файли), а Mini App - бібліотеку i18next [21]. Мова інтерфейсу обирається користувачем у налаштуваннях та зберігається у його профілі.

1.3.2 Нефункціональні вимоги

Продуктивність. Час відповіді API для запитів із кешу не повинен перевищувати 200 мс. Кешування з TTL 30 хвилин для даних розкладу, 5 хвилин для порожніх відповідей та 5 секунд у середовищі розробки забезпечує оптимальний баланс між актуальністю даних та навантаженням на сервер. Обмеження швидкості відправлення повідомлень 25 повідомлень за 2.5 секунди запобігає блокуванню ботом з боку Telegram API.

Надійність. Використання durable черг RabbitMQ гарантує збереження повідомлень навіть у разі перезапуску брокера. Транзакційне оновлення даних через Prisma \$transaction забезпечує атомарність операцій із базою даних. Автоматичне відключення сповіщень при отриманні HTTP-статусів 403 або 400 від Telegram API запобігає повторним невдалим спробам надсилання повідомлень заблокованим ботам. Експоненціальний backoff при повторних спробах обробки задач у Bull зменшує навантаження на систему при тимчасових збоях.

Масштабованість. Мікросервісна архітектура з чотирма незалежними сервісами дозволяє масштабувати кожен компонент окремо відповідно до його навантаження. Використання черг повідомлень як механізму міжсервісної комунікації забезпечує слабку зв'язність між компонентами та можливість горизонтального масштабування обробників.

Безпека. Автентифікація користувачів Mini App здійснюється через криптографічну валідацію initData за допомогою бібліотеки @tma.js/init-data-node, що виключає можливість підробки ідентичності користувача. API-ендпоінти, що потребують автентифікації, захищені відповідними guards, які перевіряють валідність підпису ініціалізаційних даних Telegram Mini App.

Спостережуваність. Усі чотири мікросервіси інтегровані з платформою Sentry [18] для відстеження помилок у реальному часі. Elasticsearch [16] із Kibana [17] забезпечують збір, зберігання та візуалізацію логів за двома індексами: `incoming_updates` (вхідні повідомлення від користувачів до бота) та `outgoing_updates` (вихідні повідомлення від бота до користувачів), що дозволяє відстежувати весь цикл взаємодії.

Розгортання. Система розгортається через Docker [19] із використанням `multi-stage builds` на базі образу `node:22-alpine`, що забезпечує мінімальний розмір контейнерів. Резервне копіювання бази даних здійснюється автоматично кожні 12 годин із збереженням у S3-сумісне сховище. Docker Compose [19] у продукційній конфігурації об'єднує всі сервіси та допоміжну інфраструктуру в єдиний стек розгортання.

Зручність використання. Інтерфейс Mini App побудований з використанням компонентів Telegram UI, що забезпечує візуальну консистентність із основним інтерфейсом месенджера Telegram. Підтримка темної та світлої тем оформлення, адаптивних шрифтів та нативних жестів навігації створює відчуття інтегрованості з платформою.

Доступність. Триступенева стратегія доступу до даних (кеш -> база даних -> першоджерело) забезпечує максимальну доступність розкладу навіть при частковій деградації інфраструктури.

На таблиці 1.1 видно що система ТТА DUIKT є єдиним рішенням, яка реалізує серверну агрегацію даних, щоденні нагадування, виявлення змін у розкладі, прозорість джерела даних, повноцінну персоналізацію, зворотний зв'язок та адміністративні функції одночасно. При цьому система не потребує встановлення окремого додатку та доступна на всіх платформах, де встановлений Telegram.

Таблиця 1.1

Порівняльна таблиця існуючих рішень та ТТА DUKT

Критерій	Веб МКР	Додаток МКР	Skedy	ТТА DUKT
Серверна агрегація даних	-	-	-	+
Офлайн-доступ	-	-	+	+
Push-сповіщення	-	-	+	+
Щоденні нагадування	-	-	-	+
Виявлення змін у розкладі	-	-	-	+
Прозорість джерела даних	-	-	-	+
Кросплатформність	+	+	-	+
Необхідність встановлення	-	+	+	-
Персоналізація	-	-	-	+
Багатомовність	-	-	+	+
Зворотний зв'язок	-	-	-	+
Адміністративні оголошення	-	-	-	+
Моніторинг та логування	-	-	+	+
Актуальність підтримки	+	-	+	+

1.4 Інформаційна модель предметної галузі

Інформаційна модель системи ТТА DUIKT побудована на основі реляційної бази даних PostgreSQL та описана за допомогою Prisma ORM. Модель включає 10 основних сутностей та 2 перелічувані типи, що відображають структуру предметної галузі - організаційну ієрархію університету, профілі користувачів та дані розкладу. Схема бази даних пройшла 6 міграцій протягом розробки, кожна з яких фіксує еволюцію моделі даних від початкової структури до поточного стану.

Центральною сутністю моделі є Setting - модель профілю та налаштувань користувача. Первинним ключем моделі є поле chatId типу BigInt, що відповідає унікальному ідентифікатору Telegram-чату користувача. Використання chatId як первинного ключа є обґрунтованим архітектурним рішенням, оскільки кожен користувач Telegram має гарантовано унікальний ідентифікатор, який є стабільним протягом усього часу існування облікового запису. Модель містить поле type, яке приймає значення перелічуваного типу Type із двома варіантами: STUDENT (студент) та TEACHER (викладач). Поле language має тип перелічуваного типу LANGUAGE із варіантами UK (українська, значення за замовчуванням) та EN (англійська).

Крім того, модель включає щонайменше 10 бульових полів, що відповідають окремим параметрам персоналізації: увімкнення щоденних сповіщень, сповіщення про зміни розкладу, відображення номера пари, відображення часу заняття, відображення типу заняття, відображення викладача/групи, відображення аудиторії та інші. Кожне бульове поле має значення за замовчуванням, що забезпечує готовність до використання одразу після реєстрації користувача.

Модель Structure представляє структурний підрозділ університету (наприклад, навчально-науковий інститут). Кожен структурний підрозділ має унікальний ідентифікатор, отриманий із системи МКР, та назву. Ця модель є кореневим елементом ієрархії як для студентської, так і для викладацької гілок.

Модель Faculty представляє факультет або відділення в межах структурного підрозділу. Факультет пов'язаний зі структурним підрозділом через зовнішній

ключ, що формує відношення «один до багатьох» (один структурний підрозділ містить декілька факультетів). Кожен факультет має унікальний ідентифікатор та назву.

Модель Course представляє курс навчання в межах конкретного факультету. Особливістю цієї моделі є складений унікальний ключ, що складається з полів `courseId` та `facultyId`. Це рішення обумовлене тим, що ідентифікатор курсу не є глобально унікальним - він повторюється для кожного факультету. Лише комбінація курсу та факультету однозначно ідентифікує конкретний курс конкретного факультету.

Модель Group представляє академічну групу в межах конкретного курсу. Група є кінцевим елементом студентської ієрархії та основною одиницею, для якої формується розклад занять. Кожна група має унікальний ідентифікатор та назву.

Модель Chair представляє кафедру в межах структурного підрозділу. Кафедра пов'язана зі структурним підрозділом через зовнішній ключ, аналогічно до факультету, формуючи відношення «один до багатьох».

Модель ChairTeacher представляє зв'язок між кафедрою та викладачем. Ця модель реалізує частину ієрархії для викладацької гілки, зберігаючи інформацію про належність конкретного викладача до конкретної кафедри.

Модель Student зберігає зв'язок між профілем користувача (Setting) та обраною академічною групою. Ця модель фіксує ідентичність користувача-студента в системі, дозволяючи системі автоматично визначати, розклад якої групи необхідно відображати або надсилати у сповіщенні.

Модель Teacher аналогічно зберігає зв'язок між профілем користувача (Setting) та обраним викладачем (ChairTeacher). Ця модель фіксує ідентичність користувача-викладача в системі.

Модель Schedule є найбільш об'ємною моделлю в системі та безпосередньо зберігає дані розкладу занять. Кожен запис у моделі відповідає одному заняттю на конкретну дату. Модель містить поля для дати проведення, номера пари (часового слоту), часу початку та завершення, назви дисципліни, типу заняття, прізвища викладача, назви групи, номера аудиторії та назви будівлі. Критичною особливістю

моделі є наявність двох опціональних зовнішніх ключів: `group_id` (посилання на модель `Group`) та `teacher_id` (посилання на модель `ChairTeacher`). Опціональність обох ключів дозволяє зберігати розклад як для групи (коли заповнений `group_id`), так і для викладача (коли заповнений `teacher_id`), а також обробляти випадки, коли відповідний об'єкт ще не створений у базі даних.

Два перелічувані типи доповнюють модель. `Type` визначає роль користувача в системі та має два значення: `STUDENT` та `TEACHER`. Цей тип використовується в моделі `Setting` для фіксації обраної ролі. `LANGUAGE` визначає мову інтерфейсу та має два значення: `UK` (українська) та `EN` (англійська). Значенням за замовчуванням є `UK`, що відповідає тому факту, що переважна більшість користувачів системи є україномовними.

Зв'язки між моделями формують чітку ієрархічну структуру, що відображає організаційну будову університету. Студентська гілка: `Structure` -> (1:N) -> `Faculty` -> (1:N) -> `Course` -> (1:N) -> `Group` -> (1:N) -> `Schedule`. Викладацька гілка: `Structure` -> (1:N) -> `Chair` -> (1:N) -> `ChairTeacher` -> (1:N) -> `Schedule`. Профілі користувачів: `Setting` -> (1:1) -> `Student` -> `Group` та `Setting` -> (1:1) -> `Teacher` -> `ChairTeacher`. Ця структура забезпечує нормалізацію даних, запобігає дублюванню інформації та дозволяє ефективно виконувати запити на отримання розкладу для конкретної групи чи викладача з використанням JOIN-операцій.

Схема бази даних є спільною для всіх мікросервісів системи завдяки виділенню `Prisma` схеми у окремий пакет `@tta-duikt/prisma` в межах монорепозіторію `npnm workspaces` [3]. Це забезпечує єдине джерело правди щодо структури даних та усуває ризик розсинхронізації схем між сервісами. Кожна міграція фіксується у системі контролю версій та може бути відтворена на будь-якому екземплярі бази даних, що забезпечує відтворюваність середовища.

2 ЗАСОБИ РЕАЛІЗАЦІЇ ЗАСТОСУНКУ

2.1 Середовище розробки

2.1.1 Visual Studio Code / Cursor

Система ТТА DUIKT була створена за допомогою двох інтегрованих середовищ розробки (IDE): Visual Studio Code та Cursor. Visual Studio Code (VS Code) - це редактор коду з відкритим кодом, створений компанією Microsoft, який є найпопулярнішим інструментом для розробки на TypeScript [2] та JavaScript [20]. Додаток побудовано на базі платформи Electron і має багато функцій: підсвічування синтаксису для кількох сотень мов, інтелектуальне автодоповнення коду (IntelliSense), вбудований термінал, інтеграцію з Git, підтримку розширень та можливість налаштування робочого простору з монорепозіторіями. Для цілей проекту ТТА DUIKT особливо цінною є підтримка робочого простору `pnpm` [3], оскільки вона дозволяє правильно індексувати залежності між пакетами в монорепозіторії та перемикається між типами та модулями.

Cursor - це інноваційна інтегрована середовище розробки (IDE), створена на основі новітніх досягнень у галузі штучного інтелекту та VS Code. Ця IDE підтримує автозавершення коду за допомогою великих мовних моделей, пошук коду в межах усього проєкту та генерацію коду на основі текстових описів. За допомогою Cursor рутинні завдання, такі як створення CRUD-ендпойнтів, написання тестів та спрощення надлишкового коду, виконуються значно швидше. Обидві IDE можна використовувати як взаємозамінні, оскільки вони мають однакові налаштування, розширення та робочі простори.

2.2 Мови програмування

2.2.1 TypeScript

TypeScript [2] - це мова програмування зі статичною типізацією від компанії Microsoft, яка є надмножиною JavaScript [20]. Іншими словами, усі коректні програми, написані на JavaScript, автоматично є коректними програмами, написаними на TypeScript. Проте TypeScript запроваджує додаткову систему статичної типізації, яка дає змогу виявляти численні помилки під час компіляції. Поточна версія TypeScript, яка використовується в нашому проєкті - це версія 6.0.2, одна з найновіших.

Однією з найбільших переваг, яку TypeScript надає проєкту TTA DUIKT, є можливість писати всю логіку однією мовою. Дійсно, усі чотири мікросервіси (API, Bot, Mini App, Loader) та два пакети (@tta-duikt/prisma, @tta-duikt/types) використовують TypeScript як свою мову. Таким чином, для всього проєкту доступна уніфікована система типів. Типи даних, описані в пакеті @tta-duikt/types, можна імпортувати в будь-який сервіс без дублювання та зусиль із синхронізації, а якщо структура даних змінилася, компілятор TypeScript знайде всі місця, де слід внести зміни.

Крім того, важливо зазначити, що проєкт TTA DUIKT повністю розроблений з використанням стандарту ECMAScript Module (ESM), тобто сучасного стандарту для модульної системи JavaScript, який є офіційною частиною специфікації. ESM дозволяє проводити статичний аналіз залежностей, tree shaking та забезпечує належну семантику імпорту/експорту. Поєднання використання ESM та робочих просторів rprmt дає змогу отримати цілісну систему монорепозиторію з точки зору використання модулів, усуваючи будь-які проблеми сумісності, які часто виникають при одночасному використанні CommonJS та ESM [1].

Статичні типи TypeScript особливо корисні для додатків на базі мікросервісів, оскільки найскладніші проблеми виникають саме на рівні інтерфейсів сервісів (наприклад, коли у повідомленні в черзі є помилка або повідомлення, отримане через RabbitMQ, не відповідає очікуваному формату).

Спільні типи між API та Mini App через бібліотеку `@tta-duikt/types` гарантують, що клієнт отримує саме те, що повертає сервер, і роблять будь-які зміни в контракті сервісу видимими вже під час компіляції.

2.2.2 SQL

SQL (Structured Query Language) - це універсальна мова, що дозволяє взаємодіяти з реляційними базами даних. У рамках проекту ТТА DUKT використовується пакет Prisma ORM, який перетворює оператори SQL на основі декларативного опису структури даних та API додатка. Prisma перетворює виклики функцій (`findMany`, `create`, `update`, `$transaction`) в оптимізовані SQL-запити для системи баз даних PostgreSQL, що дозволяє програмістам працювати з базою даних на рівні абстракції об'єктів TypeScript та користуватися можливостями, які надає реляційна модель даних. Файли міграції, створені Prisma, містять SQL-оператори (`CREATE TABLE`, `ALTER TABLE`, `CREATE INDEX` тощо), які відображають зміни в структурі бази даних з плином часу.

2.3 Веб-фреймворки та бібліотеки

2.3.1 Node.js та NestJS

Node.js - це середовище виконання для запуску програм на JavaScript та TypeScript на стороні сервера, засноване на двигуні V8, розробленому для Google Chrome. Node.js використовує цикл подій та неблокуючу модель вводу-виводу, яка найкраще працює під час обробки численних одночасних з'єднань та операцій вводу-виводу (таких як мережеві виклики, запити до баз даних, взаємодія з чергами повідомлень). Для проекту ТТА DUKT необхідна версія Node.js 20 або вище, оскільки вона включає підтримку нативних модулів ESM та розширених мережевих API.

NestJS v.11.1.19 - це прогресивна платформа для створення високопродуктивних і масштабованих серверних додатків на базі Node.js. NestJS пропонує архітектурні принципи проектування, запозичені з Angular: модулі,

контролери, сервіси (провайдери), гварди, конвеєри (пайпи), інтерцептори та декоратори. Ці принципи гарантують правильний розподіл відповідальності та структурування коду, що є критично важливим для великих проєктів з численними кінцевими точками.

У рамках проєкту ТТА DUIKT для створення мікросервісу API використовується фреймворк NestJS із HTTP-адаптером Fastify (версія 5.8.5). У цьому випадку Fastify використовується замість Express, оскільки перший працює швидше за Express завдяки більш оптимізованому парсеру JSON, ефективнішому маршрутизатору та меншій кількості рівнів абстракції. Різниця у швидкості стає очевидною в API, які обробляють багато HTTP-запитів за секунду (наприклад, коли всі студенти одночасно підключаються до системи, щоб отримати розклад перед початком занять).

Архітектура API-сервісу побудована навколо кількох десятків модулів NestJS, які реалізують логіку кожного окремого домену в окремому модулі. Ці домени включають такі сутності, як розклад, ідентифікація (студентів та викладачів), налаштування, відгуки, структурні одиниці, факультети, кафедри, групи, кешування тощо. У свою чергу, кожен модуль складається з контролера, сервісу та, за потреби, деяких елементів безпеки (guard та pipe).

Автоматичне формування документації API здійснюється за допомогою модуля Swagger / OpenAPI, який було інтегровано в NestJS за допомогою декораторів. Кожен кінцевий пункт (ендпойнт) доповнюється інформацією, що описує очікувані вхідні дані, тіло запиту, формат вихідних даних та коди помилок. На основі цих декораторів автоматично створюється опис OpenAPI, який візуально відображається за допомогою Scalar - зручного інтерфейсу для перегляду та тестування документації API.

2.3.2 grammY

grammY - це інноваційна бібліотека для програмування ботів Telegram, створена на базі TypeScript та JavaScript. На відміну від інших фреймворків для ботів Telegram (наприклад, Telegraf, node-telegram-bot-api), grammY спеціально

розроблено для використання з TypeScript, пропонуючи вичерпні визначення типів, автозавершення та статичну перевірку помилок. Бібліотека використовує парадигму проектування проміжного програмного забезпечення (подібно до Express/Koa). У grammY кожна команда бота визначається як функція проміжного програмного забезпечення, яка обробляє оновлення та пересилає їх далі по конвеєру обробки.

У рамках проєкту TTA DUIKT використовується цей фреймворк, а також багато інших плагінів. Ці плагіни допомагають виконувати певні завдання в боті. По-перше, плагін auto-retry допомагає автоматично повторювати запит до API Telegram у разі отримання коду статусу «429 Too Many Requests», очікуючи значення, встановлене в заголовку відповіді Retry-After. Плагін Commands допомагає налаштовувати команди бота (start, help, privacy, developer_info, daily, announcement), включаючи створення меню команд, що здійснюється в автоматичному режимі залежно від контексту (звичайний користувач та адміністратор). Плагін Conversations реалізує концепцію розмов, що означає, що ви можете створювати складні інтерактивні сценарії спілкування, наприклад, вибір структури, факультету, курсу, а потім групи для ідентифікації студента. Плагін I18n (формат Fluent) дозволяє локалізувати повідомлення бота за допомогою файлів .flt різними мовами.

Він також дозволяє обробляти більш складні лінгвістичні поняття, такі як відмінювання, рід та множина. Плагін prse-mode автоматично встановлює режим розбору повідомлень - HTML або Markdown. Плагін runner дозволяє запускати бота в режимі довгого опитування та одночасно обробляти вхідні оновлення. Нарешті, плагін storage-redis зберігає сесії бота в KeyDB, зберігаючи стан розмови між користувачем.

2.3.3 React та Vite

React версії 19.2.5 є бібліотекою для побудови користувацьких інтерфейсів, розробленою компанією Meta (Facebook). React реалізує компонентний підхід, де інтерфейс декомпонується на ізольовані, багаторазово використовувані

компоненти, кожен з яких управляє власним станом та рендерингом. Версія 19 React включає суттєві покращення, зокрема React Compiler для автоматичної оптимізації рендерингу, покращену підтримку серверних компонентів та нові хуки для управління формами та транзакціями стану.

Збірка Mini App здійснюється за допомогою rolldown-vite версії 7.3.1 - високопродуктивного інструменту збірки, що поєднує швидкий сервер розробки (з гарячою заміною модулів - HMR) та оптимізовану продукційну збірку. Vite використовує нативні ESM-модулі браузера під час розробки, що забезпечує миттєвий запуск сервера розробки незалежно від розміру проєкту, та Rolldown (написаний на Rust) для продукційної збірки з tree-shaking, code-splitting та мініфікацією.

Стек бібліотек Mini App включає низку ретельно обраних інструментів. @tanstack/react-query забезпечує управління серверним станом: кешування запитів, автоматичне перезавантаження при фокусуванні вікна, оптимістичні оновлення та дедуплікацію запитів. Zustand є мінімалістичною бібліотекою управління клієнтським станом, яка забезпечує глобальний стан додатку без надмірної складності Redux. @tma.js/sdk-react - офіційний React-SDK для Telegram Mini Apps, який забезпечує доступ до нативних функцій платформи: тема оформлення, кнопка «Назад», головна кнопка, haptic feedback, біометрична автентифікація. @telegram-apps/telegram-ui - бібліотека UI-компонентів, стилізованих відповідно до дизайн-системи Telegram, що забезпечує візуальну інтеграцію Mini App з основним інтерфейсом месенджера. @mantine/core - бібліотека UI-компонентів загального призначення, що доповнює Telegram UI компонентами, відсутніми в останній. @radix-ui - набір примітивів доступності (accessible primitives) для складних інтерактивних компонентів. Vault - бібліотека для реалізації drawer-компонентів (висувних панелей) з нативними жестами. React-day-picker - компонент вибору дат, використовуваний для вибору діапазону перегляду розкладу.

Tailwindcss - utility-first CSS-фреймворк, що забезпечує швидку розробку інтерфейсу через класи-утиліти безпосередньо в JSX-розмітці. Lucide-react - бібліотека іконок, що забезпечує консистентну іконографію інтерфейсу. P18next -

фреймворк інтернаціоналізації для Mini App з підтримкою lazy-loading мовних пакетів, що дозволяє завантажувати англійську локалізацію лише за потреби, зменшуючи початковий розмір бандлу.

Маршрутизація Mini App реалізована через HashRouter із бібліотеки react-router-dom. Використання HashRouter замість BrowserRouter є технічною вимогою платформи Telegram Mini Apps, оскільки Mini App завантажується у вбудованому WebView, який не підтримує History API для SPA-навігації. HashRouter використовує фрагмент URL (після символу '#') для визначення поточного маршруту, що є повністю клієнтським механізмом і не потребує серверної підтримки. Сторінки Mini App включають: / (головна), /identity (вибір ролі), /identity/student (ідентифікація студента), /identity/teacher (ідентифікація викладача), /schedule (перегляд розкладу), /settings (налаштування), /feedback (зворотний зв'язок).

2.4 Система управління базами даних

2.4.1 PostgreSQL

PostgreSQL версії 16 є потужною об'єктно-реляційною системою управління базами даних з відкритим вихідним кодом, яка розвивається понад 35 років та є однією з найбільш розвинених СУБД у світі. PostgreSQL підтримує складні типи даних, повнотекстовий пошук, JSON/JSONB-поля, масиви, діапазони, геопросторові дані та користувацькі типи. Система забезпечує повну відповідність стандарту ACID (Atomicity, Consistency, Isolation, Durability), що гарантує цілісність даних навіть у разі збоїв.

У проєкті TTA DUKT PostgreSQL є єдиним реляційним сховищем, що зберігає всі структуровані дані системи: профілі та налаштування користувачів, організаційну ієрархію університету (структури, факультети, курси, групи, кафедри, викладачі) та дані розкладу занять. Вибір PostgreSQL обумовлений його надійністю, широкою підтримкою з боку Prisma ORM, ефективністю при роботі зі складними JOIN-запитами (що є типовими для ієрархічної структури даних

розкладу) та підтримкою транзакцій, необхідних для атомарного оновлення даних розкладу через Prisma `$transaction`.

2.4.2 Prisma

Prisma версії 6.19.0 є сучасним ORM (Object-Relational Mapping) для TypeScript та Node.js, який забезпечує типобезпечний доступ до бази даних. На відміну від традиційних ORM, які використовують патерн Active Record або Data Mapper, Prisma генерує типобезпечний клієнт на основі декларативної схеми, описаної у `.prisma`-файлах. Кожна зміна схеми автоматично відображається у згенерованих TypeScript-типах, що забезпечує повну синхронізацію між базою даних та кодом додатку.

У проєкті ТТА DUIKT Prisma-схема використовує можливість `multi-file schema`, що дозволяє розділити опис моделей на декілька файлів для кращої організації коду. Схема виділена у окремий пакет `@tta-duikt/prisma` в межах монорепозіторію `rnpm workspaces`, що дозволяє всім мікросервісам імпортувати згенерований клієнт Prisma та використовувати єдину схему бази даних. Протягом розробки було створено 6 міграцій, кожна з яких фіксує конкретний етап еволюції схеми: від початкового створення таблиць до додавання нових полів, індексів та обмежень. Міграції зберігаються у системі контролю версій та автоматично застосовуються при розгортанні системи.

2.5 Кешування та сесії

2.5.1 KeyDB

KeyDB є високопродуктивним, Redis-сумісним сховищем даних у пам'яті з відкритим вихідним кодом. KeyDB повністю сумісний із протоколом Redis, що дозволяє використовувати всі існуючі Redis-клієнти та бібліотеки без модифікацій. Головною перевагою KeyDB є підтримка багатопотоковості, на відміну від однопотокового Redis, що забезпечує вищу продуктивність на багатоядерних серверах.

У проєкті ТТА DUIKT KeyDB використовується для двох ключових задач: кешування даних розкладу (cache-aside pattern) та зберігання сесій Telegram-бота. Стратегія кешування cache-aside передбачає, що при запиті даних система спочатку перевіряє наявність результату в кеші KeyDB, якщо дані знайдені (cache hit), вони повертаються безпосередньо з кешу, якщо дані відсутні (cache miss), система звертається до бази даних або першоджерела, зберігає результат у кеші з відповідним TTL (Time To Live) та повертає його клієнту.

Система використовує диференційовані значення TTL залежно від типу даних та середовища. Для даних розкладу у продукційному середовищі TTL становить 30 хвилин - це забезпечує баланс між актуальністю даних та навантаженням на базу даних і першоджерело. Для порожніх відповідей (коли розклад для обраних параметрів відсутній) TTL становить 5 хвилин - коротший час кешування дозволяє швидше виявити появу нових даних. У середовищі розробки TTL становить лише 5 секунд, що дозволяє розробникам бачити зміни майже в реальному часі. HTTP-заголовок X-Data-Status у відповідях API інформує клієнтів про джерело даних: значення `cached` вказує на отримання даних із KeyDB, а `fresh` - на отримання безпосередньо з бази даних або першоджерела.

Для зберігання сесій бота `grammY` використовує плагін `storage-redis`, сумісний з KeyDB. Сесії зберігають стан діалогу користувача (наприклад, на якому етапі ідентифікації він знаходиться, які параметри вже обрав), що дозволяє відновити діалог після перезапуску бота або переключення між різними `middleware`-обробниками.

2.6 Брокер повідомлень та черги

2.6.1 RabbitMQ

RabbitMQ є одним із найпоширеніших брокерів повідомлень з відкритим вихідним кодом, який реалізує протокол AMQP (Advanced Message Queuing Protocol). RabbitMQ забезпечує надійну доставку повідомлень між розподіленими

компонентами системи, підтримуючи різні патерни обміну повідомленнями: point-to-point, publish/subscribe, request/reply (RPC) та routing.

У проєкті ТТА DUIKT RabbitMQ використовується для синхронної міжсервісної комунікації за паттерном RPC (Remote Procedure Call) між мікросервісом Mini App API та мікросервісом Bot. Система використовує 2 durable черги: feedback (для передачі повідомлень зворотного зв'язку від користувачів) та language (для синхронізації зміни мови інтерфейсу між Mini App та ботом). Durable черги зберігають повідомлення на диску, що гарантує їхнє збереження навіть у разі перезапуску брокера RabbitMQ.

Паттерн RPC реалізований через механізм correlationId та тимчасових черг відповідей: відправник генерує унікальний correlationId, додає його до повідомлення разом із ім'ям черги для відповіді, та очікує відповідь із відповідним correlationId. Тайм-аут очікування відповіді встановлений у 5000 мілісекунд, після чого запит вважається невдалим і повертається відповідна помилка. Цей підхід дозволяє Mini App API синхронно запитувати дії від бота (наприклад, надіслати повідомлення адміністратору) та отримувати підтвердження виконання.

2.6.2 Bull

Bull версії 4.16.5 є бібліотекою для управління чергами задач у Node.js, побудованою на основі Redis. Bull забезпечує надійну обробку фонових задач із підтримкою пріоритетів, повторних спроб, відкладеного виконання, обмеження швидкості (rate limiting) та конкурентної обробки.

У проєкті ТТА DUIKT Bull використовується для трьох черг задач у мікросервісі Bot: dailyAlert (щоденні нагадування про розклад), message (надсилання індивідуальних повідомлень користувачам) та poll (надсилання опитувань). Ключовою конфігурацією є обмеження швидкості (rate limiter): максимум 25 повідомлень за 2500 мілісекунд (2.5 секунди). Це обмеження обумовлене лімітами Telegram Bot API, які дозволяють ботові надсилати не більше 30 повідомлень на секунду у різні чати. Встановлення ліміту на рівні 25/2.5s забезпечує запас безпеки та запобігає отриманню помилки 429 Too Many Requests.

Стратегія повторних спроб використовує експоненціальний backoff, у разі невдалого надсилання повідомлення (наприклад, через тимчасову недоступність Telegram API), задача повторюється з експоненціально зростаючою затримкою між спробами. Це зменшує навантаження на систему при масових збоях та підвищує ймовірність успішної доставки. Особливою поведінкою є автоматичне відключення сповіщень при отриманні HTTP-статусів 403 (Forbidden - користувач заблокував бота) або 400 (Bad Request - чат не існує): у таких випадках система автоматично вимикає прапорець щоденних сповіщень у профілі відповідного користувача, запобігаючи повторним невдалим спробам.

2.7 Моніторинг та спостережуваність

2.7.1 Sentry

Sentry є хмарною платформою для моніторингу помилок та продуктивності додатків у реальному часі. Sentry автоматично перехоплює необроблені виключення, помилки промісів, мережеві збої та інші аномалії, збираючи детальний контекст: стек виклику, змінні середовища, параметри запиту, стан додатку та breadcrumbs (хронологію подій, що передували помилці). У проєкті ТТА DUIKT Sentry інтегрований в усі чотири мікросервіси (API, Bot, Mini App, Loader), що забезпечує єдину точку моніторингу для всієї системи. Кожен сервіс має окремий DSN (Data Source Name), що дозволяє фільтрувати помилки за джерелом та відстежувати стабільність кожного компонента незалежно.

2.7.2 Elasticsearch та Kibana

Elasticsearch є розподіленою пошуковою та аналітичною системою, здатною зберігати, шукати та аналізувати великі обсяги даних у близькому до реального часу режимі. Kibana є візуальним інтерфейсом для Elasticsearch, що надає можливості побудови дашбордів, графіків та таблиць на основі даних, збережених в Elasticsearch.

У проєкті ТТА DUIKT Elasticsearch використовується для збору та зберігання логів взаємодії користувачів з Telegram-ботом. Система індексує дані у два окремі індекси: `incoming_updates` (вхідні оновлення - повідомлення, команди та `callback`-запити від користувачів) та `outgoing_updates` (вихідні оновлення - повідомлення, надіслані ботом користувачам). Такий поділ дозволяє окремо аналізувати поведінку користувачів (які команди використовують найчастіше, в який час найбільша активність) та поведінку системи (час відповіді, обсяг надісланих повідомлень, частота помилок). Kibana надає візуальні дашборди для моніторингу цих метрик у реальному часі.

2.8 Контейнеризація та CI/CD

2.8.1 Docker

Docker є платформою контейнеризації, яка дозволяє упаковувати додатки та їхні залежності в ізольовані, портативні контейнери. Контейнери забезпечують однакову поведінку додатку незалежно від середовища, в якому вони запускаються: на робочій машині розробника, на тестовому сервері чи у продукційному середовищі.

У проєкті ТТА DUIKT Docker використовується з `multi-stage builds` - техніка, яка дозволяє розділити процес збірки контейнера на декілька етапів для мінімізації розміру кінцевого образу. На першому етапі (`builder`) встановлюються всі залежності (включаючи `dev`-залежності), компілюється TypeScript-код та виконується збірка додатку. На другому етапі (`production`) створюється мінімальний образ на основі `node:22-alpine`, який містить лише скомпільований код та продукційні залежності. Базовий образ `node:22-alpine` побудований на Alpine Linux - мінімалістичному дистрибутиві розміром ~5 МБ, що значно зменшує розмір кінцевого контейнера та зменшує поверхню атаки з точки зору безпеки.

Docker Compose у продукційній конфігурації (`docker-compose.prod.yaml`) об'єднує всі компоненти системи в єдиний стек: чотири мікросервіси (API, Bot, Mini App, Loader), PostgreSQL 16, KeyDB, RabbitMQ, Elasticsearch, Kibana та сервіс

автоматичного резервного копіювання. Сервіс резервного копіювання виконує дампи бази даних PostgreSQL кожні 12 годин та завантажує його до S3-сумісного об'єктного сховища, забезпечуючи можливість відновлення даних у разі катастрофічного збою.

2.9 Система контролю версій

2.9.1 Git та GitHub

Git є розподіленою системою контролю версій, яка є стандартом де-факто для управління вихідним кодом у індустрії розробки програмного забезпечення. Git забезпечує повну історію змін, гілкування, злиття та можливість спільної роботи декількох розробників над одним проєктом.

Проєкт TTA DUKT організований як монорепозиторій (monorepo) - єдиний Git-репозиторій, що містить усі мікросервіси та спільні пакети. Монорепозиторій управляється за допомогою `pnpm workspaces`, що забезпечує ефективне управління залежностями між пакетами, єдиний `lockfile` та можливість запуску команд для всіх або обраних пакетів одночасно. Репозиторій розміщений на платформі GitHub, яка забезпечує хостинг коду, відстеження задач (issues), код-рев'ю (pull requests) та інтеграцію з CI/CD-системами.

Для підтримки актуальності залежностей у проєкті налаштований GitHub Dependabot із щоденною частотою перевірки оновлень. Dependabot автоматично створює pull requests із оновленнями залежностей, що дозволяє оперативно отримувати виправлення безпеки та нові функції бібліотек. У контексті проєкту з великою кількістю залежностей (NestJS, grammY, React, Prisma, Bull, численні плагіни та утиліти) автоматизація оновлення залежностей є критично важливою для підтримки безпеки та стабільності системи.

2.10 Інструменти дизайну інтерфейсу

2.10.1 Figma

Figma є хмарним інструментом для проєктування користувацьких інтерфейсів та прототипування, який є стандартом індустрії для UX/UI-дизайну. Figma працює безпосередньо у веб-браузері, що забезпечує доступ із будь-якого пристрою без необхідності встановлення нативного додатку, хоча десктопний додаток також доступний. Ключовою перевагою Figma є можливість спільної роботи в реальному часі: декілька учасників можуть одночасно працювати над одним файлом дизайну, залишати коментарі та переглядати зміни.

У проєкті ТТА DUIKT Figma використовувалася для проєктування інтерфейсу Mini App: створення макетів усіх сторінок (головна, ідентифікація, розклад, налаштування, зворотний зв'язок), визначення кольорової палітри відповідно до дизайн-системи Telegram, проєктування компонентів інтерфейсу та їхніх станів (завантаження, помилка, порожній стан, заповнений стан), а також прототипування навігаційних переходів між сторінками. Макети Figma слугували специфікацією для розробки фронтенд-частини та забезпечували консистентність візуального оформлення між різними сторінками та компонентами Mini App.

3 ПРОЄКТУВАННЯ СИСТЕМИ

3.1 Проєктування архітектури застосунку

Розроблювана система «ТТА DUIKT» (Timetable Assistant DUIKT) являє собою мікросервісну архітектуру, що складається з чотирьох окремих сервісів та двох спільних пакетів, об'єднаних у єдиний монорепозіторій під управлінням менеджера пакетів `pnpm` із підтримкою `workspaces`. Такий підхід дозволяє ізолювати відповідальність кожного компонента, забезпечити незалежне масштабування та спростити процес розгортання і супроводу системи в цілому.

Архітектура системи побудована на принципах мікросервісного підходу, де кожен сервіс виконує чітко визначену функцію та взаємодіє з іншими компонентами через стандартизовані протоколи комунікації. Основними сервісами системи є:

1. **API Service** - серверний застосунок, побудований на базі фреймворку `NestJS` із використанням HTTP-адаптера `Fastify`. Цей сервіс є центральним вузлом системи, який обробляє всі запити від клієнтських додатків, виконує бізнес-логіку, взаємодіє з базою даних та зовнішнім API університету. Сервіс надає `RESTful API` з документацією `OpenAPI/Swagger` та підтримкою `Scalar` для інтерактивного перегляду документації.

2. **Bot Service** - Telegram-бот, реалізований за допомогою фреймворку `grammy`, який забезпечує інтерфейс взаємодії з користувачами через платформу Telegram. Бот відповідає за відправку щоденних сповіщень про розклад, сповіщень про зміни у розкладі, обробку адміністративних команд та розсилку оголошень. Сервіс інтегрується з системою черг `Bull` для обмеження швидкості відправки повідомлень відповідно до лімітів Telegram Bot API.

3. **Mini App Service** - клієнтський веб-застосунок, реалізований як Telegram Mini App за допомогою `React` та збірника `Vite`. Цей сервіс надає графічний інтерфейс для перегляду розкладу, управління налаштуваннями користувача,

відправки зворотного зв'язку та ідентифікації ролі (студент або викладач). Mini App працює безпосередньо у середовищі Telegram, використовуючи TMA SDK для інтеграції з нативними елементами платформи.

4. Loader Service - фоновий сервіс, що виконує періодичну синхронізацію даних із зовнішнім API університету. Реалізований як cron-задача, яка кожні 30 хвилин обходить усю ієрархію навчального закладу (структури, факультети, курси, групи, кафедри, викладачі) та ініціює оновлення розкладу через запити до API Service.

Система використовує два спільні пакети, що забезпечують типобезпечність та консистентність між сервісами:

1. @tta-duikt/prisma - пакет, який містить Prisma-схему бази даних, згенерований Prisma Client та міграції. Використовується безпосередньо сервісами API та Bot для доступу до бази даних PostgreSQL 16.

2. @tta-duikt/types - пакет, що містить спільні TypeScript-типи, перелічення (enums) та константи, які використовуються усіма сервісами системи. Зокрема, цей пакет визначає імена черг RabbitMQ, коди помилок HTTP-відповідей, формати дат та інші спільні контракти.

Інфраструктурний рівень системи складається з наступних компонентів:

1. PostgreSQL 16 - основна реляційна база даних для зберігання усієї предметної інформації: розклад занять, дані користувачів, структура навчального закладу, налаштування.

2. KeyDB - високопродуктивний in-memoу сховище даних, сумісне з протоколом Redis, що використовується для кешування відповідей API, зберігання сесій Telegram-бота та конфігурації черг Bull.

3. RabbitMQ - брокер повідомлень, що забезпечує асинхронну міжсервісну комунікацію між API та Bot за шаблоном RPC (Remote Procedure Call). Використовується для доставки зворотного зв'язку від користувачів до адміністратора та інвалідації мовних сесій.

4. Elasticsearch - пошукова система та аналітична платформа, що використовується для індексації вхідних та вихідних оновлень Telegram-бота з метою моніторингу та аналітики.

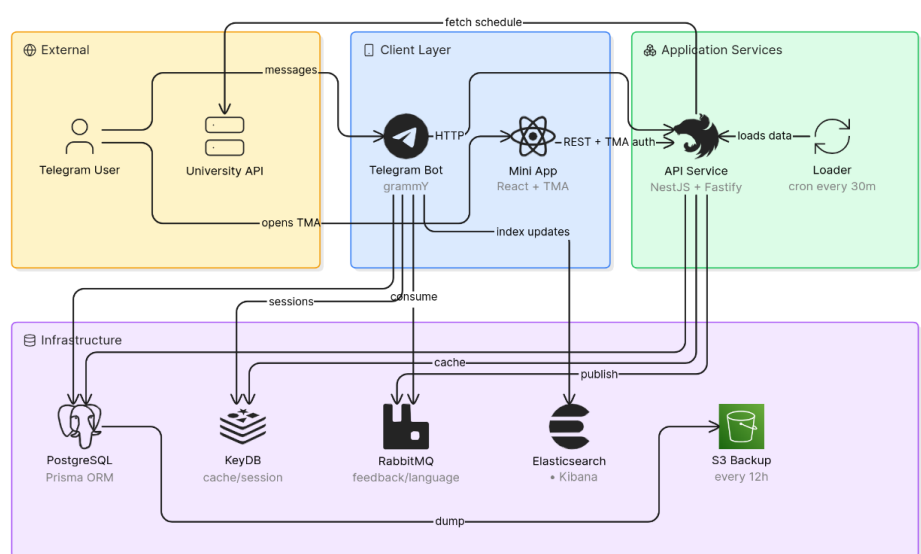
5. Kibana - візуалізаційна платформа для перегляду та аналізу даних, збережених в Elasticsearch.

Патерни комунікації між компонентами системи організовані наступним чином:

1. HTTP-взаємодія: Mini App взаємодіє з API через HTTPS-запити із заголовком авторизації TMA, Bot звертається до API за допомогою типобезпечного HTTP-клієнта openapi-fetch, Loader ініціює оновлення розкладу через HTTP-запити до API із заголовками, що забороняють кешування.

2. RabbitMQ: API відправляє повідомлення до Bot для доставки зворотного зв'язку адміністратору та інвалідації мовних сесій у KeyDB. Використовується шаблон запит-відповідь із кореляційними ідентифікаторами.

3. Bull Queues: Bot використовує внутрішні черги Bull для контрольованої відправки повідомлень у Telegram із обмеженням швидкості (rate limiting), що запобігає перевищенню лімітів Telegram Bot API.



eraser

Рис. 3.1 Дизайн високого рівня

3.1.1 Діаграма діяльності користувача

Діаграма діяльності описує повний цикл взаємодії користувача із системою «ТТА ДУІКТ», починаючи від першого контакту з Telegram-ботом до регулярного використання функціональних можливостей Mini App та отримання автоматичних сповіщень.

Первинна взаємодія починається з того, що користувач відкриває Telegram-бота та натискає кнопку запуску Mini App, яка відображається як WebApp-кнопка в інтерфейсі бота. Після натискання Telegram завантажує веб-додаток у вбудованому WebView-компоненті.

При завантаженні Mini App виконується ініціалізація TMA SDK: монтуються компоненти `themeParams`, `miniApp`, `viewport` (з розширенням на весь екран), `mainButton` та `swipeBehavior` (з вимкненням вертикального свайпу для запобігання випадковому закриттю). Далі додаток відправляє GET-запит до ендпоінту `/user/me` із заголовком авторизації, що містить TMA `init-data`.

Якщо API повертає помилку з кодом 404 та повідомленням `USER_NOT_FOUND`, це означає, що користувач ще не пройшов ідентифікацію. У цьому випадку додаток перенаправляє користувача на сторінку `Identity`, де пропонується обрати роль: «Студент» або «Викладач». Кожна роль представлена окремою карткою з відповідною SVG-іконкою.

Для ролі «Студент» користувач послідовно обирає: Структуру навчального закладу -> Факультет -> Курс -> Групу. Кожен наступний крок стає доступним лише після вибору попереднього, а при зміні вибору на попередньому кроці всі наступні скидаються. Дані для кожного випадаючого списку завантажуються динамічно з API за допомогою хуків `@tanstack/react-query`.

Для ролі «Викладач» процес аналогічний, але містить менше кроків: Структура -> Кафедра -> Викладач. Після заповнення всіх полів з'являється кнопка «Зберегти», натискання якої відправляє PATCH-запит до `/user/me` із обраними даними.

Після успішної ідентифікації користувач потрапляє на головну сторінку (`MainPage`), де відображається навігаційне меню з доступними розділами: розклад,

налаштування, ідентифікація (зміна ролі), донат та зворотний зв'язок. Також відображається інформація про обліковий запис: структура, факультет, курс, група (або кафедра та ім'я викладача), а також час з моменту реєстрації.

На сторінці розкладу (SchedulePage) користувач взаємодіє із компонентом react-day-picker, який відображає календар із підтримкою вибору діапазону дат (до 30 днів). При виборі дати або діапазону з'являється mainButton із текстом, що описує обрану дію. Натискання кнопки відкриває модальне вікно ScheduleModal, реалізоване як Vault Drawer (висувна панель знизу).

Модальне вікно розкладу завантажує дані з API, групує їх за датами та відображає за допомогою компонента Mantine Timeline. Кожен елемент Timeline має кольорове кодування: зелена суцільна лінія для минулих занять, червона штрихова для поточного заняття та штрихова без кольору для майбутніх. Також доступна функція поширення розкладу через TMA shareURL(), яка форматує текстове представлення розкладу.

Сторінка налаштувань (SettingsPage) дозволяє перемикає параметри профілю: сповіщення про оновлення розкладу, щоденні сповіщення, сповіщення про оголошення, відображення емодзі, беззвучні сповіщення без розкладу, доставка сповіщень без розкладу, отримання сповіщень у вихідні, автоматичне відкриття розкладу на завтра. Також доступний селектор мови (українська/англійська).

Сторінка зворотного зв'язку (FeedbackPage) містить текстову область з лічильником символів (максимум 2000) та кнопкою відправки, що активується при введенні тексту та змінює колір на червоний при перевищенні ліміту або блокуванні функції зворотного зв'язку.

Паралельно з використанням Mini App, бот автоматично відправляє щоденні сповіщення про розклад на наступний день о 15:00 за київським часом (з неділі по четвер для основної аудиторії, у п'ятницю - для тих, хто підписався на сповіщення у вихідні). Також бот відправляє сповіщення про зміни в розкладі при виявленні різниці між даними з API університету та локальною базою.

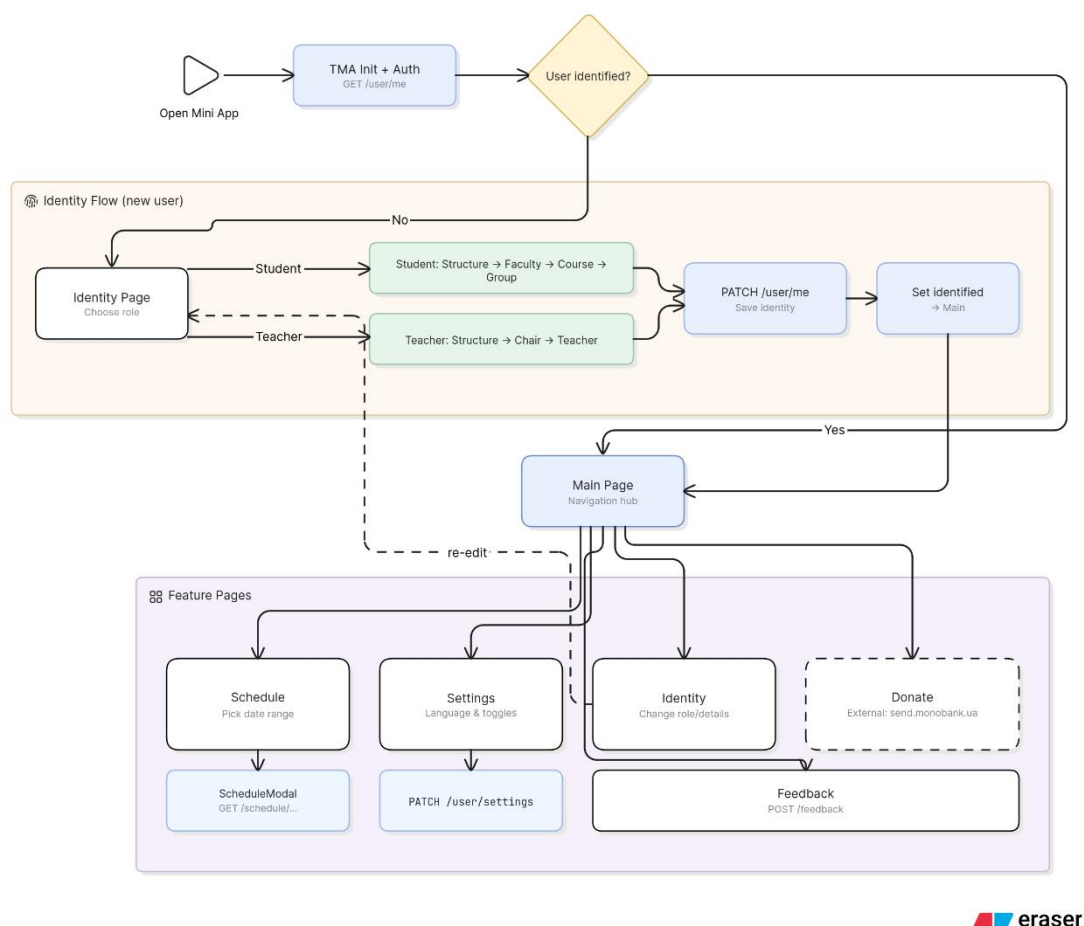


Рис. 3.2 Діаграма користувацького шляху

3.1.2 Діаграма варіантів використання

Діаграма варіантів використання системи «ТТА DUKT» визначає три основних актори, кожен з яких має визначений набір доступних дій.

1. Неавторизований користувач (користувач без встановленої ідентичності) має обмежений набір можливостей:

- відкрити Telegram Mini App через бота.
- переглянути сторінку ідентифікації, де доступний вибір ролі (студент або викладач).
- пройти процес ідентифікації: послідовно обрати структуру, факультет/кафедру, курс/викладача та групу.
- зберегти обрану ідентичність та перейти до функціоналу авторизованого користувача.

При спробі доступу до будь-якого іншого маршруту, окрім сторінки ідентифікації, неавторизований користувач автоматично перенаправляється на сторінку вибору ролі.

2. Авторизований користувач (студент або викладач з підтвердженою ідентичністю) має повний набір функціональних можливостей:

- переглядати розклад за обрану дату або діапазон дат (до 31 дня) через календар Mini App.
- змінювати налаштування сповіщень та персоналізації: увімкнення/вимкнення параметрів через перемикачі.
- змінювати мову інтерфейсу (українська або англійська) із миттєвим застосуванням.
- відправляти зворотний зв'язок розробнику через спеціальну форму (до 2000 символів).
- отримувати щоденні сповіщення від бота о 15:00 з розкладом на наступний день.
- отримувати сповіщення про виявлені зміни у розкладі.
- поширювати розклад у текстовому форматі через функцію TMA shareURL.
- змінювати свою ідентичність: перейти від ролі студента до викладача та навпаки.

3. Адміністратор (користувач із правами адміністратора, визначений через змінну середовища ADMIN_ID) має додаткові привілейовані можливості:

- відправляти оголошення (повідомлення або опитування) цільовій аудиторії: всім користувачам, лише студентам, лише викладачам або конкретному користувачу за ідентифікатором чату.
- налаштовувати параметри розсилки: зі звуком або без.
- обирати тип оголошення: текстове повідомлення або опитування (poll із до 10 варіантами відповіді).
- ініціювати ручний запуск щоденного сповіщення через команду /daily.
- блокувати функцію зворотного зв'язку для конкретного користувача через inline-кнопку у повідомленні зворотного зв'язку.

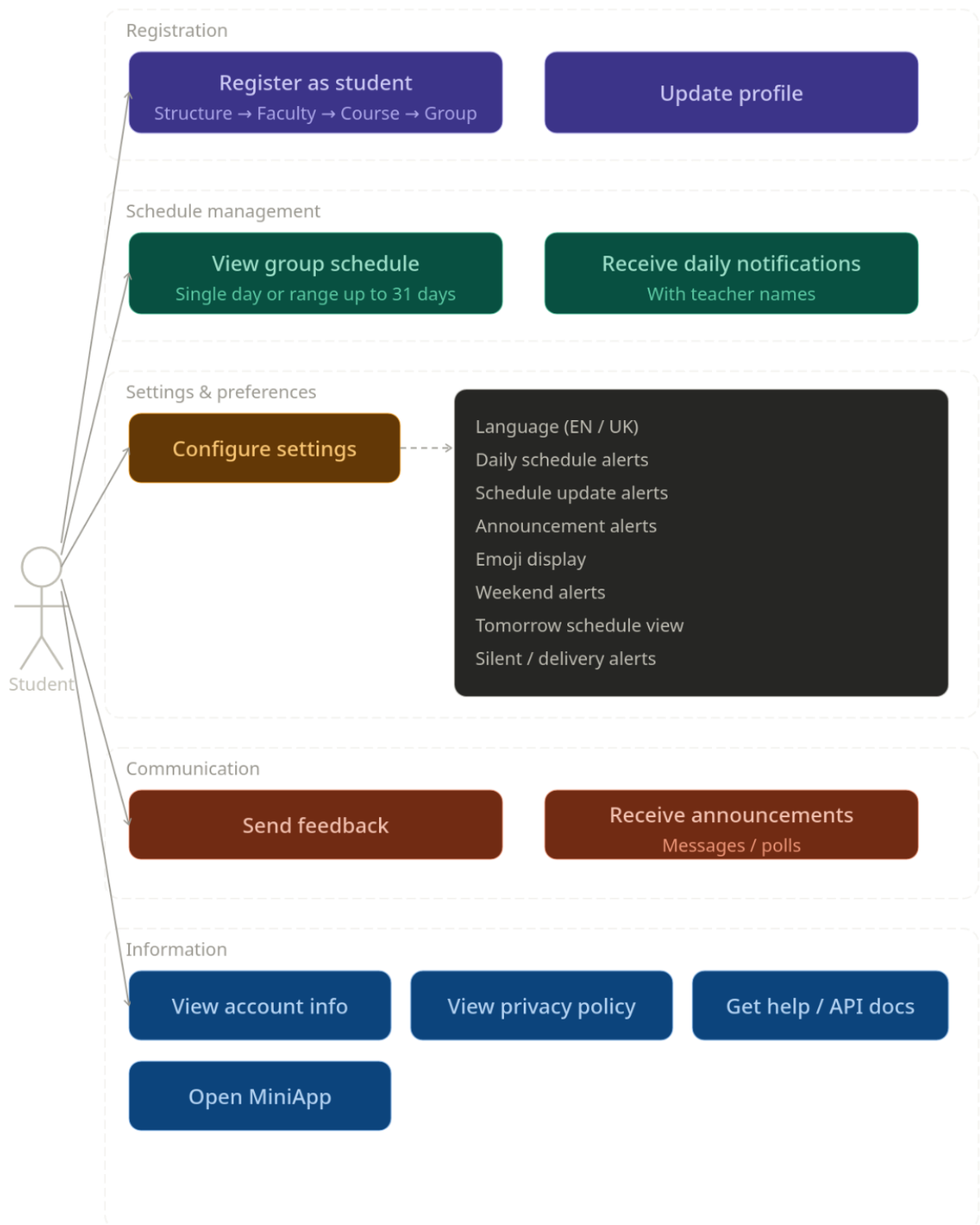


Рис. 3.3 Діаграма прецедентів №1

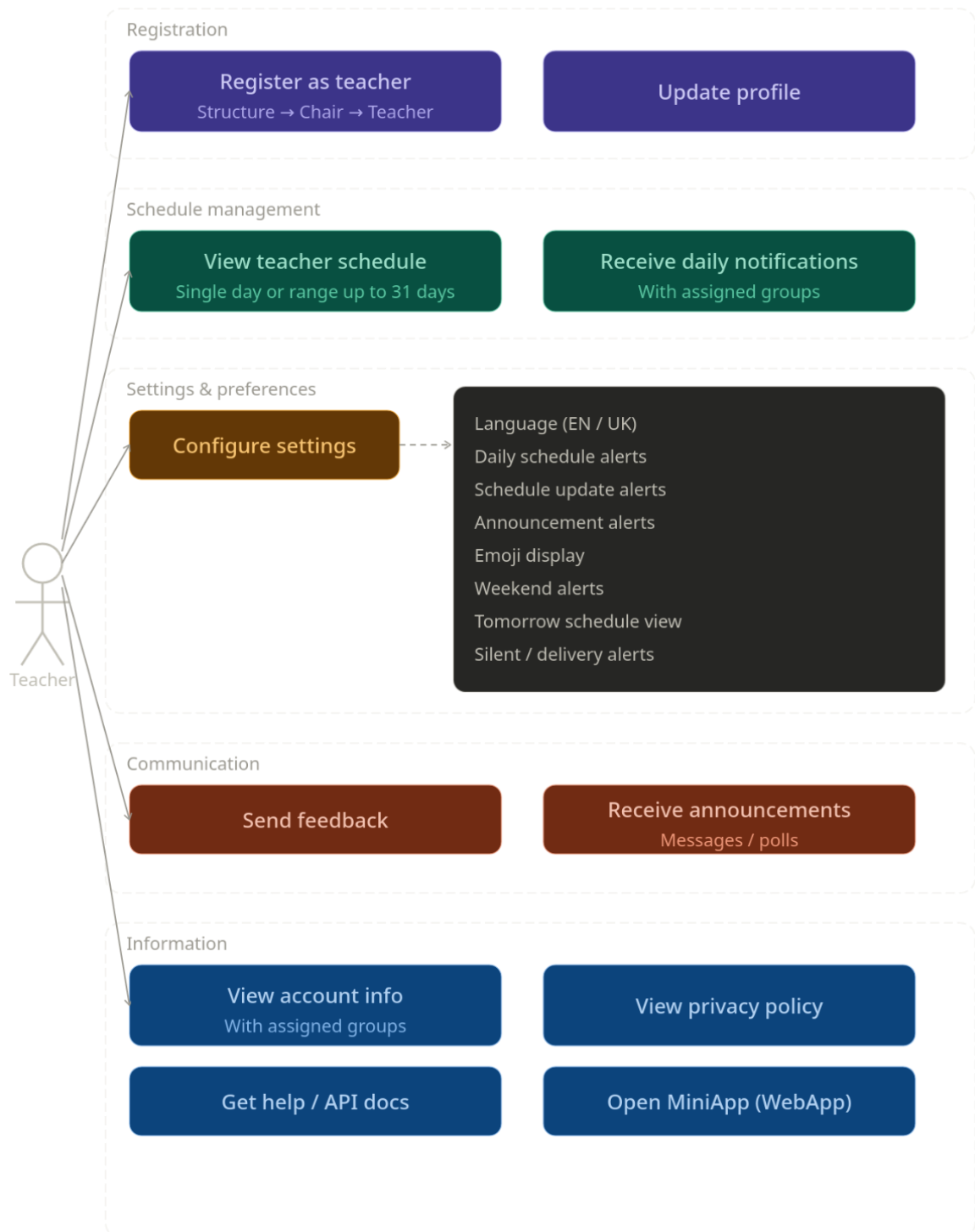


Рис. 3.4 Діаграма прецедентів №2

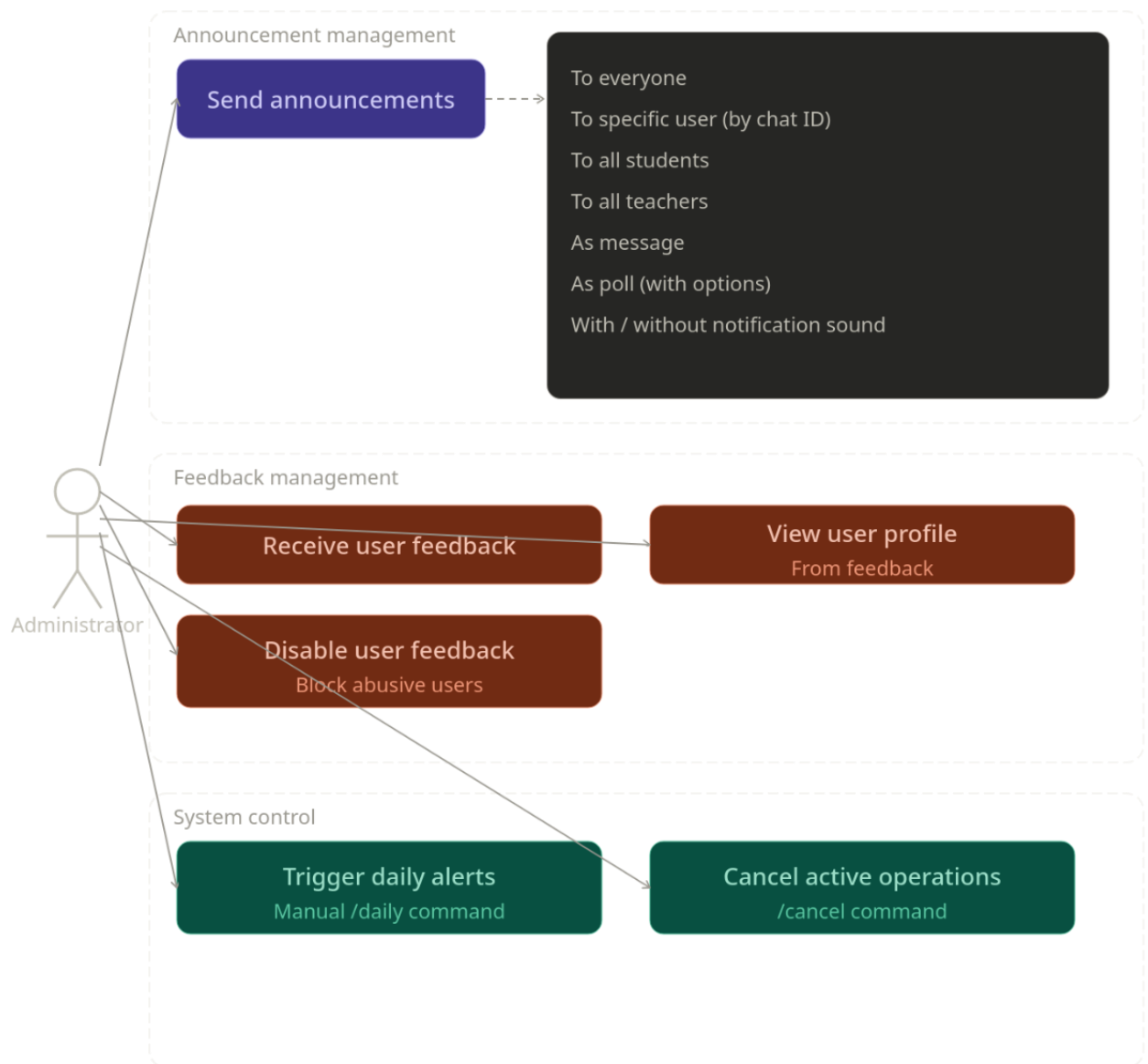


Рис. 3.5 Діаграма прецедентів №3

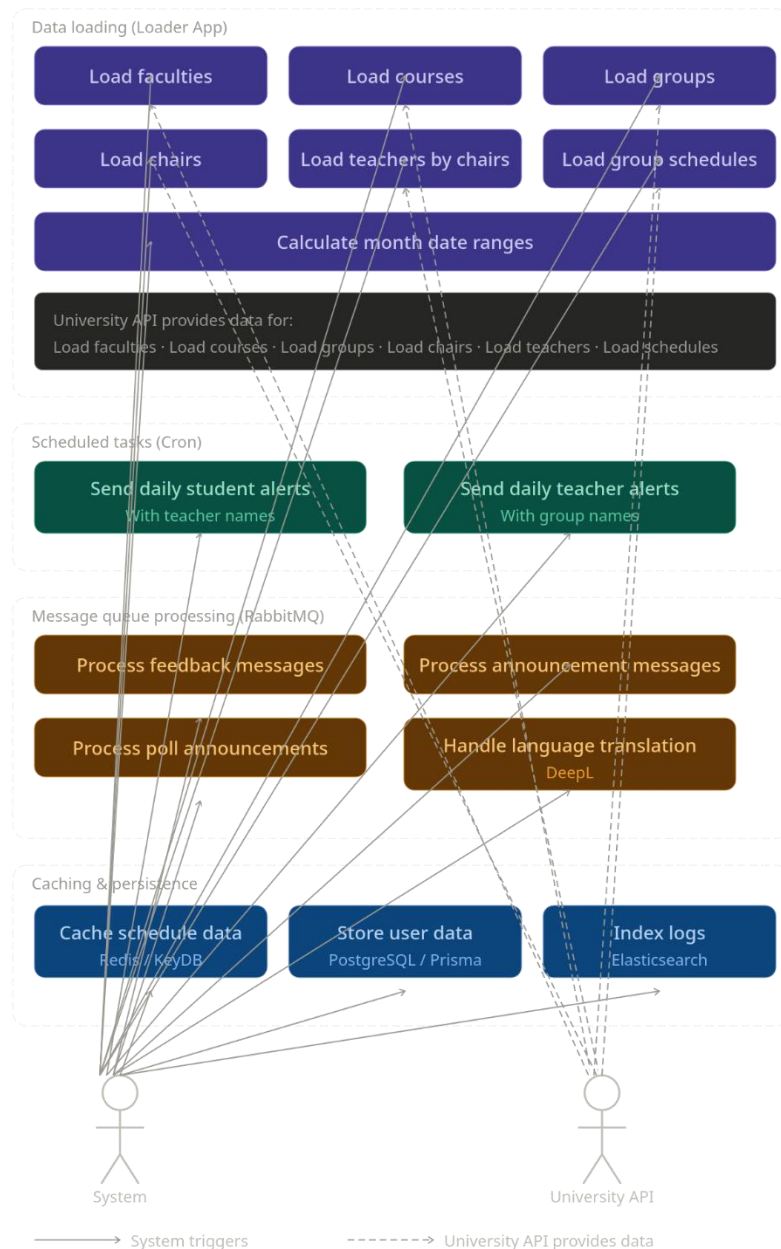


Рис. 3.6 Діаграма прецедентів №4

3.2 Архітектура клієнтської частини (Telegram Mini App)

Клієнтська частина системи «ТТА ДУІКТ» реалізована як Telegram Mini App - веб-застосунок, що працює в інтегрованому WebView-компоненті месенджера Telegram. Архітектура побудована на основі бібліотеки React із використанням

збірника Vite та мови TypeScript, що забезпечує типобезпечність, модульність та високу продуктивність.

Точка входу та ініціалізація. Вхідною точкою застосунку є файл `index.tsx`, який ініціалізує Sentry для моніторингу помилок із підтримкою `replay`-записів, трасування продуктивності та інтеграцією з консоллю браузера. Після ініціалізації Sentry створюється кореневий React-елемент. Кореневий компонент `Root` формує багаторівневу структуру провайдерів, що забезпечують функціональність усього додатку.

Ієрархія компонентів побудована за наступним принципом:

- `ErrorBoundary` - перехоплення та відображення помилок рендерингу із `fallback`-компонентом `ErrorBoundaryError`.

- `TmaInit` - ініціалізація Telegram Mini App SDK, необхідна для роботи ТМА-специфічних функцій.

- `I18nextProvider` - забезпечення інтернаціоналізації через `i18next` із підтримкою української (за замовчуванням) та англійської мов, причому англійська завантажується лише за потребою (`lazy-loading`).

- `QueryClientProvider` - провайдер `@tanstack/react-query` для управління серверним станом, кешування запитів та автоматичного оновлення даних.

- `MantineProvider` - провайдер UI-бібліотеки Mantine для компонента `Timeline`, який використовується у відображенні розкладу.

- `Toaster` - компонент для відображення `toast`-сповіщень.

Головний компонент `App`. Компонент `App` є ядром додатку, що виконує ініціалізацію ТМА SDK, завантаження даних користувача та маршрутизацію.

Ініціалізація ТМА SDK включає монтування компонентів `themeParams`, `miniApp`, `viewport` (із розширенням на весь екран), `mainButton` та `swipeBehavior`. Останній вимикає вертикальний свайп, щоб запобігти випадковому закриттю Mini App при прокрутці контенту. CSS-змінні прив'язуються до параметрів теми Telegram, забезпечуючи адаптацію кольорової схеми.

Маршрутизація. Застосунок використовує HashRouter замість BrowserRouter, що є обов'язковою вимогою для Telegram Mini App, оскільки WebView-контейнер не підтримує History API.

Усі сторінки завантажуються через механізм `React.lazy()` для оптимізації розміру початкового бандлу. Компонент `AppRoutes` реалізує захист маршрутів: якщо користувач не ідентифікований і намагається перейти на сторінку, відмінну від `identity`, він автоматично перенаправляється на сторінку ідентифікації.

Управління станом. Глобальний стан додатку управляється через Zustand - легковагу бібліотеку стану для React:

Стор зберігає три ключові стани: `isUserIdentified` (чи ідентифікований користувач), `user` (дані користувача - студент або викладач) та `settings` (налаштування). Типізація забезпечується за допомогою спільного пакету `@ttdiikt/types` та згенерованих OpenAPI-типів.

Отримання даних з сервера. Для взаємодії з API використовується бібліотека `openapi-fetch`, яка генерує типобезпечний клієнт на основі OpenAPI-специфікації. Авторизація реалізована через заголовок `Authorization: tma ${initDataRaw}`, де `initDataRaw` - це raw init-data, отримана від TMA SDK. Для кешування та управління серверним станом використовується `@tanstack/react-query` із кастомними хуками (наприклад, `useStructuresQuery`, `useFacultiesQuery`, `useCoursesQuery`, `useGroupsQuery`).

Інтернаціоналізація. Система підтримує дві мови: українську (UK) як мову за замовчуванням та англійську (EN), яка завантажується лише при виборі користувачем через функцію `ensureI18nResources`. Це забезпечує оптимізацію початкового завантаження - ресурси англійської мови не включаються до основного бандлу.

Відображення розкладу. Розклад відображається у компоненті `ScheduleModal`, який реалізований як висувна панель знизу (bottom sheet) за допомогою бібліотеки `Vaul Drawer`.

Функція поширення. При відкритому модальному вікні `mainButton` перемикається на режим «поширити» (share) та при натисканні викликає функцію

handleShareSchedule, яка формує текстове представлення розкладу та використовує TMA shareURL() для його поширення.

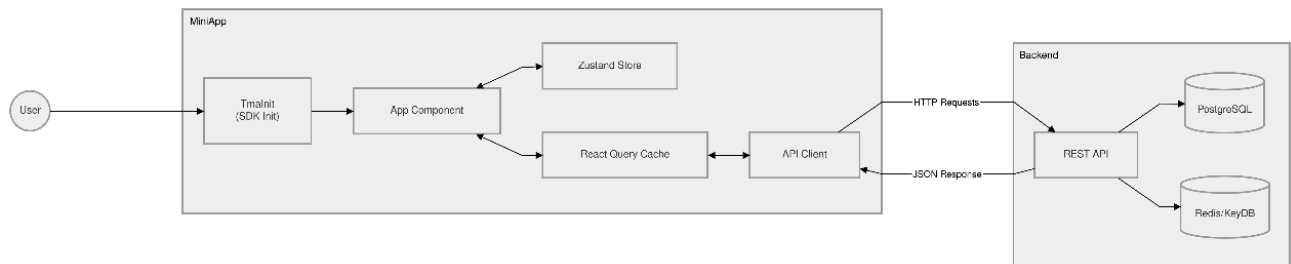


Рис. 3.7 Діаграма потоку даних

3.3 Архітектура серверної частини (API + Bot + Loader)

3.3.1 API Service (NestJS)

Серверний API побудований на базі фреймворку NestJS із використанням HTTP-адаптера Fastify, що забезпечує значно вищу пропускну здатність порівняно з Express. Архітектура модуля побудована за принципом модульної декомпозиції, де кожен функціональний домен інкапсульований у окремий NestJS-модуль.

Центральним елементом архітектури є AppModule, який реєструє глобальні компоненти: EnvalidModule для валідації змінних середовища, SentryModule для моніторингу помилок через Sentry, CacheModule для кешування через KeyDB, ClientsModule для двох черг RabbitMQ (feedback та language). TMA-middleware застосовується виключно до контролерів UserController та FeedbackController, оскільки лише ці ендпоінти потребують автентифікації через Telegram Mini App.

Точка входу API налаштовує Fastify-адаптер із підтримкою HTTPS, CORS, глобальний ValidationPipe для валідації вхідних даних та інтерактивну документацію API.

CORS-конфігурація явно дозволяє експозицію заголовка X-Data-Status (CACHE_HEADER), який використовується клієнтами для визначення того, чи отримані дані з кешу або є свіжими. ValidationPipe із параметрами whitelist: true та

transform: true автоматично видаляє невідомі поля із запитів та трансформує типи відповідно до DTO.

API побудований за трирівневою архітектурою: Controllers (рівень представлення) -> Services (рівень бізнес-логіки) -> Prisma (рівень доступу до даних). Функціональні модулі мають ієрархічні залежності: GroupService залежить від CourseService, який залежить від FacultyService, який у свою чергу залежить від StructureService. Ця ієрархія відображає предметну структуру навчального закладу.

3.3.2 Bot Service (grammY)

Telegram-бот побудований на базі фреймворку grammY та використовує runner-режим для асинхронної обробки оновлень. Архітектура бота базується на ланцюжку middleware, який організований наступним чином:

1. Трансформер вихідних запитів: indexOutgoingUpdate() індексує всі вихідні API-виклики до Elasticsearch для аналітики, autoRetry забезпечує автоматичне повторення запитів до 5 разів із максимальною затримкою 2500 секунд.

2. Глобальні middleware: indexIncomingUpdate індексує вхідні оновлення до Elasticsearch, commands реєструє обробники команд, myChatMemberEvent обробляє події зміни статусу бота у чаті.

3. Middleware приватних чатів: sequentialize забезпечує послідовну обробку оновлень від одного користувача, session зберігає сесію в KeyDB через RedisAdapter із TTL 3600 секунд у production та 30 секунд у development, і18n надає функції локалізації, далі йдуть обробники команд.

Bull-черги: бот використовує три черги (dailyAlertQueue, messageQueue, pollQueue) із конфігурацією обмеження швидкості 25 повідомлень за 2500 мілісекунд для запобігання перевищенню лімітів Telegram Bot API.

RabbitMQ-споживачі: два споживачі (feedback та language) обробляють повідомлення від API Service - доставку зворотного зв'язку адміністратору та інвалідацію сесій при зміні мови.

3.3.3 Loader Service

Loader Service є найпростішим компонентом архітектури, що виконує єдину задачу - періодичну синхронізацію даних про розклад та не зберігає дані самостійно - він лише ініціює звернення до API Service, який виконує конвеєр `fetch -> diff -> update`. Алгоритм обходу ієрархії виконується паралельно для двох гілок: `teachers` (структури -> кафедри -> викладачі) та `groups` (структури -> факультети -> курси -> групи) через `Promise.all`. Діапазон дат: від сьогодні + 30 днів:

3.4 Архітектура бази даних

База даних системи «ТТА ДУІКТ» реалізована на PostgreSQL 16 та описана за допомогою Prisma ORM із підтримкою мультифайлової схеми. Схема складається з 10 моделей та 2 перелічень, які повністю описують предметну область - структуру навчального закладу, розклад занять та налаштування користувачів.

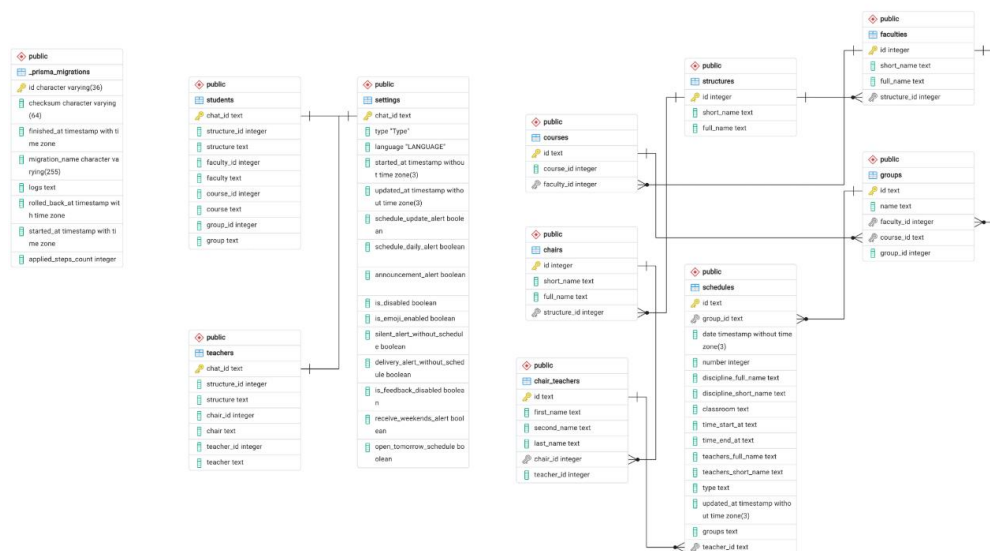


Рис. 3.8 ERD-діаграма бази даних

4 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ

4.1 Реалізація серверної частини (API)

4.1.1 TMA Authentication Middleware

Автентифікація запитів до захищених ендпоінтів API реалізована через спеціалізований NestJS-middleware, який валідує init-data Telegram Mini App. Цей middleware застосовується виключно до контролерів UserController та FeedbackController, оскільки решта ендпоінтів (розклад, ієрархія навчального закладу) є публічними та не потребують автентифікації.

Алгоритм роботи middleware:

1. Із заголовка Authorization витягуються стратегія та дані. Очікуваний формат: tma <init-data>.
2. Якщо стратегія не дорівнює "tma" або дані відсутні, генерується виняток ForbiddenException з кодом INVALID_STRATEGY.
3. Функція validate() із бібліотеки @tma.js/init-data-node перевіряє підпис init-data за допомогою токена бота (BOT_TOKEN), що гарантує автентичність даних - вони дійсно надіслані сервером Telegram.
4. При успішній валідації дані парсяться функцією parse() та зберігаються в об'єкт запиту як req.tma, що робить їх доступними для контролерів.
5. При помилці ExpiredError (дані прострочені) генерується відповідний виняток STRATEGY_DATA_EXPIRED.

Тип ModifiedFastifyRequest розширює стандартний FastifyRequest Fastify, додаючи поле tma: InitData до raw-запиту, що забезпечує типобезпечний доступ до автентифікаційних даних у контролерах.

4.1.2 Cache Interceptor

Кешування відповідей API реалізоване через NestJS-перехоплювач (interceptor), який реалізує шаблон Cache-Aside із використанням KeyDB як сховища кешу та має наступну логіку:

1. Ключ кешу: використовується `originalUrl` запиту, що включає шлях та параметри запиту. Це забезпечує унікальність кешу для різних комбінацій параметрів.

2. Перевірка кешу: спочатку виконується спроба отримати дані з KeyDB. Якщо дані знайдено, вони повертаються через `of()` - RxJS-оператор, що створює Observable з одним значенням.

3. TTL-стратегія: якщо дані відсутні в кеші, виконується звернення до сервісу через `next.handle()`, і результат зберігається в кеш із TTL 30 хвилин для непорожніх відповідей та 5 хвилин для порожніх. У development-середовищі TTL становить 5 секунд для зручності розробки.

4. Заголовок X-Data-Status: кожна відповідь супроводжується заголовком, який вказує на статус даних: "fresh" (дані отримані від зовнішнього API університету) або "cached" (дані з кешу або локальної бази). Це дозволяє клієнтам відображати відповідні індикатори користувачу.

4.1.3 Schedule Service з логікою порівняння

Сервіс розкладу є найскладнішим компонентом бізнес-логіки API. Він реалізує повний конвеєр отримання, валідації, порівняння та оновлення даних розкладу. Конвеєр обробки розкладу включає наступні етапи:

1. Перевірка існування сутності: метод `isEntityExists()` перевіряє, чи група або викладач існують у локальній базі. Якщо ні - повертається порожній результат.

2. Отримання даних з API університету: виконується POST-запит до зовнішнього API. При помилці або недоступності API прапорець `isRequestSuccess` встановлюється в `false`.

3. Валідація відповіді: бібліотека `io-ts` забезпечує runtime-перевірку структури відповіді API. Схема `responseSchema` описує очікуваний формат JSON,

включаючи дати, уроки та періоди. Якщо відповідь не відповідає схемі, повертаються кешовані дані з бази.

4. Трансформація даних: метод `transformData()` перетворює відповідь API у внутрішній формат, зокрема конвертуючи `dateUpdated` у зону `Europe/Kyiv` за допомогою `date-fns-tz`.

5. Порівняння (diff): бібліотека `deeper-object-diff` порівнює поточні дані з бази та нові дані з API. Якщо різниці немає - повертаються існуючі дані без запису.

6. Атомарне оновлення: при наявності змін виконується Prisma-транзакція `$transaction`, яка атомарно видаляє старі записи та створює нові в межах обраного діапазону дат. Це гарантує консистентність даних.

4.1.4 Feedback Service з RabbitMQ RPC

Сервіс зворотного зв'язку реалізує асинхронну доставку повідомлень від користувачів до адміністратора через шаблон RPC (Request-Reply Pattern) із використанням RabbitMQ та спочатку перевіряє, чи не заблоковано функцію зворотного зв'язку для конкретного користувача (поле `isFeedbackDisabled` у `Setting`). Далі валідує довжину повідомлення (максимум 2000 символів). Потім відправляє повідомлення до черги RabbitMQ через `ClientProxy.send()`, що реалізує RPC-шаблон - очікує відповідь від споживача (бота) із таймаутом 5 секунд. Якщо доставка не вдалася або таймаут перевищено, повертається `{ isDelivered: false }`.

4.1.5 User Service з управлінням ідентичністю

Сервіс користувача реалізує операції CRUD для управління ідентичністю та налаштуваннями:

```
// apps/api/src/modules/user/user.service.ts
async patchUser(
  req: ModifiedFastifyRequest,
  body: PatchUserBodyDto,
): Promise<GetStudentResponseDto | GetTeacherResponseDto> {
  const { id } = this.tmaObject(req.raw);
  const chatId = String(id);
  const { type: payloadType, ...data } = body.payload;

  try {
    if (payloadType === Type.STUDENT && "groupId" in data) {
```

```

const payload = body.payload as PatchStudentBodyDto;

const [student, settings] = await this.prismaService.$transaction([
  this.prismaService.student.upsert({
    where: { chatId },
    update: {
      ...data,
      settings: { update: { type: payload.type, updatedAt: new Date() } },
    },
    create: {
      ...data,
      settings: {
        connectOrCreate: { where: { chatId }, create: { chatId, type:
payload.type } },
      },
    },
    include: { settings: { select: this.selectedSettings } },
  }),
  this.prismaService.setting.upsert({
    where: { chatId },
    create: { chatId, type: payload.type },
    update: { type: payload.type, updatedAt: new Date() },
    select: { type: true, language: true, startedAt: true },
  }),
  this.prismaService.teacher.deleteMany({ where: { chatId } }),
]);

return plainToInstance(GetStudentResponseDto, { type: settings.type,
startedAt: settings.startedAt, ...student });
}
// ... аналогічна логіка для TEACHER
} catch (err) {
  throw new
InternalServerErrorException(INTERNAL_REQUEST_CODES.DATABASE_UNAVAILABLE);
}
}

```

Операція зміни ідентичності виконується в рамках Prisma-транзакції, яка атомарно: upsert студента/викладача, upsert налаштувань та видаляє запис попередньої ролі (teacher.deleteMany при створенні студента і навпаки). Це забезпечує консистентність - користувач може мати лише одну активну роль. При зміні налаштувань мови відправляється RPC-повідомлення через RabbitMQ для інвалідації сесії бота у KeyDB.

4.2 Реалізація клієнтської частини (Mini App)

4.2.1 Потік ідентифікації

Сторінка ідентифікації (Identity.tsx) надає користувачеві вибір ролі через два інтерактивних картки з SVG-іконками:

```
// apps/miniapp/src/pages/Identity.tsx
export const IdentityPage = (): ReactNode => {
  const { t } = useTranslation();
  const navigation = useNavigate();

  const handleClickOnCard = (card: UserType) => {
    const pathname = card === "STUDENT" ? NavigationRoutes.IDENTITY_STUDENT :
NavigationRoutes.IDENTITY_TEACHER;
    return navigation({ pathname });
  };

  const cards: Array<{ type: UserType; label: LangKeys }> = [
    { type: "STUDENT", label: "general.student" },
    { type: "TEACHER", label: "general.teacher" },
  ];

  return (
    <div className="flex text-center justify-between min-h-dvh bg-cover bg-
[url(/background.png)]">
      <div className="m-auto">
        <Title level="1" weight="3" className="text-white">
          {t("identity_page.title")}
        </Title>
        <div className="flex space-x-5">
          {cards.map((card, i) => (
            <Card type="plain" onClick={() => handleClickOnCard(card.type)}>
              {card.type === "STUDENT" && <StudentSvg className="identity-card-svg"
/>}

              {card.type === "TEACHER" && <TeacherSvg className="identity-card-svg"
/>}

              <CardCell>{t(card.label)}</CardCell>
            </Card>
          ))}
        </div>
      </div>
    </div>
  );
};
```

4.2.2 Сторінка розкладу

Сторінка розкладу використовує react-day-picker у режимі вибору діапазону (mode="range") з максимальним діапазоном 30 днів:

```
// apps/miniapp/src/pages/Schedule.tsx
export const SchedulePage = (): ReactNode => {
  const [selectedDate, setSelectedDate] = useState<DateRange>();
  const [isModalOpen, setIsModalOpen] = useState(false);
```

```

return (
  <Section footer={t("schedule_page.footer")}>
    <DatePicker
      mode="range"
      locale={settings?.language === "EN" ? enUS : uk}
      weekStartsOn={1}
      max={30}
      selected={selectedDate}
      onSelect={handleClickOnDate}
      showOutsideDays
      animate
      classNames={{
        selected: cn(borderStyle, "bg-[#1EA5E9] text-white"),
        today: cn("rounded-md", todayButtonStyle),
        range_start: cn(borderStyle, "rounded-l-md"),
        range_end: cn(borderStyle, "rounded-r-md"),
        range_middle: "bg-[#1EA5E9]",
      }}
    />
    <ScheduleModal isModalOpen={isModalOpen}
      handleModalChange={handleModalChange} selectedDate={selectedDate} />
  </Section>
);
};

```

4.2.3 Сторінка налаштувань

Сторінка налаштувань динамічно генерує перемикачі на основі об'єкта налаштувань із серверу, фільтруючи системні ключі:

```

// apps/miniapp/src/pages/Settings.tsx
export const SettingsPage = (): ReactNode => {
  const updateSetting = async (key: CamelCaseKeys | "language", value: string |
boolean) => {
    const { error, data } = await client.PATCH("/user/settings", {
      params: { header: { Authorization: `tma ${initDataRaw}` } },
      body: { [key]: value },
    });

    if (data) {
      if (key === "language" && typeof value === "string") {
        await ensureI18nResources(value);
        await i18next.changeLanguage(value);
      }
      setSettings(data);
    }
  };

  return (
    <Section footer={t("settings_page.footer")}>
      <Select
        header={t("settings_page.selector_title")}
        value={settings.language}
        onChange={(event) => updateSetting("language", event.target.value)}
      >
        <option value={"EN"}>English</option>
        <option value={"UK"}>Українська</option>
      </Select>
    </Section>
  );
};

```

```

    {Object.entries(settings)
      .filter(([key]) => !SYSTEM_SETTINGS_KEYS.includes(key as SystemKeys))
      .map(([key, value]) => (
        <Cell
          Component="label"
          before={<Checkbox checked={Boolean(value)} onChange={() =>
updateSetting(key as CamelCaseKeys, !value)} />}
          description={t(getSettingLocalization(key, "description"))}
        >
          {t(getSettingLocalization(key, "title"))}
        </Cell>
      ))}
  </Section>
);
};

```

4.2.4 Сторінка зворотного зв'язку

Сторінка зворотного зв'язку реалізує інтерактивну форму з обмеженням довжини повідомлення та станоцентричним управлінням кнопкою mainButton:

```

// apps/miniapp/src/pages/Feedback.tsx
export const FeedbackPage = (): ReactNode => {
  const [message, setMessage] = useState("");

  const sendFeedback = useCallback(async () => {
    if (message.length > MAX_FEEDBACK_MESSAGE_LENGTH) return;

    const { error, data } = await client.POST("/feedback", {
      params: { header: { Authorization: `tma ${initDataRaw}` } },
      body: { message },
    });

    if (data) {
      const { isDelivered } = data;
      toast({
        description: isDelivered ? t("feedback_page.feedback_delivered") :
t("feedback_page.feedback_not_delivered"),
      });
      if (isDelivered) return setMessage("");
    }
  }, [initDataRaw, message]);

  useEffect(() => {
    if (settings?.isFeedbackDisabled) {
      mainButton.setParams({ bgColor: "#ED4245", text:
t("feedback_page.feedback_not_allowed") });
      return;
    }

    const isFeedbackTooLong = message.length > MAX_FEEDBACK_MESSAGE_LENGTH;

    if (isFeedbackTooLong && mainButton.isEnabled()) {
      mainButton.disable();
      mainButton.setParams({ bgColor: "#ED4245", text:
t("feedback_page.feedback_too_long") });
    }

    const offClick = mainButton.onClick(sendFeedback);
    return () => {

```

```

        offClick();
    };
    }, [message.length, sendFeedback, settings?.isFeedbackDisabled]);

    return (
        <Section footer={t("feedback_page.footer", { max: MAX_FEEDBACK_MESSAGE_LENGTH
    }}}>
        <Textarea
            maxLength={MAX_FEEDBACK_MESSAGE_LENGTH}
            header={t("feedback_page.title")}
            value={message}
            onChange={({ target }) => setMessage(target.value)}
            style={{ height: "35vh" }}
            autoFocus
        />
    </Section>
    );
};

```

4.3 Реалізація Telegram-бота

4.3.1 Конфігурація Bull-черг

Telegram Bot API має суворі обмеження на кількість повідомлень: не більше 30 повідомлень на секунду для ботів, та 20 повідомлень на хвилину для конкретного чату. Для дотримання цих лімітів використовується система черг Bull із налаштованим rate limiter:

```

// apps/bot/src/helpers/bullConfig.ts
export const bullConfig: QueueOptions = {
    redis: { host: env.KEYDB_HOST, port: env.KEYDB_PORT, password: env.KEYDB_PASSWORD
},
    limiter: { max: 25, duration: 2500 },
    defaultJobOptions: {
        delay: 100,
        removeOnComplete: true,
        removeOnFail: false,
        backoff: { type: "exponential", delay: 1000 },
    },
    settings: { retryProcessDelay: 2500 },
};

```

4.3.2 Менеджер черги щоденних сповіщень

Черга щоденних сповіщень обробляє завдання відправки повідомлень з розкладом на наступний день:

```

// apps/bot/src/queues/dailyAlertQueueManager.ts
export const dailyAlertQueueManager = new
Queue<DailyAlertQueueInput>("DAILY_ALERT_QUEUE_MANAGER", bullConfig);

dailyAlertQueueManager.process(async ({ data }) => {
    const { settings, isSilentAlert, message, preparedEntities } = data;

    const entities: MessageEntity[] = preparedEntities.map((entity) => ({
        type: "bold",

```

```

    offset: message.indexOf(entity),
    length: entity.length,
  }));

const resultText = new FormattedString(message)
  .plain("\n\n")
  .italic(i18n.t(settings.language, "daily_alert_delivery_footer"));

const keyboard = new InlineKeyboard().webApp(
  i18n.t(settings.language, "open_schedule_label"),
  `${env.MINIAPP_URL}#schedule`,
);

try {
  await bot.api.sendMessage(settings.chatId, resultText.toString(), {
    entities: [...resultText.entities, ...entities],
    disable_notification: isSilentAlert,
    reply_markup: keyboard,
  });
} catch (err) {
  console.error(err);
  await queueErrorHandler(err, settings.chatId);
}
});

```

4.4 Реалізація Loader-сервісу

Loader-сервіс реалізує механізм фоновій синхронізації даних, який кожні 30 хвилин обходить повну ієрархію навчального закладу та ініціює оновлення розкладу:

```

// apps/loader/src/app.ts
async function mainCron() {
  console.log(`START LOADER: ${new Date().toISOString()}`);
  const startLoaderTime = performance.now();

  const monthDates = calculateMonthDates(new Date());

  try {
    const structures = await loadStructures();

    const loadTeachersData = async () => {
      for (const structure of structures) {
        const chairs = await loadChairs(structure.id);
        for (const chair of chairs) {
          const teachers = await loadTeachersByChair(structure.id, chair.id);
          for (const teacher of teachers) {
            await loadTeacherSchedule(teacher.id, monthDates.startAt,
monthDates.endAt);
          }
        }
      }
    };
  }
}

```

```

    }
  }
};

const loadGroupsData = async () => {
  for (const structure of structures) {
    const faculties = await loadFaculties(structure.id);
    for (const faculty of faculties) {
      const courses = await loadCourses(structure.id, faculty.id);
      for (const course of courses) {
        const groups = await loadGroups(structure.id, faculty.id,
course.course);
        for (const group of groups) {
          await loadGroupSchedule(group.id, monthDates.startAt,
monthDates.endAt);
        }
      }
    }
  }
};

await Promise.all([loadTeachersData(), loadGroupsData()]);
} catch (err) {
  console.error(err);
}

const endLoaderTime = performance.now();
console.log(`LOADER DONE: ${((endLoaderTime - startLoaderTime) / 1000 / 60}
MIN.`);
}
schedule("*/30 * * * *", () => mainCron(), { timezone: "Europe/Kyiv" });

```

Алгоритм обходу ієрархії:

1. Завантаження структур: отримання списку навчальних структур через API.
2. Паралельна обробка двох гілок через Promise.all:

- гілка викладачів: структури -> кафедри -> викладачі кафедри -> розклад кожного викладача.

- гілка груп: структури -> факультети -> курси -> групи -> розклад кожної групи.

3. Послідовний обхід: всередині кожної гілки обробка виконується послідовно (for...of), щоб не перевантажувати API університету паралельними запитами.

Кожен виклик `loadGroupSchedule` або `loadTeacherSchedule` ініціює GET-запит до `API Service`, який у свою чергу виконує конвеєр `fetch -> diff -> update`. `Loader` не зберігає та не обробляє дані самостійно - вся логіка порівняння та оновлення інкапсульована в `API Service`.

4.5 Інтеграція та взаємодія компонентів

Взаємодія між компонентами системи «ТТА DUIKT» організована через декілька комунікаційних шаблонів, кожен з яких обраний відповідно до характеру взаємодії.

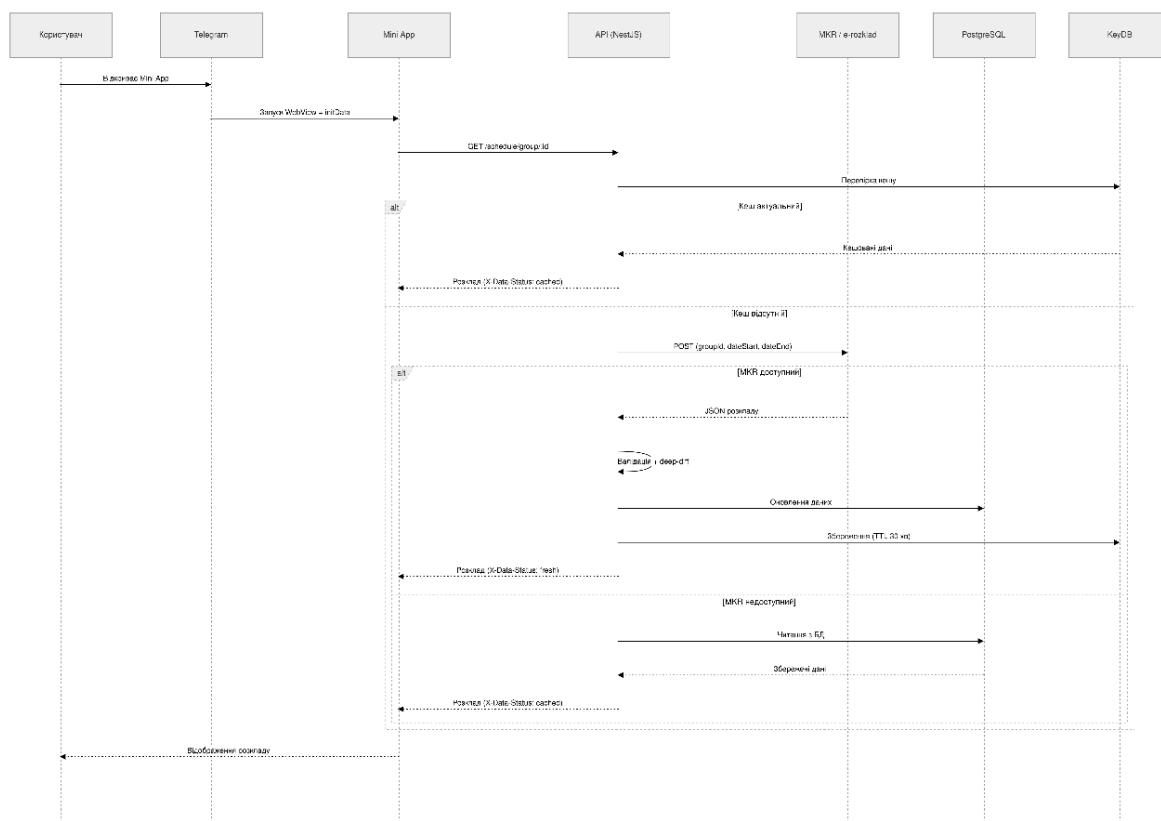


Рис. 4.1 Діаграма послідовності перегляду розкладу

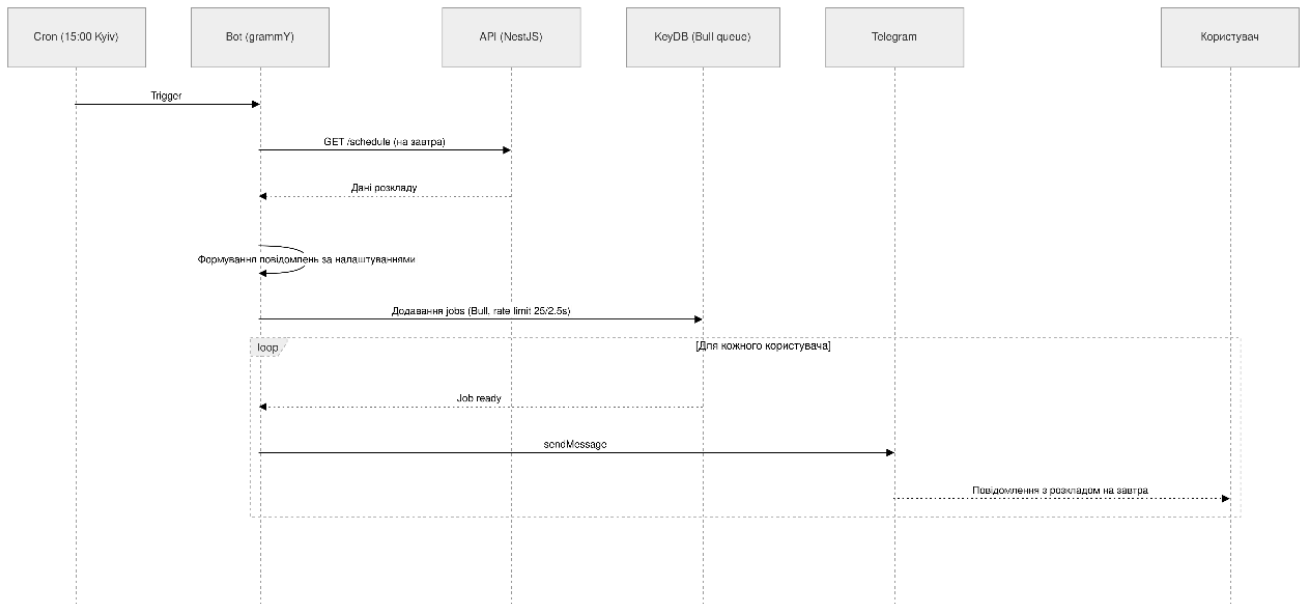


Рис. 4.2 Діаграма послідовності щоденного сповіщення

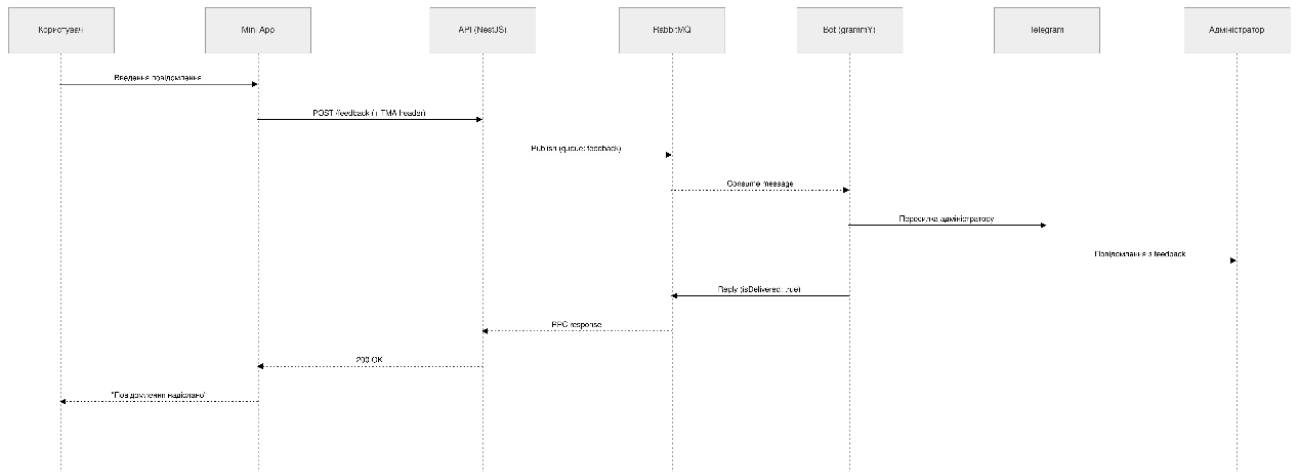


Рис. 4.3 Діаграма послідовності зворотного зв'язку

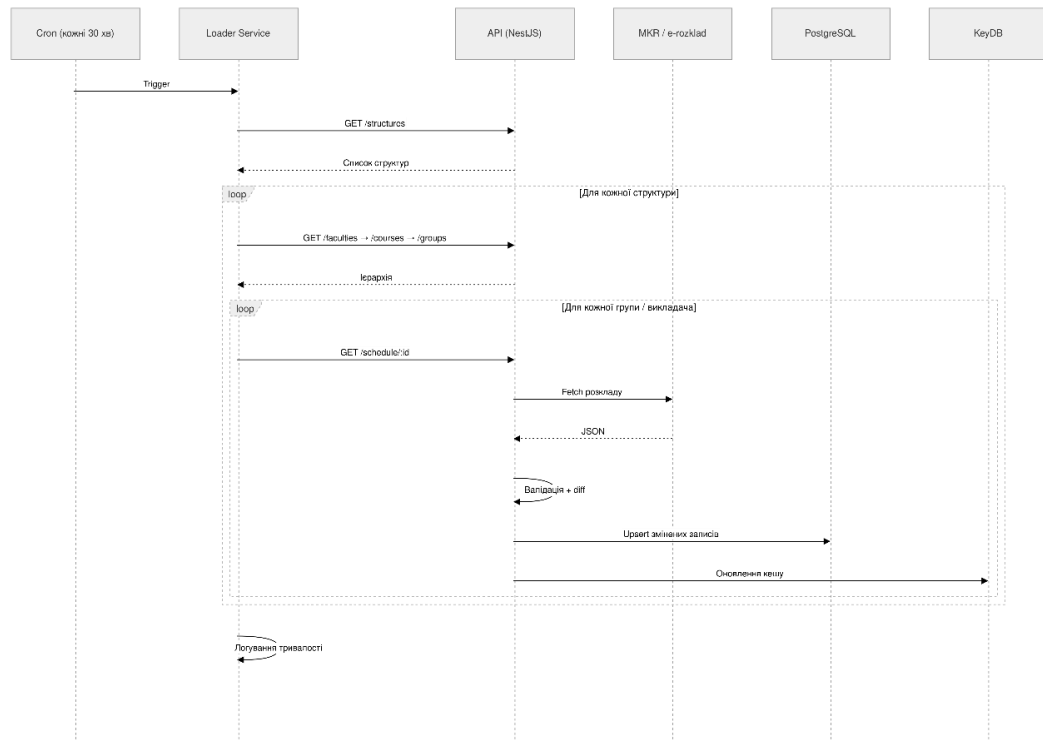


Рис. 4.4 Діаграма послідовності актуалізації розкладу

4.6 Тестування застосунку

Тестування системи «ТТА DUIKT» виконувалось комплексно з використанням функціонального тестування API-ендпоінтів, перевірки бізнес-логіки бота та інтеграційного тестування наскрізних сценаріїв взаємодії між компонентами.

4.6.1 Функціональне тестування API

Тестування API виконувалось через HTTP-клієнт із перевіркою коректності відповідей, кодів статусу та заголовків. Основні тестові сценарії:

Модуль User:

- GET /user/me (знайдений користувач): перевірка, що відповідь містить дані студента або викладача разом із налаштуваннями, тип відповідає GetStudentResponseDto або GetTeacherResponseDto, статус 200.

- GET /user/me (користувач не знайдений): перевірка, що повертається статус 404 із повідомленням USER_NOT_FOUND при валідному ТМА-токені для нового користувача.

- GET /user/me (невалідна стратегія): перевірка, що повертається статус 403 із повідомленням INVALID_STRATEGY при відсутньому або некоректному заголовку Authorization.

- PATCH /user/me (створення студента): перевірка, що при відправці payload з типом STUDENT створюються записи Student та Setting, видаляється запис Teacher (якщо існує), статус 200.

- PATCH /user/me (створення викладача): аналогічно для типу TEACHER, з видаленням Student.

- PATCH /user/settings: перевірка оновлення кожного булевого параметра та мови, валідація RabbitMQ-повідомлення інвалідації сесії.

Модуль Schedule:

- GET /schedule/group/:groupId: перевірка отримання розкладу групи з валідними даними, заголовок X-Data-Status встановлено коректно.

- GET /schedule/teacher/:teacherId: аналогічно для викладача.

- валідація діапазону дат: перевірка, що діапазон > 31 день повертає статус 400 із повідомленням MAX_DAYS_EXCEEDED.

- поведінка кешу: перший запит повертає X-Data-Status: fresh, повторний - X-Data-Status: cached.

- поведінка при недоступності зовнішнього API: перевірка, що повертаються кешовані дані з бази з isRequestSuccess: false.

- GET /health/status: перевірка доступності сервісу, статус 200.

4.6.2 Тестування Telegram-бота

Команда /start: перевірка, що бот відповідає привітальним повідомленням із кнопкою відкриття Mini App.

ВИСНОВКИ

У рамках виконання кваліфікаційної роботи бакалавра було спроектовано та реалізовано мікросервісну систему «ТТА DUIKT», призначену для агрегації, кешування та персоналізованого доставлення інформації про розклад навчальних занять учасникам освітнього процесу Державного університету інформаційно-комунікаційних технологій. Систему побудовано як сукупність слабо зв'язаних сервісів, що взаємодіють через брокер повідомлень та спільну базу даних, із клієнтським інтерфейсом у вигляді Telegram-бота та вбудованого Mini App. За результатами проведеної роботи можна сформулювати такі висновки.

1. Проаналізовано предметну область організації доступу до розкладу навчальних занять університету та інформування учасників освітнього процесу. Встановлено, що основними проблемами є: монолітна залежність від веб-системи МКР, нестабільність інфраструктури в умовах аварійних відключень електроенергії, а також відсутність проактивних механізмів доставлення змін розкладу до кінцевих користувачів.

2. Виконано огляд існуючих програмних рішень: офіційний веб-доступ через систему МКР, офіційний мобільний застосунок МКР (середній рейтинг 2,7 із 5), а також сторонній Android-клієнт Skedy. Виявлено ключові недоліки наявних рішень: обмежена автономність роботи за тривалої недоступності сервера МКР, слабка або відсутня серверна агрегація даних, недостатня персоналізація сповіщень та фрагментованість користувацького досвіду.

3. Обрано та обґрунтовано технологічний стек реалізації: TypeScript спільно з Node.js як єдина мова розроблення для всіх компонентів системи, NestJS із HTTP-адаптером Fastify - для побудови API-сервісу, фреймворк grammY - для реалізації Telegram-бота, React у поєднанні зі збирачем Vite - для Mini App, PostgreSQL із Prisma ORM - для зберігання даних, KeyDB - для кешування, RabbitMQ та Bull - для асинхронного обміну повідомленнями, Sentry, Elasticsearch і Kibana - для моніторингу та логування, Docker - для контейнеризації та розгортання.

4. Спроектовано мікросервісну архітектуру системи, що складається з чотирьох сервісів (API, Bot, Loader, Mini App) та двох спільних пакетів (prisma, types), організованих як npm-монорепозиторій. Розроблено проєктну документацію із застосуванням UML-діаграм: діаграми діяльності, діаграми варіантів використання, діаграми розгортання, діаграми пакетів, діаграми класів та ER-діаграми бази даних.

5. Реалізовано основну функціональність системи, що включає:

- криптографічну автентифікацію Telegram Mini App [8] (ТМА) з валідацією підпису initData.
- патерн cache-aside із використанням KeyDB (TTL 30 хвилин для даних, 5 хвилин для порожніх відповідей).
- механізм виявлення змін розкладу на основі бібліотеки deep-object-diff із транзакційним оновленням бази даних.
- контрольоване доставлення сповіщень через черги Bull із обмеженням швидкості 25 повідомлень за 2,5 секунди.
- щоденні сповіщення о 15:00 за київським часом (з неділі по четвер - розклад на наступний день, у п'ятницю - на суботу).
- понад 8 параметрів персоналізації, синхронізованих між Mini App та ботом через RabbitMQ.
- ієрархічне каскадне оновлення даних, що ініціюється сервісом Loader кожні 30 хвилин.
- двомовну підтримку інтерфейсу: Project Fluent для бота та i18next для Mini App.
- адміністративні функції: розсилання оголошень, створення опитувань, управління зворотним зв'язком.
- контейнеризацію всіх сервісів за допомогою Docker із резервним копіюванням бази даних до S3 кожні 12 годин.

6. Проведено функціональне тестування всіх кінцевих точок API, команд бота, сценаріїв взаємодії з Mini App та міжсервісної інтеграції. Усі протестовані кінцеві точки повертають очікувані статуси та коректні відповіді.

7. Система обслуговує близько 1000 щоденних активних користувачів у ДУІКТ, що підтверджує практичну цінність обраного підходу.

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

1. Кугаївський А.О., Залива В. В. Визначення вимог до телеграм-боту для забезпечення надійного доступу до розкладу навчальних занять. II Всеукраїнська науково-технічна конференція «Виклики та рішення в програмній інженерії», 26 листопада 2025 року, Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. - К: ДУІКТ, 2025. С. 160-163.

2. Кугаївський А.О., Залива В. В. Визначення вимог до системи агрегації розкладу з підтримкою автономної роботи. VII Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 23 квітня 2026 року, Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. - К: ДУІКТ, 2026. С. 463-466.

ПЕРЕЛІК ПОСИЛАНЬ

1. Node.js - About Node.js. Node.js. URL: <https://nodejs.org/en/about>.
2. TypeScript Documentation. TypeScript. URL: <https://www.typescriptlang.org/docs>.
3. pnpm - Motivation. pnpm. URL: <https://pnpm.io/motivation>.
4. NestJS Documentation. NestJS. URL: <https://docs.nestjs.com>.
5. Fastify Documentation. Fastify. URL: <https://fastify.dev/docs/latest>.
6. grammY - The Telegram Bot Framework. grammY. URL: <https://grammy.dev>.
7. Telegram Bot API. Telegram. URL: <https://core.telegram.org/bots/api>.
8. Telegram Mini Apps. Telegram. URL: <https://core.telegram.org/bots/webapps>.
9. React Documentation. React. URL: <https://react.dev>.
10. Vite Guide. Vite. URL: <https://vitejs.dev/guide>.
11. PostgreSQL Documentation. PostgreSQL. URL: <https://www.postgresql.org/docs>.
12. Prisma Documentation. Prisma. URL: <https://www.prisma.io/docs>.
13. KeyDB Documentation. KeyDB. URL: <https://docs.keydb.dev>.
14. RabbitMQ Documentation. RabbitMQ. URL: <https://www.rabbitmq.com/documentation.html>.
15. Bull - Redis-based queue for Node. GitHub. URL: <https://github.com/OptimalBits/bull>.
16. Elasticsearch Reference. Elastic. URL: <https://www.elastic.co/guide/en/elasticsearch/reference/current/index.html>.
17. Kibana Guide. Elastic. URL: <https://www.elastic.co/guide/en/kibana/current/index.html>.
18. Sentry Documentation. Sentry. URL: <https://docs.sentry.io>.
19. Docker Documentation. Docker. URL: <https://docs.docker.com>.
20. JavaScript Guide - MDN Web Docs. Mozilla. URL: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide>.

21. i18next Documentation. i18next. URL: <https://www.i18next.com>.
22. Fluent Project. Mozilla. URL: <https://projectfluent.org>.
23. deep-object-diff - npm. npm. URL: <https://www.npmjs.com/package/deep-object-diff>.
24. Цифрова трансформація освіти і науки. Міністерство освіти і науки України. URL: <https://mon.gov.ua/osvita-2/tsifrova-transformatsiya-osviti-i-nauki>.
25. Більше половини українців використовують Telegram як основне джерело інформації - дослідження KMIC. Detector Media. URL: <https://detector.media/infospace/article/244490>.
26. Design and implementation of an android and web-based university timetable customization system. Bayero Journal of Pure and Applied Sciences. DOI: 10.4314/bajopas.v10i1.50. URL: <https://www.ajol.info/index.php/bajopas/article/view/162301>.
27. Расписание занятий МКР. App Store. URL: <https://apps.apple.com/us/app/%D1%80%D0%B0%D1%81%D0%BF%D0%B8%D1%81%D0%B0%D0%BD%D0%B8%D0%B5-%D0%B7%D0%B0%D0%BD%D1%8F%D1%82%D0%B8%D0%B9-%D0%BC%D0%BA%D1%80/id1382175045>.

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ



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



Розробка застосунку для агрегації та відображення корпоративного розкладу

Виконав студент 4 курсу

групи ПД-41

Кугаївський Андрій Олександрович

Керівник роботи

доктор філософії (PhD) Залива Віталій Вікторович

Київ – 2026

МЕТА, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи - підвищення доступності до розкладу занять для користувачів за рахунок розробки розподіленої системи з серверною агрегацією даних, кешуванням та інтеграцією з платформою Mini Apps.

Об'єкт дослідження - процес організації доступу студентів та викладачів до розкладу навчальних занять в умовах нестабільної роботи інформаційних систем університету.

Предмет дослідження - методи, моделі та програмні засоби розробки розподілених систем для агрегації, кешування та відображення даних розкладу через платформу Mini Apps.

ЗАДАЧІ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1. Аналіз предметної галузі.
2. Аналіз існуючих аналогів.
3. Визначення вимог до програмного забезпечення.
4. Огляд та аналіз засобів реалізації розподіленої системи агрегації розкладу.
5. Проєктування мікросервісної архітектури застосунку.
6. Реалізація функціональності застосунку.
7. Тестування реалізованого функціоналу.

3

АНАЛІЗ АНАЛОГІВ

Критерій	МКР	Skedy	ТТА DUKT
Платформа	IOS + Android	Android	Telegram
Необхідність встановлення	Так	Так	Ні
Джерело даних	МКР API	МКР API	МКР API + Власний кеш
Автономна робота	Відсутня	Часткова	Присутня
Сповіщення про розклад та зміни у розкладі	Ні	Ні	Так
Персоналізація	Відсутня	Часткова	Присутня
Система кешування	Відсутня	Часткова	Присутня

4

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні вимоги:

1. Перегляд розкладу за обраний період через Mini Apps.
2. Ідентифікація ролі користувача в системі.
3. Щоденні сповіщення про заняття на наступний день.
4. Сповіщення про зміни у розкладі.
5. Збереження налаштувань користувача.

Нефункціональні вимоги:

1. Забезпечення доступу до розкладу при недоступності МКР завдяки кешуванню у KeyDB та персистентному збереженню у PostgreSQL.
2. Обробка сповіщень через RabbitMQ для запобігання втрати інформації.
3. Мікросервісна архітектура має дозволяти незалежне горизонтальне масштабування через Docker-контейнери без зміни доменної логіки.
4. Система має обслуговувати не менше 1000 одночасних активних користувачів, час відповіді API на запит не повинен перевищувати 300 мс.
4. Інтегрований Sentry у всі мікросервіси для централізованого збору логів.
5. Docker контейнеризація із застосуванням CI/CD на базі GitHub Actions.
6. Підтримка Mini Apps на всіх пристроях і операційних системах.

5

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

Backend:

1. NestJS - фреймворк для Node.js.
2. grammY - фреймворк для Telegram Bot.
3. Prisma - ORM для роботи з базою.



Frontend:

1. React - бібліотека для побудови UI.
2. Zustand - управління станом системи.



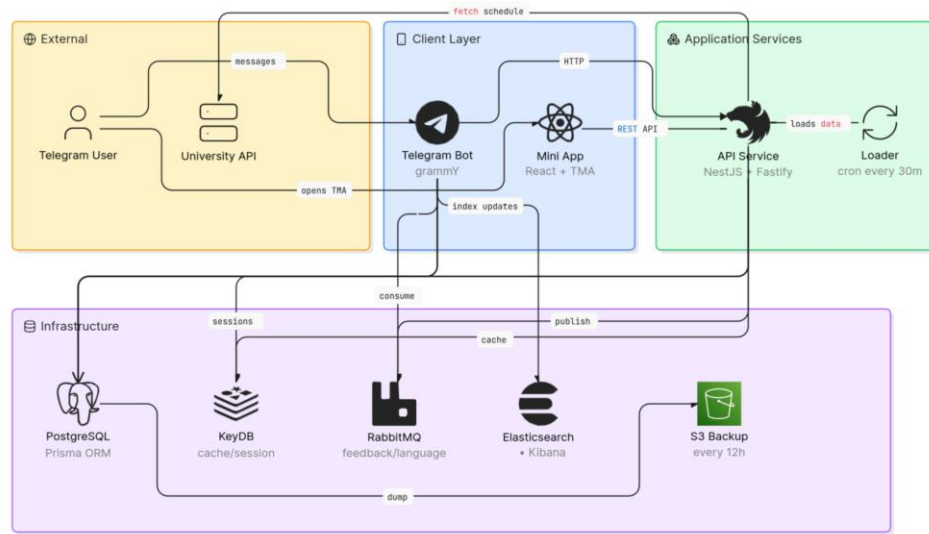
Інфраструктура:

1. PostgreSQL + KeyDB.
2. RabbitMQ + BullMQ.
3. Elasticsearch, Kibana та Sentry.
4. Docker + k3s.



6

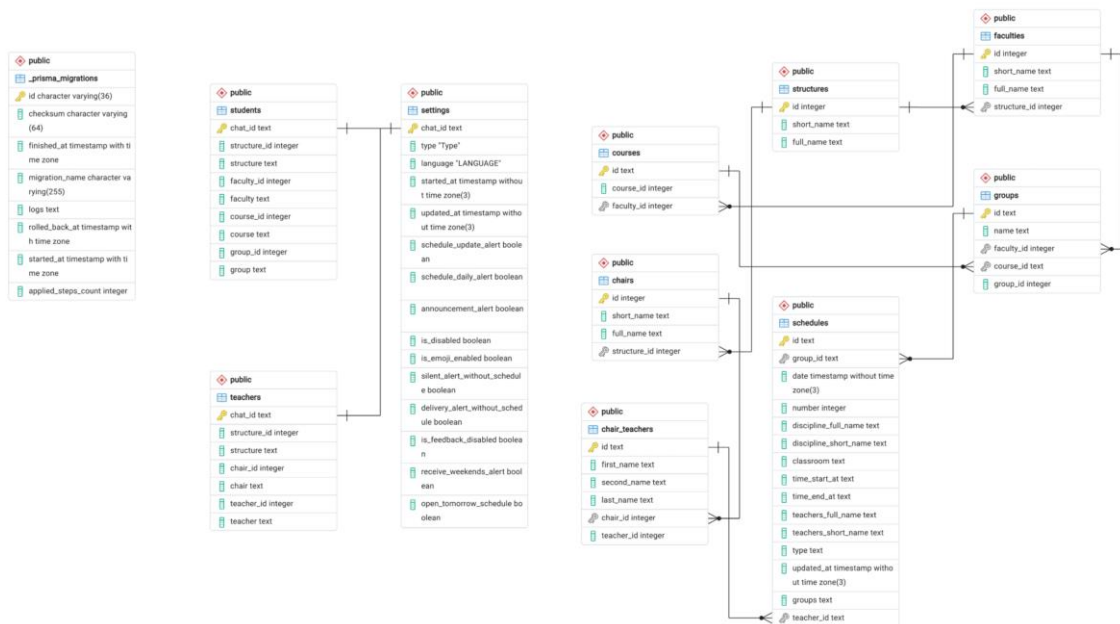
ДІАГРАМА АРХІТЕКТУРИ ПРОГРАМНОЇ СИСТЕМИ



eraser

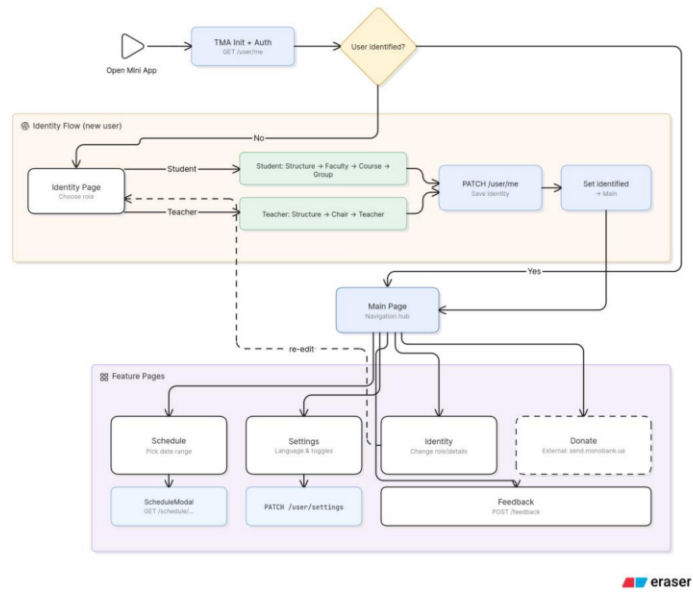
7

ERD-ДІАГРАМА БАЗИ ДАНИХ



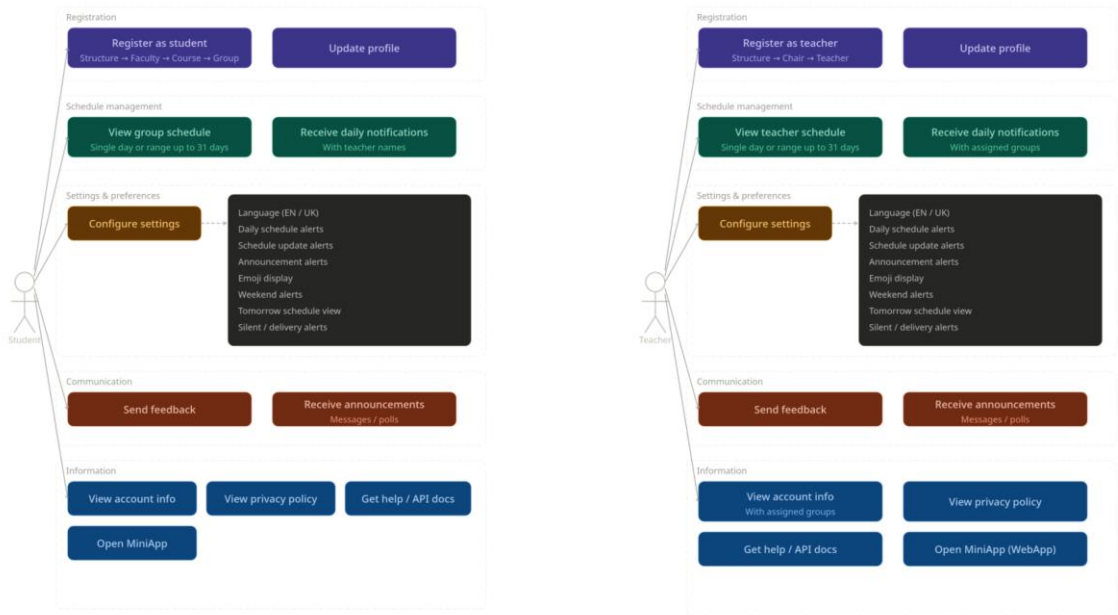
8

ДІАГРАМА КОРИСТУВАЦЬКОГО ШЛЯХУ

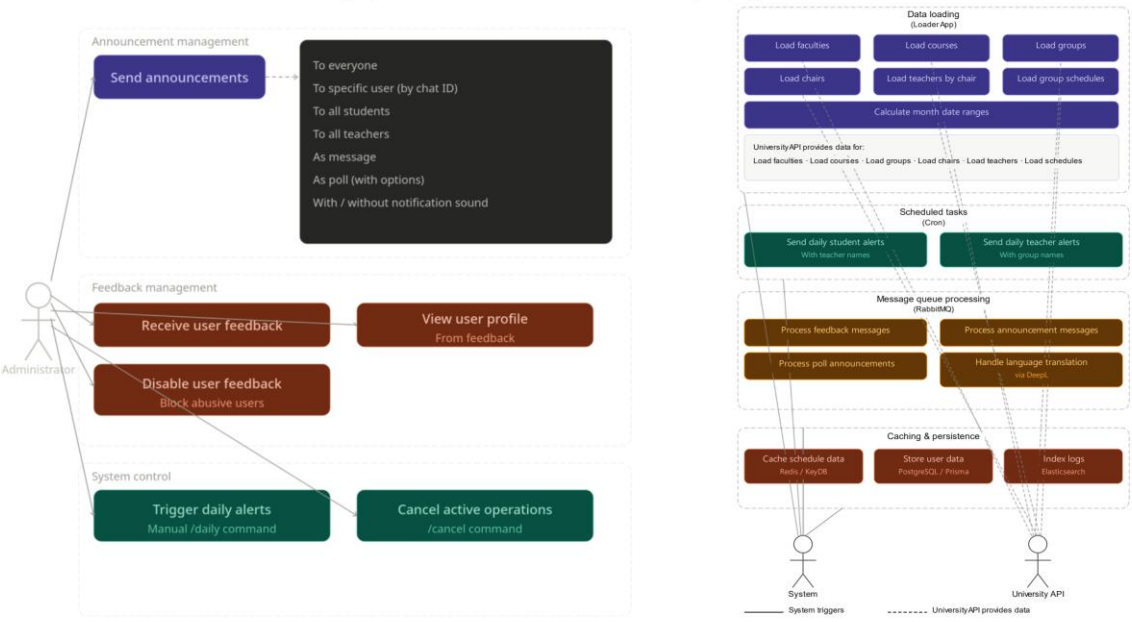


eraser

ДІАГРАМА ПРЕЦЕДЕНТІВ

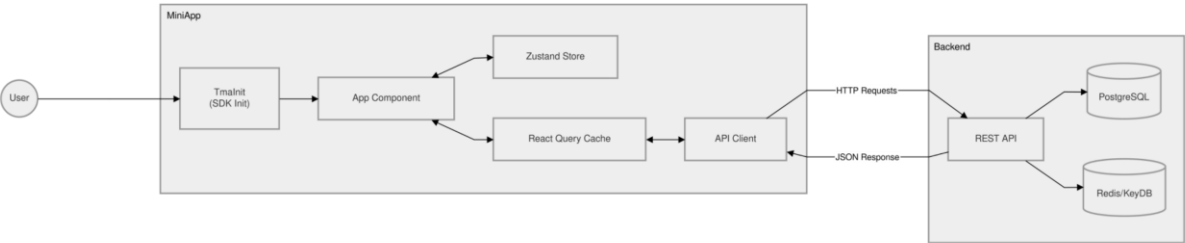


ДІАГРАМА ПРЕЦЕДЕНТІВ



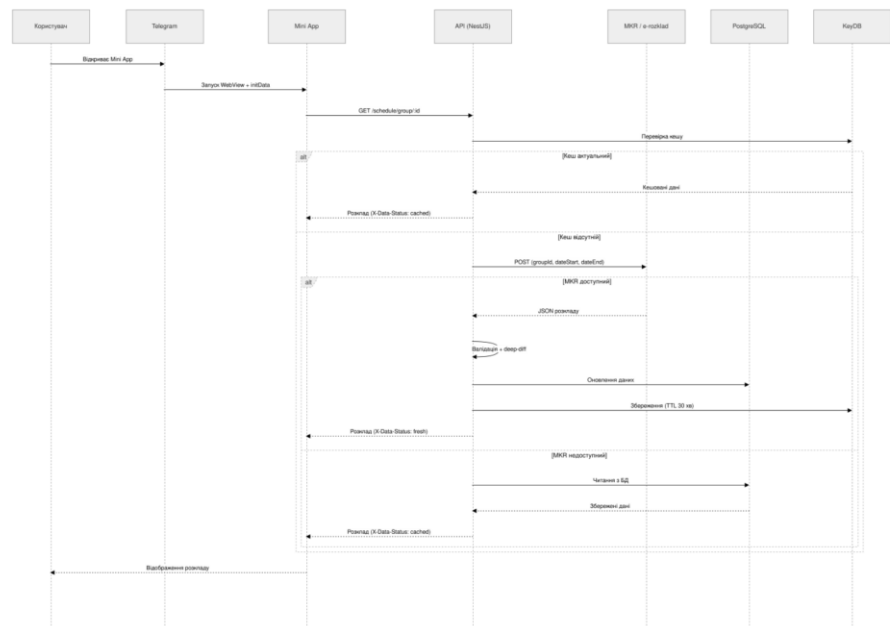
11

ДІАГРАМА ПОТОКУ ДАНИХ



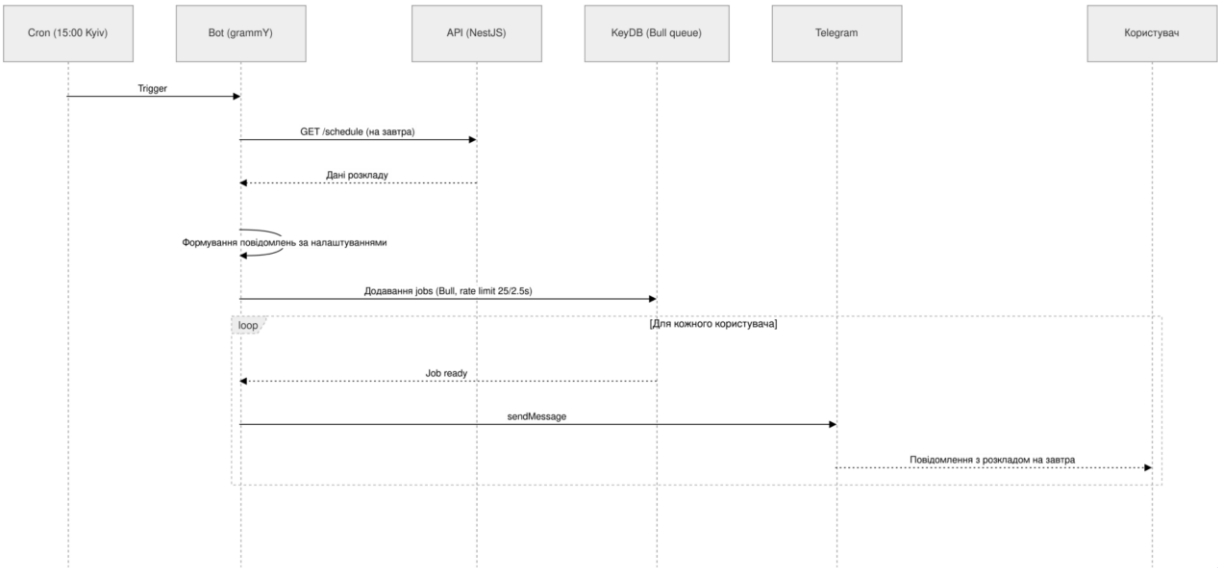
12

ДІАГРАМА ПОСЛІДОВНОСТІ



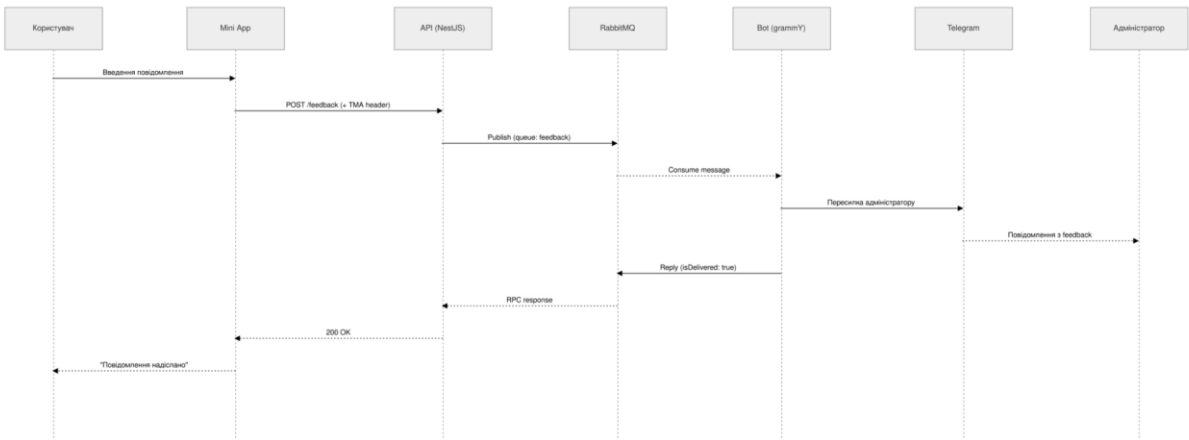
13

ДІАГРАМА ПОСЛІДОВНОСТІ

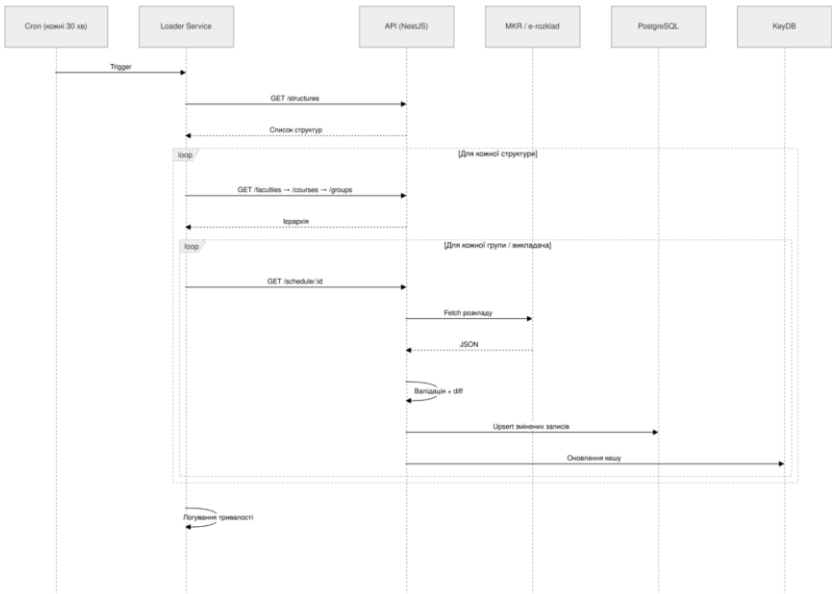


14

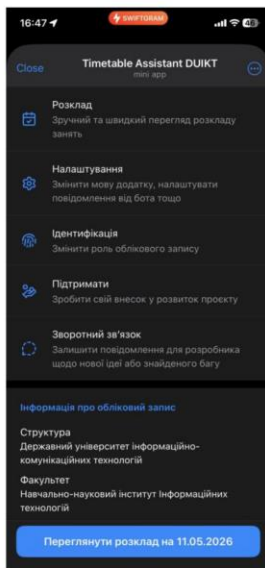
ДІАГРАМА ПОСЛІДОВНОСТІ



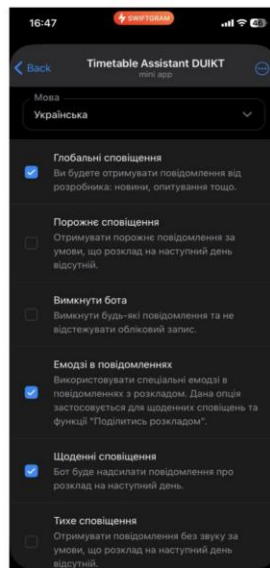
ДІАГРАМА ПОСЛІДОВНОСТІ



ЕКРАННІ ФОРМИ



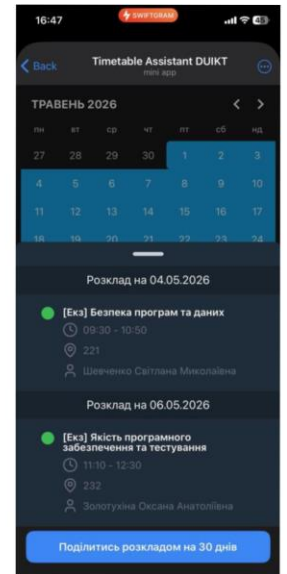
Головна



Налаштування

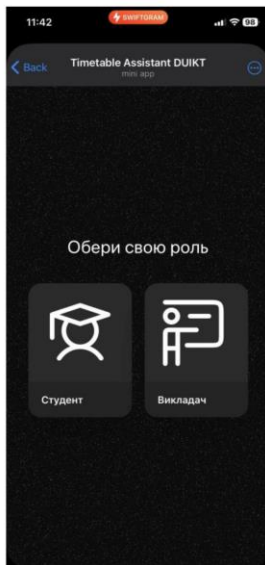


Вибір розкладу

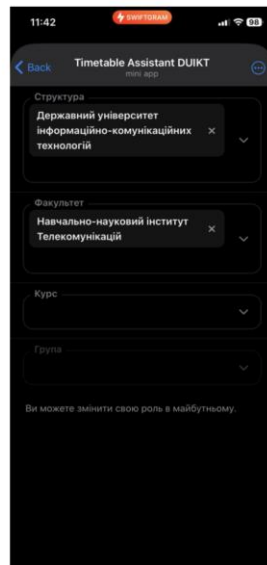


Розклад 17

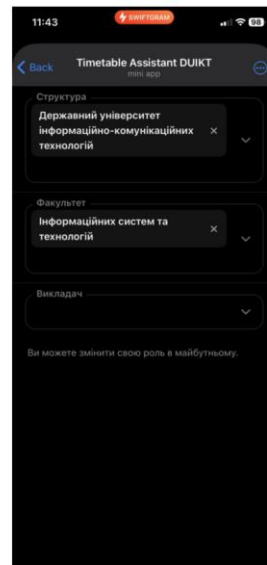
ЕКРАННІ ФОРМИ



Вибір ролі



Ідент. студента



Ідент. викладача



Зв. зв'язок 18

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

1. Кугаївський А.О., Залива В. В. Визначення вимог до телеграм-боту для забезпечення надійного доступу до розкладу навчальних занять. Матеріали II Всеукраїнської науково-технічної конференції «Виклики та рішення в програмній інженерії», 26 листопада 2025 року, м. Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 2025. С. 160-163.
2. Кугаївський А.О., Залива В. В. Визначення вимог до системи агрегації розкладу з підтримкою автономної роботи. Матеріали VII Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 23 квітня 2026 року, м. Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ 2026. С. 463-466.

19

ВИСНОВКИ

1. Проведено аналіз предметної галузі організації доступу до розкладу навчальних занять, виявлено ключові проблеми - критичну залежність від доступності єдиного джерела даних, відсутність проактивного інформування про зміни та обмежену персоналізацію в існуючих рішеннях.
2. Проаналізовано існуючі програмні засоби та на основі їхніх недоліків сформовано функціональні і нефункціональні вимоги.
3. На основі визначених вимог обрано технологічний стек: Node.js як платформу виконання, NestJS із Fastify для побудови REST API, grammY для інтеграції з Telegram Bot API, React із Vite для Mini Apps, PostgreSQL і Prisma для персистентного зберігання, KeyDB для кешування, RabbitMQ і Bull для асинхронної комунікації та керованої доставки сповіщень, Docker для контейнеризації.
4. Спроектовано мікросервісну архітектуру системи з чотирьох незалежних сервісів (API, бот, Mini Apps, loader), що забезпечує розділення відповідальностей та можливість горизонтального масштабування кожного компонента окремо.
5. Реалізовано ключовий функціонал: автентифікацію через Mini Apps, багаторівневе кешування з індикацією актуальності даних, щоденні сповіщення та сповіщення про зміни розкладу, багатомовний інтерфейс, адміністративні інструменти розсилки, зворотний зв'язок та моніторинг через Sentry.
6. Проведено ручне функціональне тестування всіх компонентів системи: перевірено сценарії перегляду розкладу, доставки щоденних сповіщень, коректної роботи кешу при симуляції недоступності МКР, синхронізації налаштувань між Mini Apps і ботом та обробки зворотного зв'язку. Результати тестування підтвердили відповідність реалізації функціональним і нефункціональним вимогам. Система успішно запущена у промислову експлуатацію та обслуговує близько 1000 активних користувачів щоденно.

20

ДОДАТОК Б. ЛІСТИНГИ ПРОГРАМНИХ МОДУЛІВ

```
// apps/miniapp/src/index.tsx
sentryInit({
  dsn: import.meta.env.VITE_SENTRY_DSN,
  environment: import.meta.env.DEV ? "development" :
"production",
  release: import.meta.env.VITE_APP_VERSION,
  sendDefaultPii: true,
  integrations: [
    captureConsoleIntegration({ levels: ["warn", "error"]
}),
    browserTracingIntegration(),
    browserSessionIntegration(),
    replayCanvasIntegration(),
    replayIntegration({
      maskAllInputs: false,
      maskAllText: false,
      networkDetailAllowUrls: [window.location.origin,
/* */],
      networkCaptureBodies: true,
    }),
  ],
  tracesSampleRate: 1.0,
  replaysSessionSampleRate: 0.5,
  replaysOnErrorSampleRate: 1.0,
});
```

```
ReactDOM.createRoot(root).render(<Root />);
// apps/miniapp/src/Root.tsx
export const Root = (): ReactNode => (
  <ErrorBoundary fallback={ErrorBoundaryError}>
    <TmaInit>
      <I18nextProvider i18n={i18n}>
        <QueryClientProvider client={queryClient}>
          <MantineProvider>
            <AnalyticsLoader />
            <App />
            <Toaster />
          </MantineProvider>
        </QueryClientProvider>
      </I18nextProvider>
    </TmaInit>
  </ErrorBoundary>
);
```

```
// apps/miniapp/src/App.tsx
export const App = (): ReactNode => {
  const initDataRaw = useRawInitData();
  const isDark = useSignal(miniApp.isDark);
  const { t } = useTranslation();
```

```
  const isUserIdentified = useStore((state) =>
state.isUserIdentified);
  const setIsUserIdentified = useStore((state) =>
state.setIsUserIdentified);
  const setUser = useStore((state) => state.setUser);
```

```
  const setSettings = useStore((state) =>
state.setSettings);
```

```
  const [loading, setLoading] = useState(true);
  const [mounted, setMounted] = useState(false);
```

```
  useEffect(() => {
    themeParams.mount();
    miniApp.mount();
    viewport.mount().then(() => {
      viewport.expand();
      viewport.bindCssVars();
    });
    mainButton.mount();
    swipeBehavior.mount.ifAvailable();
    themeParams.bindCssVars();
    miniApp.bindCssVars();
    setMounted(true);
```

```
  return () => {
    swipeBehavior.unmount();
    mainButton.unmount();
    miniApp.unmount();
    themeParams.unmount();
  };
}, []);
```

```
  useEffect(() => {
    if (!swipeBehavior.isMounted()) return;
    swipeBehavior.disableVertical.ifAvailable();
  }, [mounted]);
```

```
  useEffect(() => {
    if (!initDataRaw) return;
```

```
  async function getUser() {
    try {
      const { error, data } = await client.GET("/user/me",
{
      params: { header: { Authorization: `tma
${initDataRaw}` } },
    });
```

```
    if (error && error.message ===
NOT_FOUND_REQUEST_CODES.USER_NOT_FOU
ND) {
      setIsUserIdentified(false);
    }
```

```
    if (data) {
      const { settings, ...user } = data;
      setUser(user);
      setSettings(settings);
      await ensureI18nResources(settings.language);
      await i18next.changeLanguage(settings.language);
```

```

        setIsUserIdentified(true);
    }

    setLoading(false);
  } catch (err) {
    console.error(err);
  }
}

getUser();
}, [initDataRaw, setIsUserIdentified, setSettings,
setUser]);

return (
  <AppRoot appearance={isDark ? "dark" : "light"}
platform={"base"}>
    {loading && (
      <div className="min-h-dvh flex">
        <Spinner size="l" className="m-auto" />
      </div>
    )}
    {!loading && (
      <HashRouter>
        <AppRoutes isUserIdentified={isUserIdentified}
routes={routes} />
      </HashRouter>
    )}
  </AppRoot>
);
};
// apps/miniapp/src/navigation/routes.tsx
export enum NavigationRoutes {
  MAIN = "/",
  IDENTITY = "/identity",
  IDENTITY_STUDENT = "/identity/student",
  IDENTITY_TEACHER = "/identity/teacher",
  SCHEDULE = "/schedule",
  SETTINGS = "/settings",
  FEEDBACK = "/feedback",
}
// apps/miniapp/src/store/store.ts
const useStore = create<StoreState>()((set) => ({
  isUserIdentified: false,
  setIsUserIdentified: (isUserIdentified) => set({
isUserIdentified }),
  user: null,
  setUser: (user: UserData) => set({ user }),
  settings: null,
  setSettings: (settings: SettingsData) => set({ settings }),
}));
// apps/api/src/modules/app/app.module.ts
@Module({
  imports: [
    EnvalidModule.forRoot({ validators, isGlobal: true }),
    SentryModule.forRoot(),
    CacheModule.registerAsync<RedisClientOptions>({
  imports: [EnvalidModule.forRoot({ validators })],
  inject: [ENVALID],
  isGlobal: true,
  useFactory: (env: Config) => ({
    isGlobal: true,
    store: redisStore,

```

```

    password: env.KEYDB_PASSWORD,
    socket: { host: env.KEYDB_HOST, port:
env.KEYDB_PORT },
  })),
  }),
  ClientsModule.registerAsync({
    isGlobal: true,
    clients: [
      {
        name:
RABBITMQ_QUEUE_NAMES.FEEDBACK,
        imports: [EnvalidModule.forRoot({ validators })],
        inject: [ENVALID],
        useFactory: (env: Config) => ({
          transport: Transport.RMQ,
          options: {
            urls: [
              {
                hostname: env.RABBITMQ_HOST,
                username: env.RABBITMQ_USER,
                password: env.RABBITMQ_PASSWORD,
                port: env.RABBITMQ_PORT,
              },
            ],
            queue:
RABBITMQ_QUEUE_NAMES.FEEDBACK,
            queueOptions: { durable: true },
          },
        })),
      ],
      {
        name:
RABBITMQ_QUEUE_NAMES.LANGUAGE,
        // ... аналогічна конфігурація для черги мови
      },
    ],
  }),
  PrismaModule,
  UserModule,
  FeedbackModule,
  StructureModule,
  FacultyModule,
  CourseModule,
  GroupModule,
  ChairModule,
  TeacherModule,
  ScheduleModule,
  HealthModule,
],
  providers: [{ provide: APP_FILTER, useClass:
SentryGlobalFilter }],
})
export class AppModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {

    consumer.apply(TMAMiddleware).forRoutes(UserContr
oller, FeedbackController);
  }
}
// apps/api/src/main.ts
const fastifyAdapter = new FastifyAdapter({ https:
setCerts() });

```

```

fastifyAdapter.enableCors({
  credentials: true,
  exposedHeaders: [CACHE_HEADER],
  methods: ["GET", "POST", "PATCH"],
  allowedHeaders: ["Authorization", "Content-Type",
"Cache-Control", "Pragma"],
});

const app = await
NestFactory.create<NestFastifyApplication>(AppModule, fastifyAdapter);

app.useGlobalPipes(new ValidationPipe({ whitelist: true,
transform: true }));

const document = SwaggerModule.createDocument(app,
buildSwaggerDocument(env));
app.use("/docs", apiReferenceOptions(document));

await app.listen(env.PORT, "0.0.0.0");
// apps/bot/src/app.ts
export const bot = new
Bot<MyContext>(env.BOT_TOKEN);

await commands.setCommands(bot);

bot.api.config.use(
  indexOutgoingUpdate(),
  autoRetry({ rethrowInternalServerError: true,
maxRetryAttempts: 5, maxDelaySeconds: 2500 }),
);

bot.use(indexIncomingUpdate, commands,
myChatMemberEvent);

const privateChatMiddleware = bot.chatType("private");

privateChatMiddleware.use(
  sequentialize((ctx) => String(ctx.chat.id)),
  session({
    initial: () => {
      return {};
    },
    getSessionKey: (ctx) => String(ctx.chat.id),
    storage: redisAdapterClient,
  }),
  i18n,
  handleUserLanguage,
  emptyWebAppCallback,
  startCommand,
  helpCommand,
  privacyCommand,
  developerInfoCommand,
);

privateChatMiddleware
  .filter((ctx) => ctx.chat.id === env.ADMIN_ID)
  .use(
    conversations(),
    cancelCommand,

```

```

    createConversation(announcementConversation,
ANNOUNCEMENT_CONVERSATION),
    announcementCommand,
    dailyCommand,
    disableFeedbackCallback,
  );

bot.catch(handleError);

const runner = run(bot, {
  runner: { fetch: { allowed_updates: ["callback_query",
"message", "my_chat_member"] } },
});
// apps/bot/src/clients/redisAdapterClient.ts
export const redisAdapterClient = new RedisAdapter({
  instance: keyDbClient,
  ttl: env.isProd ? 60 * 60 : 30,
});
// apps/bot/src/utils/schedule.ts
schedule("0 15 * * 0-4", () => scheduleAlert({
scheduleDailyAlert: true }), { timezone: "Europe/Kyiv"
});

schedule("0 15 * * 5", () => scheduleAlert({
receiveWeekendsAlert: true }), { timezone:
"Europe/Kyiv" });
// apps/loader/src/app.ts
async function mainCron() {
  console.log(`START LOADER: ${new
Date().toISOString()}`);

  const startLoaderTime = performance.now();
  const monthDates = calculateMonthDates(new Date());

  try {
    const structures = await loadStructures();

    const loadTeachersData = async () => {
      for (const structure of structures) {
        const chairs = await loadChairs(structure.id);
        for (const chair of chairs) {
          const teachers = await
loadTeachersByChair(structure.id, chair.id);
          for (const teacher of teachers) {
            await loadTeacherSchedule(teacher.id,
monthDates.startAt, monthDates.endAt);
          }
        }
      }
    };

    const loadGroupsData = async () => {
      for (const structure of structures) {
        const faculties = await loadFaculties(structure.id);
        for (const faculty of faculties) {
          const courses = await loadCourses(structure.id,
faculty.id);
          for (const course of courses) {
            const groups = await loadGroups(structure.id,
faculty.id, course.course);
            for (const group of groups) {

```

```

        await loadGroupSchedule(group.id,
monthDates.startAt, monthDates.endAt);
    }
}
};

await Promise.all([loadTeachersData(),
loadGroupsData()]);
} catch (err) {
    console.error(err);
}

const endLoaderTime = performance.now();
console.log(`LOADER DONE: ${((endLoaderTime -
startLoaderTime) / 1000 / 60) MIN.}`);
}

schedule("*/30 * * * *", () => mainCron(), { timezone:
"Europe/Kyiv" });
// apps/loader/src/utlis/calculateMonthDates.ts
export function calculateMonthDates(startDate: Date,
offsetDays: number = 0) {
    const start = addDays(startDate, offsetDays);
    const end = addDays(start, 30);

    return {
        startAt: format(start, DATE_FORMAT),
        endAt: format(end, DATE_FORMAT),
    };
}
// apps/api/src/middlewares/tma.middleware.ts
import { ForbiddenException, Inject, Injectable,
NestMiddleware } from "@nestjs/common";
import { ExpiredError, InitData, parse, validate } from
"@tma.js/init-data-node";
import { FORBIDDEN_REQUEST_CODES } from
"@tta-duikt/types";
import { FastifyReply, FastifyRequest } from "fastify";
import { ENVALID } from "nestjs-envalid";
import { Config } from "../environment/validators.js";

export type ModifiedFastifyRequest = FastifyRequest &
{ raw: { tma: InitData } };

@Injectable()
export class TMAMiddleware implements
NestMiddleware {
    constructor(@Inject(ENVALID) private readonly env:
Config) {}

    use(req: ModifiedFastifyRequest["raw"], _res:
FastifyReply["raw"], next: () => void) {
        const [strategy, data = ""] =
req.headers.authorization?.split(" ") ?? [];

        if (strategy !== "tma" || !data) throw new
ForbiddenException(FORBIDDEN_REQUEST_CODES
.INVALID_STRATEGY);

        try {

```

```

        validate(data, this.env.BOT_TOKEN);
        req.tma = parse(data);
    } catch (err) {
        console.error(err);

        if (ExpiredError.is(err)) throw new
ForbiddenException(FORBIDDEN_REQUEST_CODES
.STRATEGY_DATA_EXPIRED);

        throw new
ForbiddenException(FORBIDDEN_REQUEST_CODES
.INVALID_STRATEGY);
    }

    next();
}
// apps/api/src/interceptors/cache.interceptor.ts
import { CACHE_MANAGER } from "@nestjs/cache-
manager";
import { CallHandler, ExecutionContext, Inject,
InternalServerErrorException, NestInterceptor } from
"@nestjs/common";
import { CACHE_HEADER,
CACHE_HEADER_STATUS,
INTERNAL_REQUEST_CODES } from "@tta-
duikt/types";
import { Cache } from "cache-manager";
import { ENVALID } from "nestjs-envalid";
import { map, Observable, of } from "rxjs";
import { Config } from "../environment/validators.js";

const FIVE_MINUTES_TTL = 1000 * 60 * 5;
const THIRTY_MINUTES_TTL =
FIVE_MINUTES_TTL * 6;

export class CacheInterceptor implements
NestInterceptor {
    constructor(
        @Inject(ENVALID) private readonly env: Config,
        @Inject(CACHE_MANAGER) private readonly
cacheManager: Cache,
    ) {}

    private getHeaderValue(isRequestSuccess: boolean) {
        return isRequestSuccess ?
CACHE_HEADER_STATUS.FRESH :
CACHE_HEADER_STATUS.CACHED;
    }

    async intercept(context: ExecutionContext, next:
CallHandler): Promise<Observable<unknown>> {
        const httpContext = context.switchToHttp();
        const httpResponse =
httpContext.getResponse<FastifyReply>();

        const cacheKey = httpResponse.request.originalUrl;

        try {
            const retrieveCacheData: { isRequestSuccess:
boolean; data: Array<unknown> } =
await this.cacheManager.get(cacheKey);

```

```

    httpResponse.header(CACHE_HEADER,
this.getHeaderValue(retrieveCacheData?.isRequestSuccess));

    if (retrieveCacheData) return
of(retrieveCacheData.data);

    return next.handle().pipe(
    map(async (output) => {
        if (output?.data && Array.isArray(output.data)) {
            const ttl = !output.data ? FIVE_MINUTES_TTL :
THIRTY_MINUTES_TTL;

            await this.cacheManager.set(cacheKey, output,
this.env.isProd ? ttl : 1000 * 5);

            httpResponse.header(CACHE_HEADER,
this.getHeaderValue(output.isRequestSuccess));
        }

        return output.data;
    }),
    );
} catch (err) {
    console.error(err);
    throw new
InternalServerErrorException(INTERNAL_REQUEST_
CODES.UNKNOWN_ERROR);
}
}
// apps/api/src/modules/schedule/schedule.service.ts
@Injectable()
export class ScheduleService {
    constructor(
        @Inject(ENVALID) private readonly env: Config,
        private readonly prismaService: PrismaService,
    ) {}

    private responseSchema = t.array(
    t.type({
        date: t.string,
        lessons: t.array(
            t.type({
                number: t.number,
                periods: t.array(
                    t.type({
                        disciplineFullName: t.string,
                        disciplineShortName: t.string,
                        classroom: t.string,
                        timeStart: t.string,
                        timeEnd: t.string,
                        teachersName: t.string,
                        teachersNameFull: t.string,
                        typeStr: t.string,
                        dateUpdated: t.string,
                        groups: t.string,
                        // ... інші поля
                    }),
                ),
            ),
        ),
    })),
    );

```

```

    ),
    })),
    );

    async getSchedule(
        type: "GROUP" | "TEACHER",
        id: number,
        query: { startAt: string; endAt: string },
    ): Promise<{ isRequestSuccess: boolean; data:
Array<GetScheduleResponseDto> }> {
        const isEntityExists = await this.isEntityExists(id, type
=== "GROUP");
        if (!isEntityExists) return { isRequestSuccess: false,
data: [] };

        let isRequestSuccess = true;
        let response: t.TypeOf<typeof this.responseSchema>;

        try {
            const endpoint = type === "GROUP" ?
API_ENDPOINTS.GROUP_SCHEDULE :
API_ENDPOINTS.TEACHER_SCHEDULE;

            const request = await
fetchRaw([this.env.UNIVERSITY_BASE_API,
endpoint], {
                method: "POST",
                headers: { "Content-Type": "application/json",
"Accept-Language": "uk" },
                body: JSON.stringify({
                    groupId: type === "GROUP" ? id : undefined,
                    teacherId: type === "TEACHER" ? id : undefined,
                    dateStart: query.startAt,
                    dateEnd: query.endAt,
                }),
            });

            if (request.status !== 200) throw new
Error(API_NON_OK_RESPONSE);
            response = await request.json();
        } catch (err) {
            console.error(err);
            isRequestSuccess = false;
        }

        const schedule = await
this.retrieveFromDatabase(type, id, query);

        if (!response || (response &&
!this.isResponseValid(response))) {
            return { isRequestSuccess: false, data:
plainToInstance(GetScheduleResponseDto, schedule) };
        }

        const transformedResponseData =
this.transformData(response);

        const objectDiff = diff(schedule,
transformedResponseData);

        if (!Object.entries(objectDiff).length) {

```

```

    return { isRequestSuccess, data:
plainToInstance(GetScheduleResponseDto, schedule) };
}

try {
    const key = type === "GROUP" ? "groupId" :
"teacherId";
    const date = {
        lte: new Date(query.endAt).toISOString(),
        gte: new Date(query.startAt).toISOString(),
    };

    let data: GetScheduleResponseDto[];

    const result = await
this.prismaService.$transaction(async (prisma) => {
        if (type === "GROUP") {
            const group = await
prisma.group.findFirstOrThrow({ where: { groupId: id }
});
            await prisma.schedule.deleteMany({ where: {
groupId: group.id, date } });
            data = transformedResponseData.map((item) => ({
...item, [key]: group.id }));
        } else {
            const teacher = await
prisma.chairTeacher.findFirstOrThrow({ where: {
teacherId: id } });
            await prisma.schedule.deleteMany({ where: {
teacherId: teacher.id, date } });
            data = transformedResponseData.map((item) => ({
...item, [key]: teacher.id }));
        }

        const created = await
prisma.schedule.createManyAndReturn({
            data,
            select: this.selectFields,
        });

        return created;
    });

    return { isRequestSuccess, data:
plainToInstance(GetScheduleResponseDto, result) };
} catch (err) {
    console.error(err);
    throw new
InternalServerErrorException(INTERNAL_REQUEST_
CODES.DATABASE_UNAVAILABLE);
}
}
}

```