МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ
КРИВОРІЗЬКИЙ ФАХОВИЙ КОЛЕДЖ
ДЕРЖАВНОГО НЕКОМЕРЦІЙНОГО ПІДПРИЄМСТВА
«ДЕРЖАВНИЙ УНІВЕРСИТЕТ «КИЇВСЬКИЙ АВІАЦІЙНИЙ ІНСТИТУТ»
Циклова комісія *комп'ютерних систем та мереж*
(повна назва циклової комісії)

Допустити до захисту
Голова випускової циклової комісії
комп'ютерних систем та мереж
(повна назва циклової комісії)
(підпис) *Ірина КРАВЧУК*
(ім'я, ПРІЗВИЩЕ)
« 10 » 06 2025 р.

КВАЛІФІКАЦІЙНА РОБОТА
(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИКА ОСВІТНЬО-ПРОФЕСІЙНОГО СТУПЕНЯ
ФАХОВИЙ МОЛОДШИЙ БАКАЛАВР

Тема: *Чат-бот розкладу занять Криворізького фахового коледжу Державного некомерційного підприємства «Державний університет «Київський авіаційний інститут»*

Група: *3-013* Спеціальність: *123 «Комп'ютерна інженерія»*

Здобувач освіти (підпис) *Олександр МУСІЄНКО* (ім'я, ПРІЗВИЩЕ)

Керівник роботи (підпис) *Світлана ТЕРЬОШИНА* (ім'я, ПРІЗВИЩЕ)

Консультант з оформлення пояснювальної записки (підпис) *Оксана ОСАДЧА* (ім'я, ПРІЗВИЩЕ)

Кривий Ріг 2025 р.

КРИВОРІЗЬКИЙ ФАХОВИЙ КОЛЕДЖ
ДЕРЖАВНОГО НЕКОМЕРЦІЙНОГО ПІДПРИЄМСТВА
«ДЕРЖАВНИЙ УНІВЕРСИТЕТ «КИЇВСЬКИЙ АВІАЦІЙНИЙ ІНСТИТУТ»

Відділення *комп'ютерної та програмної інженерії*
Циклова комісія *комп'ютерних систем та мереж*
Освітньо-професійний ступінь *фаховий молодший бакалавр*
Спеціальність *123 «Комп'ютерна інженерія»*

ЗАТВЕРДЖУЮ
Голова випускової циклової комісії
комп'ютерних систем та мереж
(повна назва циклової комісії)
_____ *Ірина КРАВЧУК*
(підпис) (ім'я, ПРІЗВИЩЕ)
« 10 » 03 2025 р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧУ ОСВІТИ

Мусієнко Олександр Сергійович
(прізвище, ім'я, по батькові)

1.Тема роботи *Чат-бот розкладу занять Криворізького фахового коледжу Державного некомерційного підприємства «Державний університет «Київський авіаційний інститут»*

Керівник роботи *викладач вищої категорії, Терьошина Світлана Сергіївна*
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по коледжу від « *04* » *04* 2025 року № *50-ст*

2. Строк подання здобувачем освіти роботи з 19.05.2025 по *13.06.2025*

3. Вихідні дані до роботи *Розробка чат-бота розкладу занять з використанням мікросервісної архітектури.*

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
Аналіз предметної області управління навчальним розкладом, визначення недоліків існуючих підходів. Обґрунтування архітектурного підходу та технологічного стеку. Проєктування бази даних та структури системи. Впровадження контейнеризації та системи CI/CD. Розгортання системи.

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

Презентація Microsoft PowerPoint

6. Консультанти розділів роботи (проекту)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання _____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	*Узгодження технічного завдання з керівником кваліфікаційної роботи*	*17.03.2025-21.03.2025*	*виконано*
2	*Підбір та вивчення науково-технічної літератури за темою кваліфікаційної роботи*	*24.03.2025-28.03.2025*	*виконано*
3	*Обґрунтування вибору програмних засобів*	*31.03.2025-04.04.2025*	*виконано*
4	*Опис компонентів. Обґрунтування їх вибору.*	*07.04.2025-09.04.2025*	*виконано*
5	*Розробка програмного забезпечення*	*10.04.2025-28.04.2025*	*виконано*
6	*Дослідження ефективності реалізованих методів.*	*29.04.2025-02.05.2025*	*виконано*
7	*Написання пояснювальної записки*	*12.05.2025-23.05.2025*	*виконано*
8	*Перевірка на плагіат пояснювальної записки*	*26.05.2025-30.05.2025*	*виконано*
9	*Попередній захист кваліфікаційної роботи*	*02.06.2025-06.06.2025*	*виконано*
10	*Захист кваліфікаційної роботи*	*16.06.2025*	

Здобувач освіти _____ *Олександр МУСІЄНКО*
(підпис) (ім'я, ПРІЗВИЩЕ)

Керівник роботи _____ *Світлана ТЕРЬОШИНА*
(підпис) (ім'я, ПРІЗВИЩЕ)

StrikePlagiarism.com

Дата звіту 6/4/2025
Дата редагування ---

Документ прийнятий

Звіт подібності

метадані

Назва організації
Ukrainian national aviation university
Заголовок
Кваліфікаційна_робота_Мусієнко_3013
Автор Науковий керівник / Експерт
МусієнкоТерьошина С.С
підрозділ
Криворізький Фаховий коледж

Обсяг знайдених подібностей

Коефіцієнт подібності визначає, який відсоток тексту по відношенню до загального обсягу тексту було знайдено в різних джерелах. Зверніть увагу, що високі значення коефіцієнта не автоматично означають плагіат. Звіт має аналізувати компетентна / уповноважена особа.

2.60%
2.60%
КП 1

1.29%
1.29%
КЦ

25
Довжина фрази для коефіцієнта подібності 2

12300
Кількість слів

95996
Кількість символів

Тривога

У цьому розділі ви знайдете інформацію щодо текстових спотворень. Ці спотворення в тексті можуть говорити про МОЖЛИВІ маніпуляції в тексті. Спотворення в тексті можуть мати навмисний характер, але частіше характер технічних помилок при конвертації документа та його збереженні, тому ми рекомендуємо вам підходити до аналізу цього модуля відповідально. У разі виникнення запитань, просимо звертатися до нашої служби підтримки.

Заміна букв		20
Інтервали	A→	0
Мікропробіли	·	90
Білі знаки		0
Парафрази (SmartMarks)	a	22

Подібності за списком джерел

Нижче наведений список джерел. В цьому списку є джерела із різних баз даних. Колір тексту означає в якому джерелі він був знайдений. Ці джерела і значення Коефіцієнту Подібності не відображають прямого плагіату. Необхідно відкрити кожне джерело і проаналізувати зміст і правильність оформлення джерела.

10 найдовших фраз — Колір тексту

ПОРЯДКОВИЙ НОМЕР	НАЗВА ТА АДРЕСА ДЖЕРЕЛА URL (НАЗВА БАЗИ)	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)
1	https://maxkochenkov.github.io/database_group1/requirements/stakeholders-needs.html	18 0.15 %
2	https://maxkochenkov.github.io/database_group1/requirements/stakeholders-needs.html	18 0.15 %
3	https://openarchive.nure.ua/bitstream/document/15599/1/2020_M_ST_Lobintsev_AA.pdf	16 0.13 %
4	https://endway.org/threads/aiogram-blokirovka-polzovatelej-cherez-middleware.1233/	15 0.12 %

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи «Чат-бот розкладу занять Криворізького фахового коледжу Державного некомерційного підприємства «Державний університет «Київський авіаційний інститут» містить: 62 сторінки основного тексту, 23 рисунки, 1 додаток, 19 використаних джерел.

TELEGRAM-БОТ, *PYTHON*, *FASTAPI*, МІКРОСЕРВІСНА АРХІТЕКТУРА, *POSTGRESQL*, *REDIS*, *NATS*, *DOCKER*, *CI*/*CD*.

У даній кваліфікаційній роботі розробляється програмне забезпечення для автоматизації управління навчальним розкладом у закладах освіти з метою використання в *Telegram*-боті.

Метою роботи є розробка системи, яка полегшує процес формування та оновлення розкладу, забезпечуючи зручний інтерфейс та доступ до інформації для користувачів. Актуальність роботи полягає в усуненні недоліків традиційних методів управління розкладом, таких як ручні процеси, схильність до помилок та затримки в інформуванні, шляхом автоматизації та інтеграції з популярним месенджером *Telegram*.

Об’єктом дослідження є розробка автоматизованої системи для управління розкладом, яка включає мікросервіси для обробки даних, інтеграцію з *Telegram*-ботом та автоматизацію рутинних задач. Для реалізації проекту використано сучасні технології: мова програмування *Python*, фреймворк *FastAPI* для створення *API*, бібліотека *Aiogram* для розробки *Telegram*-бота, система управління базами даних *PostgreSQL*, система кешування *Redis*, брокер повідомлень *NATS*, а також контейнеризація через *Docker* та *CI*/*CD* для розгортання.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

Aiogram – асинхронний фреймворк для *Telegram Bot API*, написаний на *Python* з використанням *asyncio* [2].

API (англ. *Application Programming Interface*) – програмний інтерфейс застосунку, набір визначень підпрограм, протоколів взаємодії та засобів для створення програмного забезпечення.

CI/CD (англ. *Continuous Integration*/*Continuous Deployment*) – практики безперервної інтеграції та безперервного розгортання, спрямовані на автоматизацію процесів збірки, тестування та доставки програмного забезпечення.

CRUD (англ. *Create*, *Read*, *Update*, *Delete*) – акронім, що позначає чотири базові функції, які використовуються при роботі з сховищами даних.

DB (англ. *Database*) – база даних, організована сукупність даних, призначена для тривалого зберігання, накопичення та активного використання в інформаційних системах.

Docker – програмне забезпечення для автоматизації розгортання та управління застосунками в середовищах з підтримкою контейнеризації [3].

FastAPI – сучасний, швидкий (високопродуктивний) веб-фреймворк для створення *API* на *Python* [1].

FSM (англ. *Finite State Machine*) – скінченний автомат, математична абстракція, що моделює поведінку системи з дискретними станами та переходами між ними.

JSON (англ. *JavaScript Object Notation*) – текстовий формат обміну даними, заснований на *JavaScript*, що легко читається людьми та обробляється комп’ютерами.

LMS (англ. *Learning Management System*) – система управління навчанням, програмне забезпечення для адміністрування, документування, відстеження, звітності та надання навчальних програм.

ORM (англ. *Object-Relational Mapping*) – технологія програмування, яка пов'язує бази даних з концепціями об'єктно-орієнтованих мов програмування, створюючи абстрактну базу даних.

Pydantic – бібліотека *Python* для валідації даних та управління налаштуваннями за допомогою анотацій типів *Python* [8].

Python – високорівнева мова програмування загального призначення з динамічною типізацією та автоматичним керуванням пам'яттю.

Redis (англ. *Remote Dictionary Server*) – система керування базами даних класу *NoSQL*, що працює зі структурами даних типу «ключ — значення».

REST (англ. *Representational State Transfer*) – архітектурний стиль взаємодії компонентів розподіленого застосунку в мережі.

SQL (англ. *Structured Query Language*) – декларативна мова програмування, що застосовується для створення, модифікації й керування даними в реляційних базах даних.

SQLAlchemy – бібліотека *Python*, що надає інструментарій *SQL* та об'єктно-реляційний перетворювач (*ORM*).

TOTP (англ. *Time-based One-Time Password*) – алгоритм створення одноразових паролів, що залежать від часу, для захищеної автентифікації.

єРозклад – комплексна автоматизована мікросервісна система управління навчальним розкладом та оперативного інформування здобувачів освіти й викладачів через *Telegram*-бот.

ВСТУП

Ефективна організація навчального процесу є ключовим фактором успішності сучасних освітніх закладів. Інформаційні технології відіграють дедалі важливішу роль у модернізації та оптимізації адміністративних та освітніх функцій. Одним з найскладніших та водночас критично важливих аспектів є управління навчальним розкладом та оперативне інформування всіх учасників освітнього процесу про його зміни.

Традиційні підходи до формування та розповсюдження розкладу, що часто базуються на ручній праці, використанні табличних процесорів та паперових носіїв, характеризуються низкою суттєвих недоліків. Серед них висока трудомісткість, схильність до помилок, затримки в оновленні інформації, фрагментованість даних та недостатня оперативність інформування здобувачів освіти і викладачів, особливо у випадках непередбачених змін.

У відповідь на ці виклики, зростає потреба у створенні комплексних систем, здатних забезпечити управління розкладом, миттєве сповіщення про зміни та зручний доступ до актуальної інформації для всіх користувачів.

Метою даної роботи є розробка та впровадження ефективної автоматизованої системи для управління навчальним розкладом, оперативним інформуванням учасників освітнього процесу про зміни та автоматизацією пов’язаних процесів, включаючи обробку замін з електронних листів.

З мети роботи виникають такі завдання:

- Провести аналіз предметної області, виявити недоліки існуючих підходів до управління розкладом та інформування.
- Спроєктувати архітектуру системи “єРозклад” на основі мікросервісного підходу та розробити модель бази даних.
- Розробити *API* сервіс для централізованого управління даними розкладу, замін, практик та іспитів.

- Створити *Telegram*-бота як основний інтерфейс взаємодії для здобувачів освіти та викладачів, забезпечивши функції перегляду розкладу, отримання сповіщень та налаштування персональних уподобань.
- Впровадити механізми контейнеризації та оркестрації сервісів для спрощення розгортання та управління системою.
- Реалізувати систему безперервної інтеграції та доставки (*CI*/*CD*) для автоматизації процесів оновлення програмного забезпечення.

Об’єктом дослідження є процеси управління навчальним розкладом та інформування учасників освітнього процесу в навчальних закладах.

Предметом дослідження є архітектурні підходи, технологічний стек (*Python*, *FastAPI*, *Aiogram*, *PostgreSQL*, *Redis*, *NATS*, *Docker*, *Google Gmail API*), алгоритми та програмні засоби для реалізації мікросервісної системи “єРозклад”, призначеної для автоматизації управління навчальним розкладом, обробки змін та інформування користувачів.

РОЗДІЛ 1
ПРЕДМЕТНА ОБЛАСТЬ ТА ПРОЄКТУВАННЯ СИСТЕМИ

1.1 Огляд процесів управління навчальним розкладом

Нині управління навчальним розкладом у більшості навчальних закладів здійснюється з використанням ручної роботи в кабінетах кафедр. Спочатку кафедри збирають від викладачів інформацію про їхні часові вікна та аудиторні потреби — це робиться в табличних процесорах таких як *Excel* або *Google Sheets*. Деканат контролює, щоб кожна дисципліна мала призначеного викладача і належну аудиторію а також вносить корективи з урахуванням обмежень аудиторій і погоджує фінальний варіант з керівництвом закладу. Після затвердження розклад публікується на офіційному порталі навчального закладу, інколи надсилається електронною поштою. Оновлення розкладу під час семестру зазвичай відбувається вручну, відповідні файли редагуються працівниками деканату чи секретаріату кафедри, а потім повторно публікуються.

Типові проблеми такого підходу пов’язані насамперед із паперовою та фрагментованою інформацією. Друковані розклади часто втрачаються, стають неактуальними вже через перші коригування. Затримки з оновленнями нерідко виникають через необхідність проходження багатьох узгоджень. Відсутність єдиного централізованого інформування призводить до того, що частина здобувачів освіти залишається без актуальної інформації бо вони не отримали важливе повідомлення про зміни вчасно. Окрім того, через ручне введення даних можливі технічні помилки такі як подвійні бронювання аудиторій, невірні часові проміжки, конфлікти між групами.

1.2 Недоліки існуючих підходів до управління розкладом

Традиційна система управління розкладом базується на застарілих методах ручного складання та обробки інформації. Диспетчерські служби освітніх закладів досі використовують паперові носії як основний інструмент для створення

розкладу занять та змін до нього. Такий підхід характеризується високою трудомісткістю та схильністю до помилок.

Навчальні заклади змушені покладатися на загальнодоступні офісні програми, які не пристосовані для специфіки управління навчальним розкладом. Це призводить до неефективного використання людських ресурсів, оскільки співробітники диспетчерських служб витрачають значну кількість часу на рутинні операції, які могли б бути автоматизовані.

Проблема інформування учасників освітнього процесу стоїть особливо гостро. Здобувачі освіти та викладачі не мають доступу до оперативної інформації про зміни в розкладі, що створює хаос в організації навчального процесу. Єдиним способом отримання актуальної інформації залишається особисте відвідування інформаційних стендів, розташованих у навчальних корпусах. Така система інформування є вкрай неефективною, особливо для здобувачів освіти, які навчаються в різних корпусах або мають змішану форму навчання. Особливо проблематичними є ситуації, коли зміни вносяться в день проведення занять, після початку навчального дня. У таких випадках здобувачі освіти можуть просто не дізнатися про скасування або перенесення занять, що призводить до марної витрати часу та порушення навчального процесу.

1.3 Визначення основних функціональних можливостей системи

Центральним компонентом взаємодії із системою для кінцевих користувачів є *Telegram*-бот, який служить інтерфейсом доступу до всіх необхідних функцій. Вибір *Telegram* як платформи обумовлений його поширенням серед аудиторії здобувачів освіти, простотою використання та можливістю забезпечення миттєвих сповіщень.

Процес персоналізації досвіду користувача починається з реєстрації, під час якої користувач визначає свою роль в освітньому процесі та прив’язує свій акаунт до конкретної академічної групи або викладача. Ця інформація стає основою для надання персоніфікованого контенту. Система дозволяє користувачам самостійно

керувати налаштуваннями сповіщень, обираючи типи повідомлень, які вони бажають отримувати, та формат відображення інформації про викладачів.

Система забезпечує можливість перегляду розкладу на поточний день, що дозволяє швидко орієнтуватися в денному плані занять. Додатково передбачена можливість перегляду розкладу на наступний день. Для більш детального планування користувачі можуть отримати доступ до тижневого розкладу з інтерактивною навігацією між днями та можливістю перемикання між типами тижнів.

Критично важливою функцією є оперативне інформування про зміни в розкладі. Користувачі можуть отримати персоналізований список актуальних замін, що стосуються їхньої групи або викладача. Система також надає можливість перегляду всіх замін у навчальному закладі, що може бути корисним для координації між різними учасниками освітнього процесу. Інформація про заміни подається у структурованому вигляді з чітким зазначенням дати, часу та характеру змін. Користувачі, які активували відповідні налаштування, автоматично отримують персоналізовані повідомлення про зміни у їхньому розкладі.

Управління замінами в розкладі реалізовано через спеціалізовані кінцеві точки *API*, які дозволяють адміністраторам оперативно додавати нові заміни. Процес автоматичного сповіщення адміністраторів про надходження нових замін реалізований через внутрішню систему обміну повідомленнями.

1.4 Обґрунтування архітектурного підходу та технологічного стеку

1.4.1 Застосування мікросервісної архітектури

Мікросервісна архітектура представляє собою підхід до розробки програмного забезпечення, при якому велика система розбивається на набір невеликих, незалежних сервісів, кожен з яких виконує конкретну бізнес-функцію та взаємодіє з іншими через добре прописані інтерфейси.

Основні принципи мікросервісної архітектури включають децентралізоване управління даними, незалежність розгортання, відмовостійкість та

масштабованість окремих компонентів. Кожен мікросервіс має власну базу даних або область даних, що забезпечує повну автономність та дозволяє командам розробників працювати незалежно один від одного.

В системі “єРозклад” мікросервісна архітектура має чітко виділені функції такі як надання *API* для доступу до даних розкладу, взаємодія з користувачами через *Telegram*-бот, інтеграція з поштовими сервісами та автоматизована обробка файлів замін. Кожна з цих функцій має власну логіку а також вимоги до продуктивності та життєвий цикл розробки.

У монолітній архітектурі збій в одному компоненті може призвести до відмови всієї системи. Мікросервісна архітектура дозволяє уникати подібних проблем. Перевагою є масштабування різних частин системи.

Реалізація мікросервісної архітектури в “єРозклад” базується на чотирьох основних компонентах. Перший компонент – це *API* сервіс (*e_schedule_api*), який виступає центральним хабом для доступу до всіх даних системи. Він надає *REST API* для отримання даних про розклад, заміни, групи, викладачів та іспити.

Другий компонент – *Telegram*-бот (*e_schedule_bot*) – являє собою окремий сервіс, який створений для взаємодії з користувачами через платформу *Telegram*. Він має власну базу даних для збереження інформації про користувачів, їх налаштування та історію взаємодій. Важливою особливістю цього сервіса є його здатність працювати як споживач повідомлень від інших сервісів через систему *NATS*, що дозволяє йому миттєво реагувати на зміни в розкладі або появу нових замін.

Третій компонент – сервіс інтеграції з *Gmail* (*e_schedule_gmail_connector*) – виконує специфічну задачу моніторингу поштової скриньки та виявлення листів з інформацією про заміни. Цей сервіс працює автономно, періодично перевіряючи нові листи та публікуючи відповідні повідомлення для інших компонентів системи.

Четвертий компонент – інструмент автоматизації обробки замін (*e_schedule_autocomplete*) – представляє собою *CLI*-додаток, який може запускатися як окремий процес або контейнер для виконання специфічних задач

обробки файлів *Excel*. Цей підхід дозволяє використовувати потужні можливості штучного інтелекту для структурування даних.

Синхронна взаємодія відбувається через *HTTP REST API*, коли *Telegram*-бот звертається до *API* сервісу за даними про розклад або заміни. Цей підхід забезпечує простоту реалізації та налагодження, а також дозволяє легко тестувати окремі компоненти.

Асинхронна взаємодія реалізована через систему обміну повідомленнями *NATS*, яка забезпечує *pub*/*sub* модель комунікації. Коли сервіс *Gmail* виявляє новий лист з замінами, він публікує відповідне повідомлення в *NATS*, а *Telegram*-бот підписується на такі повідомлення та негайно сповіщає користувачів про зміни. Такий підхід забезпечує слабке зв’язування між сервісами та високу швидкість реакції на події.

Кожен сервіс має власну область відповідальності за дані. *API* сервіс управляє основними даними про розклад, групи, викладачів та заміни. *Telegram*-бот має власну базу даних користувачів з їх налаштуваннями та підписками. Сервіс *Gmail* не зберігає дані постійно, а лише обробляє їх та передає далі.

Кожен сервіс має власний репозиторій коду та може розгортатися незалежно від інших. Це дозволяє швидше впроваджувати нові функції та виправлення помилок, не чекаючи готовності всієї системи.

1.4.2 Мова програмування *Python* та її екосистема

Python є однією з найпопулярніших мов програмування, яка поєднує в собі простоту синтаксису, потужні можливості та величезну екосистему бібліотек і фреймворків. Перевагою *Python* є його виразний та читабельний синтаксис, який значно прискорює процес розробки та полегшує підтримку коду. *Python* також підтримує різні парадигми програмування – об’єктно-орієнтовану, функціональну та процедурну, що дає розробникам гнучкість у виборі найкращого підходу для конкретних задач.

Важливим фактором при виборі *Python* стала його потужна підтримка асинхронного програмування через модуль *asyncio*. Сучасні веб-додатки, особливо

ті, що працюють з великою кількістю одночасних запитів, потребують ефективної обробки *I/O* операцій. Асинхронне програмування дозволяє обробляти тисячі одночасних з'єднань без блокування потоків виконання, що критично важливо для *API*.

Екосистема *Python* включає величезну кількість високоякісних бібліотек та фреймворків, які значно спрощують розробку складних систем. Для створення *API* сервісу була обрана бібліотека *FastAPI*, яка представляє собою сучасний, швидкий веб-фреймворк для створення *API* з автоматичною генерацією документації. *FastAPI* базується на стандартах *OpenAPI* та *JSON Schema*, що забезпечує автоматичну валідацію даних, серіалізацію та генерацію інтерактивної документації. Фреймворк також має вбудовану підтримку асинхронного програмування, що дозволяє ефективно обробляти великі навантаження.

Для роботи з базами даних в системі використовується *SQLAlchemy* – один з найпотужніших *ORM* фреймворків для *Python*. *SQLAlchemy* надає високорівневий інтерфейс для роботи з реляційними базами даних, дозволяючи розробникам працювати з даними як з *Python* об'єктами, абстрагуючись від специфіки конкретної СУБД. Важливою особливістю *SQLAlchemy* є підтримка асинхронних операцій через *asyncpg* драйвер, що забезпечує неблокуючу роботу з *PostgreSQL* базою даних.

Розробка *Telegram*-бота базується на фреймворку *Aiogram*, який є одним з найсучасніших та найпотужніших інструментів для створення ботів у *Python* екосистемі. *Aiogram* надає повну підтримку *Telegram Bot API*, включаючи всі новітні функції платформи. Фреймворк побудований на принципах асинхронного програмування, що дозволяє боту ефективно обробляти велику кількість одночасних користувачів.

Для валідації даних та управління конфігурацією в усіх сервісах системи використовується *Pydantic* – бібліотека, яка забезпечує валідацію даних з використанням *Python type hints*. *Pydantic* автоматично перевіряє типи даних під час виконання, конвертує дані до потрібних типів та генерує зрозумілі

повідомлення про помилки. Це значно підвищує надійність системи та спрощує процес налагодження.

Інтеграція з зовнішніми сервісами реалізована за допомогою спеціалізованих клієнтських бібліотек. Для роботи з *Gmail API* використовується офіційна *google-api-python-client* бібліотека, яка надає повний доступ до всіх можливостей *Google API*. Ця бібліотека включає автоматичне управління автентифікацією, обробку помилок та підтримку всіх методів *Gmail API*. Додатково використовуються *google-auth* та *google-auth-oauthlib* для безпечної автентифікації через *OAuth* 2.0 протокол.

Для *HTTP* клієнтських запитів в асинхронному коді використовується *httpx* – сучасна альтернатива популярній бібліотеці *requests*, яка підтримує як синхронні, так і асинхронні операції. *Httpx* надає тісну інтеграцію з *asyncio* та забезпечує високу продуктивність при роботі з *HTTP API*. Бібліотека підтримує *HTTP*/2, *connection pooling* та автоматичну обробку *cookies*, що робить її ідеальною для взаємодії між мікросервісами.

Для роботи з часовими зонами та планування задач використовуються відповідно бібліотеки *pytz* та *APScheduler*. *Pytz* забезпечує точну роботу з часовими зонами, а *APScheduler* дозволяє планувати виконання задач за різними розкладами – від простих інтервалів до складних *cron*-подібних правил.

1.4.3 База даних, кешування та обмін повідомленнями

Системи управління базами даних (СУБД) становлять основу будь-якої інформаційної системи, забезпечуючи надійне зберігання, організацію та доступ до даних. Існують різні типи СУБД, кожен з яких має свої переваги та недоліки. Реляційні бази даних залишаються найпоширенішим вибором для більшості бізнес-додатків завдяки своїй надійності та широкій підтримці стандартів *SQL*. Реляційні СУБД забезпечують строгу консистентність даних через механізми *ACID*-транзакцій, підтримують складні запити з об’єднаннями та агрегацією, а також мають розвинені системи індексування для оптимізації продуктивності.

NoSQL бази даних розроблені для роботи з великими обсягами неструктурованих або частково структурованих даних. *NoSQL* системи зазвичай жертвують консистентністю заради масштабованості та продуктивності, що робить їх підходящими для високонавантажених розподілених систем.

Для системи “єРозклад” обрано *PostgreSQL* як основну СУБД. *PostgreSQL* є повнофункціональною об’єктно-реляційною базою даних з відкритим вихідним кодом. Система підтримує повний спектр *ACID*-властивостей, що критично важливо для забезпечення цілісності даних про розклад занять та заміни.

PostgreSQL демонструє відмінну продуктивність при роботі з складними запитами, що особливо важливо для системи управління розкладом, де часто потрібно виконувати запити з множинними об’єднаннями таблиць груп, викладачів, дисциплін та розкладів. Розвинена система індексування, включаючи *B-tree* та *Hash* індекси, дозволяє оптимізувати продуктивність для різних типів запитів.

Системи кешування зберігають часто використовувані дані в оперативній пам’яті, що дозволяє значно прискорити доступ до них порівняно з зверненнями до диска.

Існують різні підходи до кешування на різних рівнях архітектури додатка. Кешування на рівні додатка передбачає зберігання даних у пам’яті самого додатка, що забезпечує найшвидший доступ, але обмежується ресурсами одного сервера. Розподілене кешування використовує окремі сервери або кластери для зберігання кешованих даних, що дозволяє масштабувати систему горизонтально. Кешування на рівні бази даних включає внутрішні механізми СУБД для кешування запитів та результатів.

Redis обрано як основну систему кешування. Основною перевагою *Redis* є його надзвичайно висока швидкість доступу до даних завдяки зберіганню всіх даних в оперативній пам’яті. Система підтримує різні типи даних, включаючи рядки, хеші, списки, множини та сортовані множини, що дозволяє ефективно реалізовувати різні сценарії кешування.

У контексті системи управління розкладом *Redis* використовується для кешування результатів часто виконуваних запитів до бази даних. Розклад занять для конкретної групи на поточний день, список всіх груп, інформація про викладачів та інші статичні або малозмінні дані зберігаються в *Redis* для швидкого доступу. Це особливо важливо для *Telegram*-бота, який повинен швидко відповідати на запити користувачів.

Redis підтримує механізми автоматичного видалення застарілих даних через налаштування *TTL* (*Time To Live*) для кешованих записів. Це дозволяє автоматично оновлювати кеш без необхідності ручного втручання.

Брокери повідомлень є ключовим компонентом мікросервісної архітектури, забезпечуючи асинхронну комунікацію між різними сервісами системи. Вони дозволяють розв’язати сервіси один від одного, підвищити відмовостійкість системи та забезпечити горизонтальне масштабування.

Існують різні типи систем обміну повідомленнями з різними гарантіями доставки та моделями комунікації. *At-most-once* доставка гарантує, що повідомлення буде доставлено не більше одного разу, але може бути втрачено при відмовах. *At-least-once* доставка забезпечує, що повідомлення буде доставлено принаймні один раз, але може дублюватися. *Exactly-once* доставка гарантує доставку повідомлення рівно один раз, що є найсуворішою, але й найскладнішою для реалізації гарантією.

Для системи “єРозклад” обрано *NATS* як брокер повідомлень. *NATS* є легким та високопродуктивним брокером повідомлень. Основною перевагою *NATS* є його простота та мінімальні вимоги до ресурсів, що робить його ідеальним вибором для систем з помірним навантаженням.

NATS використовує модель публікації-підписки (*pub*/*sub*) з підтримкою тематичної маршрутизації повідомлень. Система забезпечує *at-most-once* доставку повідомлень, що означає відсутність дублювання, але можливість втрати повідомлень при відмовах. Для системи управління розкладом така модель є прийнятною, оскільки більшість повідомлень мають інформаційний характер. *e_schedule_gmail_connector* публікує повідомлення про нові листи з замінами, які

обробляються *Telegram*-ботом для сповіщення адміністраторів. *API* сервіс може публікувати події про зміни в розкладі, які підхоплюються ботом для розсилки сповіщень користувачам. Така архітектура забезпечує слабке зв’язування сервісів та дозволяє легко додавати нові компоненти без зміни існуючих.

1.4.4 Технології контейнеризації

Контейнеризація є однією з найважливіших технологій сучасної розробки програмного забезпечення, яка революціонізувала підходи до розгортання, масштабування та управління додатками. Ця технологія дозволяє упаковувати додатки разом з усіма їхніми залежностями в легкі, портативні контейнери, які можуть виконуватися на будь-якій системі, що підтримує контейнеризацію.

Основною перевагою контейнеризації є забезпечення консистентності середовища виконання на всіх етапах життєвого циклу додатка.

Docker став стандартом контейнеризації завдяки своїй простоті використання, потужній екосистемі та широкій підтримці спільноти. *Docker Engine* є основним компонентом, який забезпечує створення, запуск та управління контейнерами.

Docker Images представляють собою незмінні шаблони, які використовуються для створення контейнерів. Образи будуються на основі *Dockerfile* - текстового файла, який містить інструкції для створення образу. Кожна інструкція в *Dockerfile* створює новий шар в образі, що дозволяє ефективно кешувати та перевикористовувати спільні компоненти між різними образами.

1.5 Проєктування бази даних

Концептуальна модель даних являє собою високорівневе представлення структури інформації, яка буде зберігатися та оброблятися системою управління навчальним розкладом.

Викладачі представляють собою центральну сутність освітнього процесу, яка містить персональну інформацію про педагогічний склад навчального закладу. Ця сутність включає прізвище, ім’я, по батькові викладача, а також унікальні ініціали,

що використовуються для швидкої ідентифікації у розкладі. Викладачі можуть бути задіяні у різних типах навчальної діяльності, включаючи основні заняття, заміни, практики, консультації та іспити.

Групи є основною організаційною одиницею здобувачів освіти і характеризуються унікальною назвою та приналежністю до конкретного відділення. Групи служать базовим елементом для планування розкладу, оскільки саме для них призначаються заняття з різних дисциплін.

Предмети відображають навчальні дисципліни, які викладаються у навчальному закладі. Кожен предмет має унікальну назву та може бути частиною різних типів занять - основних, замін, практик, консультацій або іспитів.

Заняття представлені через різні сутності залежно від їх типу. Основні заняття формують регулярний розклад на тиждень, заміни відображають тимчасові зміни в основному розкладі, практики організують підготовку здобувачів освіти, а консультації та іспити представляють контрольні заходи. Кожне заняття характеризується датою, часом, номером уроку та іншими специфічними параметрами.

Аудиторії є місцями проведення навчальних занять і характеризуються унікальними номерами або назвами.

Для забезпечення повноцінного функціонування системи введено ряд допоміжних сутностей, які підтримують основні бізнес-процеси та забезпечують гнучкість управління розкладом.

Відділення організують групи за структурними підрозділами навчального закладу, що дозволяє здійснювати адміністративне управління та аналіз навчального процесу.

Дні тижня стандартизують представлення календарних днів у системі, забезпечуючи однозначність при формуванні розкладу. Ця сутність дозволяє організувати циклічний розклад на тижневій основі.

Типи тижнів реалізують концепцію чергування тижнів, що є поширеною практикою в організації навчального процесу. Це дозволяє створювати різні варіанти розкладу для різних тижнів семестру.

Інтервали тижнів конкретизують типи тижнів у часі, визначаючи точні календарні періоди для кожного типу тижня протягом навчального року.

Причини замін класифікують різні обставини, що призводять до необхідності внесення змін у розклад, такі як хвороба викладача, службове відрядження, святкові дні тощо.

Практики характеризуються типом практики, зміною, місцем проведення, датами початку та закінчення, а також можливими примітками. Ця інформація необхідна для організації виробничого навчання здобувачів освіти поза межами навчального закладу.

Консультації та іспити об’єднані в одну групу сутностей, оскільки мають схожі характеристики - конкретний час проведення, предмет, викладача та аудиторію. Відмінність полягає лише у типі заходу, що визначається відповідною класифікаційною сутністю.

Розклад дзвінків регламентує часові інтервали навчального дня, визначаючи час початку та закінчення уроків, а також перерв між ними.

Розклад їдальні забезпечує інформацією про час роботи харчового блоку навчального закладу, включаючи різні типи прийомів їжі та їх часові інтервали для різних днів тижня.

Telegram-бот як інтерфейс взаємодії з користувачами має власну структуру даних, яка забезпечує персоналізацію та адміністрування системи.

Користувачі бота характеризуються унікальним ідентифікатором *Telegram*, іменем користувача, налаштуваннями інтерфейсу та ролями доступу. Кожен користувач має персональні налаштування, які визначають спосіб взаємодії з системою.

Ролі користувачів визначають рівень доступу до функціональності системи, розділяючи звичайних користувачів та адміністраторів з різними правами.

Налаштування користувачів зберігають персональні переваги щодо отримання повідомлень та способу відображення інформації.

Події користувачів фіксують історію взаємодії з ботом, що дозволяє аналізувати активність та виявляти проблеми у функціонуванні системи.

Адміністративні облікові дані забезпечують додаткові механізми безпеки для користувачів з підвищеними привілеями, включаючи двофакторну автентифікацію.

Центральною сутністю основного розкладу є *MainSchedule*, яка пов’язана з днем тижня, типом тижня та предметом через прямі зовнішні ключі. Водночас ця сутність має багато-до-багатьох зв’язки з групами, викладачами та аудиторіями через проміжні таблиці, що дозволяє одному заняттю мати декілька груп, викладачів або аудиторій одночасно.

Заміни мають більш складну структуру зв’язків, оскільки повинні відображати як предмет та викладача, що замінюється, так і предмет та викладача, що заміняє. Це реалізовано через два окремі зовнішні ключі на сутність предметів та два набори проміжних таблиць для викладачів.

Групи пов’язані з відділеннями через простий зв’язок один-до-багатьох, що відображає організаційну структуру навчального закладу. Кожна група належить рівно одному відділенню, але відділення може мати багато груп.

Практики зроблені через опціональні зв’язки з місцями проведення та примітками, що дозволяє адаптувати модель до різних форм організації виробничого навчання.

Часові інтервали для дзвінків та роботи їдальні пов’язані з днями тижня, що дозволяє мати різні розклади для різних днів, адаптуючись до особливостей навчального процесу.

Логічна модель даних представляє детальну структуру бази даних, яка безпосередньо відображає концептуальну модель у реляційної бази даних.

Таблиця *teachers* реалізує сутність викладачів з полями *teacher_id* типу *Integer* як первинний ключ, *last_name*, *first_name* та *middle_name* типу *String* з максимальною довжиною 30 символів для зберігання повного імені, та *initials* типу *String* довжиною 30 символів з унікальним обмеженням для швидкої ідентифікації викладача у розкладі. Поле *middle_name* має можливість містити *NULL* значення, оскільки не всі викладачі мають по батькові.

Таблиця *groups* містить *group_id* як первинний ключ типу *Integer*, *group_name* типу *String* довжиною 5 символів з унікальним обмеженням для назви групи, та *department_id* як зовнішній ключ типу *Integer*, що посилається на таблицю *departments*.

Таблиця *subjects* реалізована з *subject_id* як первинний ключ типу *Integer* та *subject_name* типу *String* максимальною довжиною 120 символів з унікальним обмеженням. Значна довжина поля назви предмету дозволяє зберігати повні найменування навчальних дисциплін.

Таблиця *classrooms* має просту структуру з *classroom_id* як первинним ключем типу *Integer* та *classroom_name* типу *String* довжиною 5 символів з унікальним обмеженням.

Таблиця *departments* включає *department_id* як первинний ключ типу *Integer* та *department_name* типу *String* довжиною 40 символів з унікальним обмеженням для найменування відділень.

Система часової організації навчального процесу реалізована через набір взаємопов’язаних таблиць, що забезпечують гнучкість планування та циклічність розкладу.

Таблиця *days_of_week* містить *day_of_week_id* як первинний ключ типу *Integer* та *day_of_week_name* типу *String* довжиною 9 символів з унікальним обмеженням. Ця довжина достатня для зберігання найдовших назв днів тижня українською мовою.

Таблиця *week_types* реалізована з *week_type_id* як первинним ключем типу *Integer* та *week_type_name* типу *String* довжиною 9 символів з унікальним обмеженням. Це дозволяє зберігати типи тижнів як “чисельник” та “знаменник”.

Таблиця *week_intervals* з’єднує типи тижнів з конкретними календарними періодами через поля *interval_id* як первинний ключ типу *Integer*, *start_date* та *end_date* типу *Date* з унікальними обмеженнями, та *week_type_id* як зовнішній ключ типу *Integer*.

Центральна таблиця *main_schedules* містить *schedule_id* як первинний ключ типу *Integer*, *day_of_week_id* та *week_type_id* як зовнішні ключі типу *Integer*,

lesson_number типу *SmallInteger* для номера уроку, та *subject_id* як зовнішній ключ типу *Integer*. Використання *SmallInteger* для номера уроку оптимізує використання пам'яті, оскільки кількість уроків на день обмежена.

Зв'язкові таблиці *main_schedule_groups*, *main_schedule_teachers* та *main_schedule_classrooms* реалізують відношення багато-до-багатьох між розкладом та відповідними сутностями. Кожна з цих таблиць має композитний первинний ключ, що складається з *schedule_id* та відповідного ідентифікатора сутності, обидва типу *Integer*. Використання *CASCADE* при видаленні забезпечує цілісність даних при видаленні записів розкладу. Схему організації основного розкладу занять показано на рисунку 1.1.

Рисунок 1.1 – Схема організації основного розкладу занять

Таблиця *substitutions* має складну структуру для відображення логіки заміщення занять. Вона включає *substitution_id* як первинний ключ типу *Integer*, *substitution_date* типу *Date*, *lesson_number* типу *SmallInteger*, *replaced_subject_id* та *substituting_subject_id* як зовнішні ключі типу *Integer*, де перший може бути *NULL* для випадків додавання нових занять. Поле *substitution_reason_id* типу *Integer* посилається на таблицю причин замін.

Таблиця *substitution_reasons* містить *substitution_reason_id* як первинний ключ типу *Integer* та *substitution_reason_name* типу *String* довжиною 30 символів з унікальним обмеженням.

Зв’язкові таблиці для замін включають *substitutions_groups*, *substitutions_classrooms*, *substitutions_replaced_subject_teachers* та *substitutions_substituting_subject_teachers*, кожна з композитним первинним ключем та *CASCADE* обмеженнями для забезпечення цілісності. Схему організації замін у розкладі показано на рисунку 1.2.

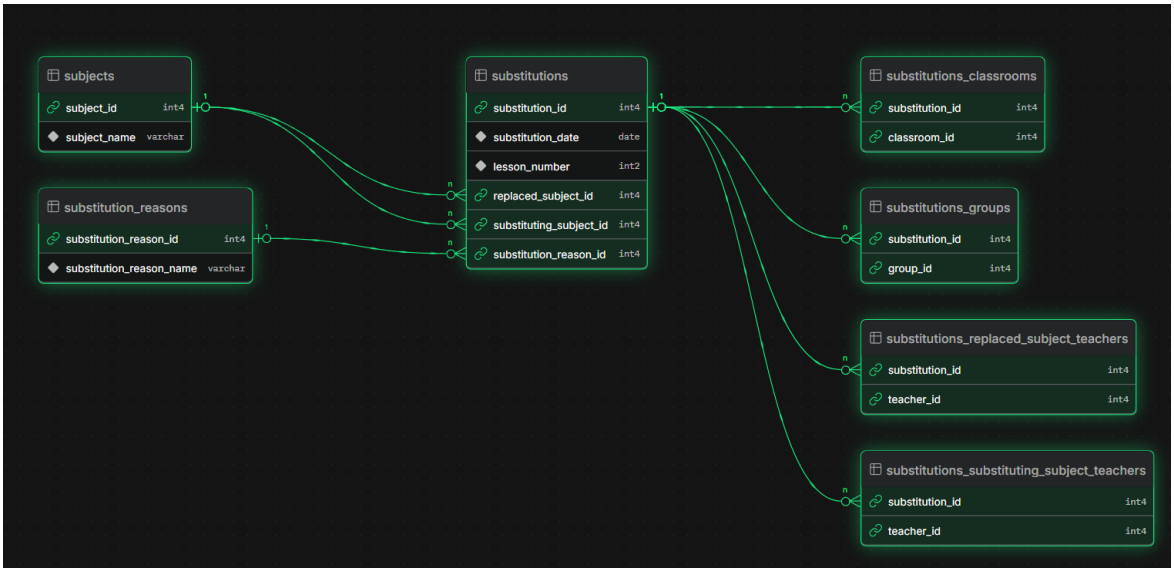

Рисунок 1.2 – Схема організації замін у розкладі

Таблиця *practice_schedules* реалізує складну структуру для організації виробничого навчання з полями *practice_id* як первинний ключ типу *Integer*, *practice_type_id*, *practice_date_id* як обов’язкові зовнішні ключі типу *Integer*, *practice_shift* типу *SmallInteger*, *practice_place_id* та *practice_note_id* як опціональні зовнішні ключі типу *Integer*.

Допоміжні таблиці включають *practice_types* з найменуванням типу практики довжиною 50 символів, *practice_places* з найменуванням місця довжиною 100 символів, *practice_dates* з парою дат початку та закінчення типу *Date*, та *practice_notes* з текстом примітки довжиною 100 символів.

Зв’язкові таблиці *practice_schedule_teachers* та *practice_schedule_groups* забезпечують відношення багато-до-багатьох з відповідними сутностями. Схему організації розкладу практик показано на рисунку 1.3.

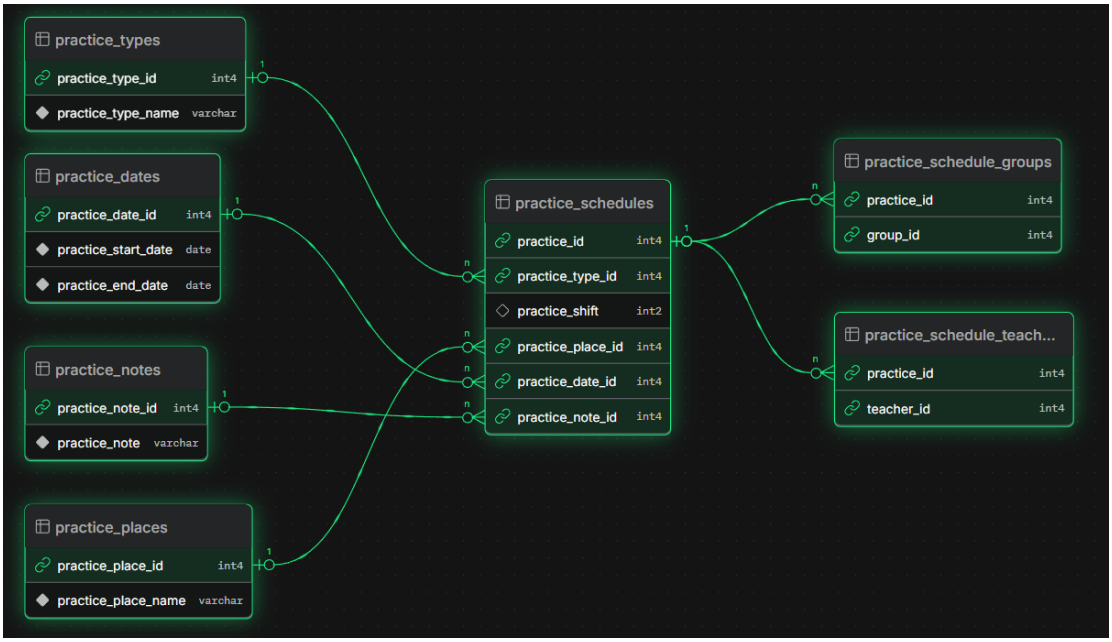

Рисунок 1.3 – Схема організації розкладу практик

Таблиця *exam_consultation_schedules* містить *exam_id* як первинний ключ типу *Integer*, *datetime* типу *DateTime* для точного часу проведення, *subject_id* та *exam_type_id* як зовнішні ключі типу *Integer*. Використання *DateTime* замість окремих полів дати та часу спрощує роботу з часовими мітками.

Таблиця *exam_consultation_types* визначає типи заходів з полем *exam_type_name* типу *String* довжиною 12 символів, достатньою для термінів “Консультація” та “Іспит”. Схему організації розкладу іспитів та консультацій показано на рисунку 1.4. Зв’язкові таблиці *exam_consultation_schedules_groups*, *exam_consultation_schedules_teachers* та *exam_consultation_schedules_classrooms* мають стандартну структуру з композитними первинними ключами.

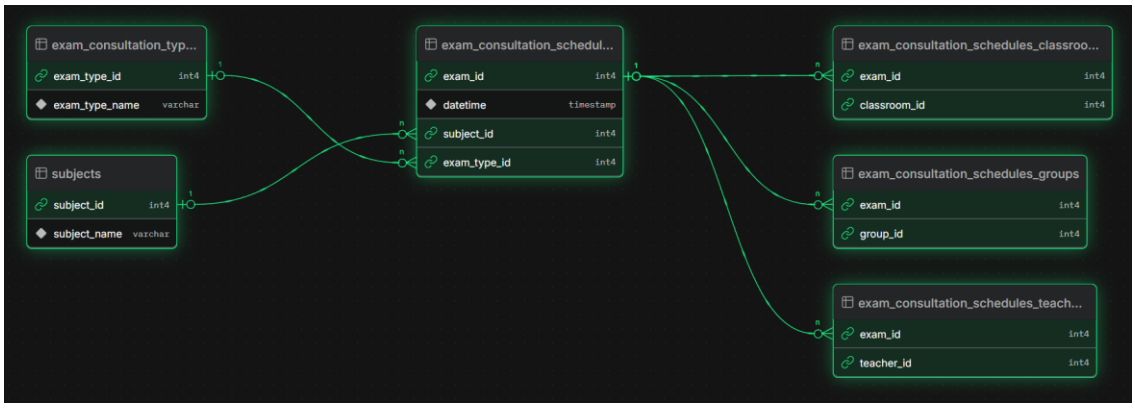

Рисунок 1.4 – Схема організації розкладу іспитів та консультацій

Таблиця *bell_schedules* включає *bell_event_id* як первинний ключ типу *Integer*, *day_of_week_id*, *bell_time_interval_id* та *bell_event_type_id* як зовнішні ключі типу *Integer*, та *lesson_number* типу *SmallInteger* як опціональне поле.

Таблиця *bell_time_intervals* містить часові інтервали з полями *bell_time_interval_start* та *bell_time_interval_end* типу *Time*, що дозволяє точно визначати тривалість уроків та перерв. Схему організації розкладу дзвінків показано на рисунку 1.5. Таблиця *bell_event_types* визначає типи подій з полем *bell_event_type_name* довжиною 20 символів для термінів як “Початок уроку” та “Кінець перерви”.

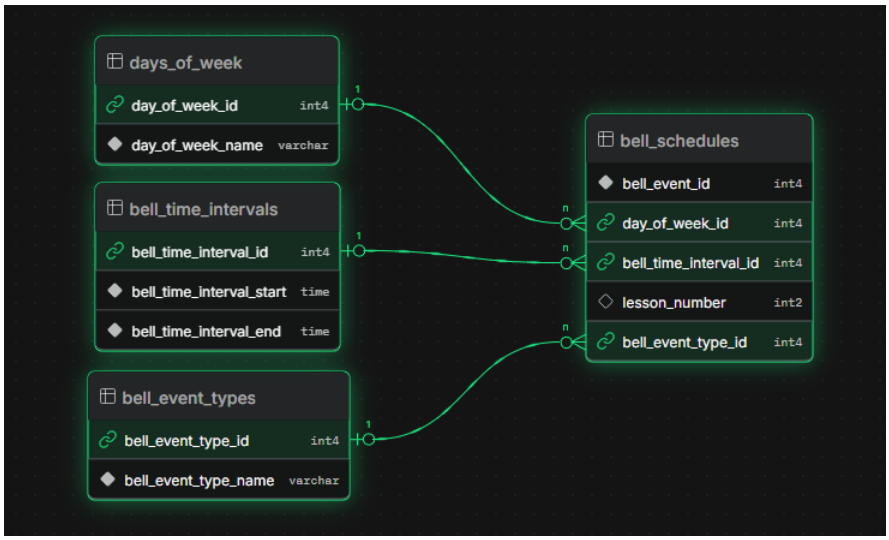

Рисунок 1.5 – Схема організації розкладу дзвінків

Таблиця *users* є центральною для системи автентифікації та авторизації з полями *user_id* як первинний ключ типу *Integer*, *telegram_user_id* типу *BigInteger* з унікальним обмеженням для ідентифікатора *Telegram*, *telegram_username* типу *String* довжиною 32 символи, *user_preference_id*, *role_id* та *interface_language_id* як зовнішні ключі типу *Integer*, та *registered_at* типу *DateTime* з автоматичним встановленням поточного часу.

Класифікаційні таблиці *user_roles*, *user_preferences*, *interface_languages* та *event_types* мають стандартну структуру з ідентифікатором та найменуванням відповідної довжини.

Таблиця *user_events* фіксує активність користувачів з полями *user_event_id* як первинний ключ типу *Integer*, *user_id* та *event_type_id* як зовнішні ключі типу *Integer*, та *event_timestamp* типу *DateTime* з автоматичним встановленням часу.

Таблиця *user_settings* зберігає персональні налаштування з полями *user_setting_id* як первинний ключ типу *Integer*, *user_id* як унікальний зовнішній ключ типу *Integer*, *mailing_subscription* та *display_full_name* типу *Boolean*.

Таблиця *admin_credentials* реалізує систему двофакторної автентифікації з полями *admin_credential_id* як первинний ключ типу *Integer*, *user_id* як унікальний зовнішній ключ типу *Integer*, *access* та *note* типу *String* довжиною 100 символів, *totp_secret* типу *String* довжиною 36 символів з унікальним обмеженням, та *create_at* типу *DateTime*. Схему таблиці користувачів показано на рисунку 1.6.

Рисунок 1.6 – Схема таблиці користувачів

РОЗДІЛ 2
РОЗРОБКА *API* СЕРВІСУ ТА *TELEGRAM*-БОТА

2.1 *API* сервіс

API сервіс системи “єРозклад” є центральним компонентом архітектури, який забезпечує уніфікований доступ до всіх даних навчального розкладу, замін, інформації про групи, викладачів та супутніх освітніх сутностей. Розробка сервісу базується на сучасному високопродуктивному веб-фреймворку *FastAPI*, що забезпечує надійну основу для створення *REST API* з автоматичною генерацією інтерактивної документації *OpenAPI* та ефективною валідацією даних завдяки глибокій інтеграції з бібліотекою *Pydantic*.

2.1.1 Архітектура та структура проєкту

Архітектурна концепція сервісу орієнтована на асинхронну обробку запитів, що досягається завдяки вбудованим можливостям *FastAPI* та використанню асинхронного драйвера *asyncpg* для взаємодії з базою даних *PostgreSQL*.

Модульна структура проєкту організована таким чином, що кожен логічний блок функціональності має власний простір імен та відповідальність. Роутери, розташовані в директорії *routers*, відповідають за обробку *HTTP*-запитів до конкретних кінцевих точок. Кожен роутер фокусується на певній предметній області: *schedule.py* обслуговує запити щодо основного розкладу занять, *substitutions.py* обробляє інформацію про заміни, *teachers.py* та *groups.py* надають доступ до довідкової інформації про викладачів та групи відповідно.

Моделі даних *Pydantic* виконують валідацію вхідних даних запитів та серіалізація даних для відповідей. Кожна модель відповіді містить всі необхідні поля з правильними типами даних, що гарантує консистентність інтерфейсу *API* та спрощує інтеграцію з клієнтськими додатками.

Шар роботи з базою даних інкапсулює всю логіку взаємодії з *PostgreSQL* через *SQLAlchemy ORM*. Менеджер бази даних містить статичні методи для

виконання *CRUD*-операцій та складних запитів з об'єднаннями таблиць, агрегацією даних та фільтрацією. Використання асинхронних операцій дозволяє ефективно обробляти велику кількість одночасних запитів без блокування потоків виконання.

Конфігураційна підсистема централізовано управляє всіма налаштуваннями сервісу через змінні середовища. Це включає параметри підключення до бази даних *PostgreSQL*, налаштування *Redis* для кешування, конфігурацію *NATS* для обміну повідомленнями та інші операційні параметри. Використання *pydantic-settings* забезпечує типобезпечність конфігурації та автоматичну валідацію налаштувань при запуску додатку.

Головний файл *main.py* виконує функцію точки входу додатку та координатора всіх компонентів. Він ініціалізує екземпляр *FastAPI*, підключає всі роутери з відповідними префіксами *URL*, налаштовує *middleware* для обробки *CORS*-запитів та управляє життєвим циклом додатку, включаючи ініціалізацію кешу та закриття з'єднань при завершенні роботи.

2.1.2 Розробка форматів запитів та відповідей

У системі "єРозклад" формати апитів та відповідей розроблені з урахуванням принципів *RESTful* архітектури, забезпечуючи інтуїтивність використання, консистентність структури даних та ефективність передачі інформації.

Центральним елементом системи є кінцева точка денного розкладу *GET /api/schedule/day*, який демонструє гнучкість системи параметризації. Параметр *type* визначає тип суб'єкта запиту - група чи викладач, що дозволяє використовувати одну кінцеву точку для різних сценаріїв використання. Такий підхід значно спрощує архітектуру додатків, оскільки уникає дублювання логіки для різних типів користувачів. Параметр *identifier* містить унікальний ідентифікатор суб'єкта - назву групи або ініціали викладача. Для забезпечення точності запитів система підтримує повні назви груп, наприклад "3-013", та повні ініціали викладачів у форматі "Призвіще І.Б.". Параметри *weekType* та *dayOfWeek* дозволяють точно специфікувати часовий контекст запиту, що критично важливо для навчальних закладів з чергуванням тижнів.

Практичний приклад запиту *GET* /*api*/*schedule*/*day*?*type*=*group*&*identifier*=3-013&*weekType*=Чисельник&*dayOfWeek*=Понеділок демонструє, як система дозволяє отримати розклад на понеділок чисельника для групи 3-013. Такий підході забезпечує інтуїтивність використання *API* та мінімізує ймовірність помилок при формуванні запитів.

Формат відповіді для денного розкладу структурований як *JSON*-об'єкт з масивом *daySchedule*, де кожен елемент представляє окрему пару. Структура кожної пари включає унікальний ідентифікатор *id*, що дозволяє клієнтським додаткам ефективно відстежувати та оновлювати конкретні записи. Поле *lessonNumber* визначає порядковий номер пари в розкладі дня, що критично важливо для правильного відображення часової послідовності занять.

Список груп *groups* представлений як масив рядків, що дозволяє обробляти випадки спільних занять декількох груп. Часові рамки *startTime* та *endTime* у форматі *ISO* 8601 з точністю до мілісекунд та часовою зоною *UTC*. Наприклад, "08:30:00.000*Z*" для початку першої пари. Такий формат забезпечує однозначність інтерпретації часу незалежно від локальних налаштувань клієнтських додатків.

Назва предмету *subject* передається як простий рядок, що дозволяє включати повні назви дисциплін без обмежень довжини. Особливу увагу приділено структурі поля *teachers*, яке представлене як масив об'єктів з повними даними викладача. Кожен об'єкт викладача включає *lastName* (прізвище), *firstName* (ім'я), *middleName* (по батькові) та *initials* (скорочені ініціали). Така деталізація забезпечує гнучність відображення інформації в клієнтських додатках - від повних імен для детального перегляду до коротких ініціалів для компактного відображення.

Список аудиторій *classrooms* представлений як масив рядків, що дозволяє обробляти випадки проведення заняття в декількох аудиторіях одночасно або вказівки альтернативних приміщень.

Кінцева точка тижневого розкладу *GET* /*api*/*schedule*/*week* розширює концепцію денного розкладу, повертаючи структуровані дані на весь тиждень. Відповідь організована як масив *weekSchedule*, де кожен елемент представляє один

день тижня з полем *dayOfWeek* та масивом *items*, що містить пари для цього дня. Поле *dayOfWeek* використовує повні назви днів українською.

Система замін представлена кінцевою точкою *GET /api/substitutions* з розширеною функціональністю фільтрації. Параметр *type* підтримує три значення: *group*, *teacher* та *all*, що дозволяє отримувати заміни для конкретної групи, викладача або всі заміни системи. Режим *all* особливо корисний для адміністративного персоналу, який потребує повного огляду всіх замін у навчальному закладі. Опціональний параметр *date* забезпечує фільтрацію замін за конкретною датою у форматі *YYYY-MM-DD* .

Структура відповіді замін є однією з найскладніших в системі, оскільки включає інформацію як про оригінальне заняття, так і про заміну. Поля *originalSubject* та *substituteSubject* зберігають назви предметів до та після заміни, що дозволяє користувачам зрозуміти, який саме предмет було замінено. Аналогічно, *originalTeachers* та *substituteTeachers* містять інформацію про викладачів у тій же детальній структурі, що й в основному розкладі. Поле *reason* містить текстове пояснення причини заміни, наприклад “Хвороба викладача” або “Службове відрядження”.

Кінцева точка практик *GET /api/practice* надає інформацію про виробничі та навчальні практики здобувачів освіти. Структура відповіді включає поле *practiceType*, яке розрізняє типи практик: “Навчальна практика”, “Виробнича практика” тощо. Поле *practicePlace* містить детальну інформацію про місце проведення практики, включаючи назву підприємства, адресу або спеціальні вказівки. Поле *practiceShift* визначає робочу зміну під час практики.

Часові рамки практики визначаються полями *practiceStartDate* та *practiceEndDate* у форматі *YYYY-MM-DD*. Додаткове поле *note* надає можливість включати важливі примітки, інструкції або додаткові вимоги до проходження практики.

Система іспитів та консультацій представлена кінцевою точкою *GET /api/exams*, що повертає об’єднану інформацію про екзамени та консультації в рамках екзаменаційної сесії. Поле *eventType* розрізняє тип події за допомогою

значень “Екзамен” або “Консультація”. Поле *dateTime* містить точний час проведення у форматі *ISO* 8601 з часовою зоною, наприклад “2025-06-16*T*12:00:00*Z*” . Поле *subjectName* містить повну назву дисципліни, для якої проводиться екзамен або консультація.

Довідкові кінцеві точки надають допоміжну інформацію, необхідну для повноцінного функціонування системи. Кінцева точка *GET* /*api*/*timetable* повертає загальний розклад дзвінків, структурований по днях тижня з детальною інформацією про час початку та закінчення кожної пари. Поле *eventType* розрізняє типи подій: “Пара”, “Перерва”, “Велика перерва”.

Розклад їдальні, доступний через *GET* /*api*/*canteen*, організований аналогічно з розподілом по днях тижня та типах прийомів їжі. Поле *mealType* включає значення “Сніданок”, “Обід”, “Вечеря”, що відображає повний цикл харчування. Часові інтервали визначаються полями *startTime* та *endTime* у форматі *HH*:*MM*:*SS*.

Кінцева точка *GET* /*api*/*week-intervals* надає критично важливу інформацію про структуру навчального року, повертаючи список тижневих інтервалів з їх типами. Кожен інтервал включає поля *startDate* та *endDate* у форматі *YYYY-MM-DD*, що чітко визначає межі кожного тижня, та поле *weekType* зі значеннями “Чисельник” або “Знаменник”.

Довідкова інформація про групи та викладачів надається через кінцеві точки *GET* /*api*/*groups* та *GET* /*api*/*teachers*. Список груп структурований за відділеннями, де поле *departmentName* містить назву відділення, а масив *groups* - список груп цього відділення.

2.1.3 Проєктування та реалізація кінцевих точок *API*

Проєктування та реалізація кінцевих точок *API* становить основу функціональності всього сервісу, визначаючи способи взаємодії зовнішніх клієнтів із системою управління навчальним розкладом. Архітектурний підхід до реалізації кінцевих точок базується на принципі модульності та розділення відповідальності.

У головному файлі додатку *main.py* всі роутери інтегруються до основного екземпляра *FastAPI* через метод *app.include_router*() з загальним префіксом /*api*

який чітко відділяє *API*-кінцеві точки від можливих статичних ресурсів або веб-інтерфейсу, забезпечує версіонування *API* та спрощує налаштування зворотного проксі-сервера.

Кожна кінцева точка реалізується як асинхронна функція *Python*, декорована відповідним *HTTP*-методом *FastAPI*. Використання асинхронних функцій є критично важливим для продуктивності системи, оскільки дозволяє серверу обробляти інші запити під час очікування відповіді від бази даних або зовнішніх сервісів. Декоратори *FastAPI*, такі як *@router.get*(), *@router.post*(), не лише визначають *HTTP*-маршрути, але й надають можливості для автоматичної валідації параметрів, генерації документації *OpenAPI* та обробки помилок.

Для взаємодії з даними є клас *DatabaseHandler*, розташований у модулі *database*/*manager.py*. Цей клас інкапсулює всю логіку доступу до даних та надає високорівневий інтерфейс для роботи з різними сутностями системи. Усі методи класу є статичними та асинхронними.

Спочатку *FastAPI* автоматично валідує параметри запиту відповідно до типів, визначених у сигнатурі функції. Якщо параметри невалідні, автоматично повертається помилка 422 *Unprocessable Entity* з детальним описом проблем валідації. Після успішної валідації функція-обробник викликає відповідний метод *DatabaseHandler* для отримання даних з бази даних. Автоматичну документацію *API Swagger UI* показано на рисунку 2.1.

Кінцева точка *GET* /*api*/*substitutions* приймає параметри *type*, *identifier* та опціональний *date*. Спочатку перевіряється валідність параметра *type*, який має бути одним із значень: “*group*”, “*teacher*” або “*all*”. Для типів “*group*” і “*teacher*” параметр *identifier* є обов’язковим.

Якщо запитуються заміни для конкретної групи, функція спочатку викликає *await DatabaseHandler.get_group_by_name*(*identifier*) для перевірки існування групи в системі. Цей крок є критично важливим для забезпечення коректності даних та інформативності повідомлень про помилки. Якщо група не знайдена, кидається виключення *DataNotFoundError*, яке потім обробляється та перетворюється на *HTTP*-відповідь з кодом 404.

Після валідації параметрів викликається основний метод отримання даних *await DatabaseHandler.get_substitutions_by_group*(*identifier*, *date*). Цей метод повертає список об'єктів *SQLAlchemy* моделі *Substitution* з усіма пов'язаними даними.

Трансформація даних є складним процесом, що включає витягування інформації з пов'язаних сутностей та форматування її у структуру, визначену *Pydantic*-моделями. Для кожної заміни необхідно сформувати список груп, витягнувши назви з пов'язаних об'єктів груп. Інформація про викладачів трансформується в структуру *TeacherInfo* з повними іменами та ініціалами. Час початку та закінчення пари отримується з розкладу дзвінків через додатковий запит *await DatabaseHandler.get_bell_schedule_pairs_map*().

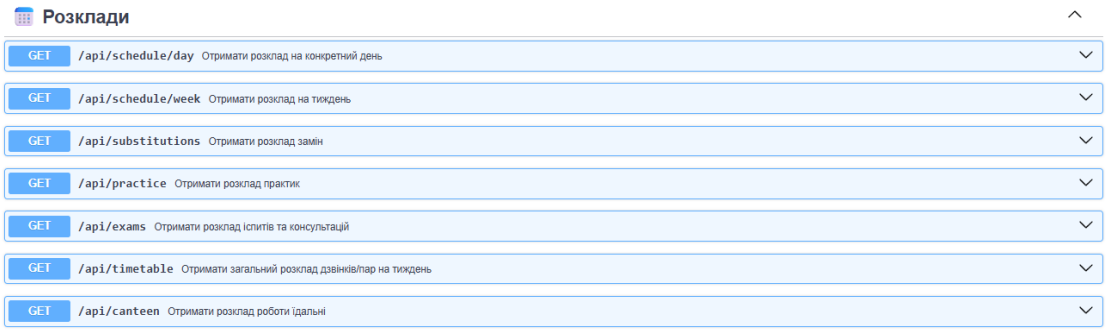

Рисунок 2.1 – Автоматична документація *API Swagger UI*

Обробка помилок у кінцевих точках реалізована через систему винятків та їх перехоплення. Специфічні бізнес-помилки, такі як *DataNotFoundError*, перехоплюються та перетворюються в інформативні *HTTP*-відповіді з відповідними статус-кодами. Загальні системні помилки логуються для подальшого аналізу адміністраторами, але клієнту повертається узагальнене повідомлення про внутрішню помилку сервера, щоб уникнути витоку чутливої інформації про архітектуру системи.

2.1.4 Впровадження механізмів кешування

Кешування дозволяє значно скоротити час відповіді на запити користувачів, зменшити навантаження на базу даних *PostgreSQL* та підвищити загальну

пропускну здатність системи. Особливо це актуально для освітнього закладу, де кількість одночасних запитів до розкладу може бути значною, особливо в пікові години на початку навчального дня або під час сесії.

Архітектура кешування системи побудована на використанні *Redis* як основного сховища кешованих даних. Інтеграція кешування з *FastAPI* реалізована через спеціалізовану бібліотеку *fastapi-cache*2, яка надає зручні декоратори та методи для роботи з кешем безпосередньо в контексті веб-фреймворку. Ця бібліотека автоматично обробляє серіалізацію та десеріалізацію даних, генерацію ключів кешу на основі параметрів запиту та керування життєвим циклом кешованих записів. Використання *fastapi-cache*2 значно спрощує процес впровадження кешування, оскільки не потребує написання додаткового коду для перетворення *Python*-об'єктів у формат, придатний для зберігання в *Redis*.

Конфігурація системи кешування централізовано зосереджена у модулі *config*/*cache.py*. Функція *init_redis_cache* відповідає за ініціалізацію з'єднання з сервером *Redis*, використовуючи параметри підключення, які зчитуються зі змінних оточення *REDIS_HOST* та *REDIS_PORT*. Асинхронний клієнт *Redis* створюється з параметрами *encoding*="*utf*-8" та *decode_responses=True* для коректної роботи з *Unicode*-текстом.

Стратегія кешування в системі базується на диференційованому підході до різних типів даних залежно від їх частоти зміни та важливості актуальності. Для відносно статичних даних, таких як список груп, інформація про викладачів, розклад дзвінків або графік роботи їдальні, застосовується тривале кешування з терміном життя від декількох хвилин до годин. Наприклад, кінцева точка отримання списку всіх груп /*api*/*groups* може бути кешований на 30 хвилин, оскільки структура груп у навчальному закладі змінюється рідко.

Для даних, що змінюються частіше, таких як основний розклад занять, застосовується помірне кешування з терміном життя 5-15 хвилин. Найбільш динамічні дані, такі як розклад замін, потребують особливого підходу до кешування. Оскільки заміни можуть додаватися протягом дня і є критично важливою інформацією для здобувачів освіти та викладачів, для них

встановлюється короткий термін кешування (1-3 хвилини) або використовується механізм активної інвалідації кешу при додаванні нових замін.

Для забезпечення актуальності кешованих даних у системі реалізований механізм селективної інвалідації кешу. Функція *invalidate_cache* приймає параметр *namespace* та видаляє з *Redis* всі ключі, що належать до вказаного простору імен. Це дозволяє точково очищувати лише ті частини кешу, які стали неактуальними внаслідок змін у відповідних даних.

2.2 *Telegram*-бот

Telegram-бот системи “єРозклад” являє собою ключовий компонент архітектури, що забезпечує безпосередню взаємодію з кінцевими користувачами системи – здобувачами освіти та викладачами навчального закладу. Основним завданням бота є надання інтуїтивного та зручного інтерфейсу для доступу до функціональності *API* сервісу через популярний месенджер *Telegram*. Архітектурне рішення щодо створення *Telegram*-бота базується на використанні сучасної асинхронної бібліотеки *aiogram*, яка є одним з найпотужніших та найбільш гнучких фреймворків для розробки ботів на *Python*.

2.2.1 Архітектура та структура проєкту

Архітектурна організація *Telegram*-бота ґрунтується на модульному підході, де кожен компонент системи відповідає за конкретну область функціональності. Центральними елементами архітектури є обробники команд та повідомлень, які містять логіку реакції бота на різноманітні типи користувацьких запитів. Ці обробники організовані у відповідності до принципу єдиної відповідальності, де кожен модуль обробників займається специфічним аспектом взаємодії з користувачем. Наприклад, обробники команд типу /*start* або /*today* зосереджені на базових операціях, тоді як обробники *callback*-запитів від *inline*-клавіатур керують інтерактивними елементами інтерфейсу.

Важливою складовою архітектури є система управління станами користувачів, реалізована за допомогою машини скінченних станів. Ця система дозволяє боту запам’ятовувати контекст поточного діалогу з користувачем та проводити його через послідовність кроків для досягнення певної мети. Типовими сценаріями використання *FSM* є процес первинної реєстрації користувача, коли бот послідовно запитує роль користувача, його навчальну групу або статус викладача, налаштування сповіщень та інші параметри. Аналогічно, *FSM* використовується для реалізації функцій пошуку розкладу за критеріями, де користувач крок за кроком уточнює параметри пошуку.

Система взаємодії з зовнішніми сервісами представлена спеціалізованими модулями *ApiClient* та *APICache*, які забезпечують надійну та ефективну комунікацію з *API* сервісом системи. *ApiClient* інкапсулює логіку формування *HTTP*-запитів до різних кінцевих точок *API*, включаючи обробку помилок та таймаутів. *APICache* реалізує інтелігентне кешування відповідей від *API* в *Redis*, що значно покращує швидкість роботи бота та зменшує навантаження на серверну частину системи.

Робота з базою даних *PostgreSQL* організована через окремий шар абстракції, представлений класом *DatabaseHandler*. Цей компонент відповідає за зберігання та управління персональними даними користувачів, їхніми налаштуваннями, ролями в системі та історією взаємодій. Використання *SQLAlchemy* як *ORM* забезпечує типобезпечність операцій з базою даних та спрощує управління схемою даних.

Система сповіщень являє собою окремий архітектурний блок, що забезпечує отримання та обробку повідомлень від інших компонентів екосистеми через брокер повідомлень *NATS*. Цей механізм дозволяє боту реагувати на зміни в розкладі, появу нових замін або інші події в режимі реального часу, автоматично інформуючи зацікавлених користувачів.

Допоміжні утиліти включають модулі для форматування даних, генерації клавіатур різних типів, налаштування системи логування, реалізації антиспам-механізмів та управління правами доступу до адміністративних функцій. Ці компоненти забезпечують якість користувацького досвіду та безпеку системи.

Структура файлової системи проєкту *Telegram*-бота організована таким чином, щоб забезпечити максимальну зрозумілість та зручність навігації по коду. Кореневий каталог *src* містить всі вихідні файли, організовані за функціональним принципом. Файл *main.py* є точкою входу в програму та відповідає за ініціалізацію всіх компонентів системи, налаштування *middleware* компонентів, реєстрацію обробників подій та запуск основного циклу роботи бота.

Каталог *api* містить модулі для взаємодії з зовнішнім *API* сервісом. Модуль *api_client.py* реалізує клас *ApiClient*, що забезпечує асинхронні *HTTP*-запити до всіх необхідних кінцевих точок *API* сервісу. Кожен метод цього класу відповідає конкретному типу запиту, такому як отримання розкладу занять, списку замін, інформації про викладачів або навчальні групи. Модуль *api_cache.py* містить реалізацію системи кешування *APICache*, яка використовує *Redis* для зберігання часто запитуваних даних та зменшення кількості звернень до *API*.

Конфігураційні параметри системи зосереджені в каталозі *config*. Файл *config.py* містить всі глобальні константи, включаючи токени доступу, параметри підключення до баз даних та зовнішніх сервісів, тексти повідомлень та налаштування за замовчуванням. Модулі *bot.py*, *nats.py* та *redis.py* відповідають за ініціалізацію відповідних компонентів системи.

Каталог *database* об’єднує всі компоненти для роботи з базою даних. Файл *models.py* містить визначення всіх моделей *SQLAlchemy*, що відображають структуру таблиць бази даних. Модуль *engine.py* забезпечує створення підключення до бази даних та налаштування сесій. Клас *DatabaseHandler* в файлі *manager.py* інкапсулює всю логіку взаємодії з базою даних, надаючи високорівневий інтерфейс для операцій створення, читання, оновлення та видалення даних. Сервіс *UserDataService* в окремому файлі реалізує кешування даних користувачів для оптимізації продуктивності.

Найбільший за обсягом каталог *handlers* містить всі обробники подій бота, розподілені за функціональними областями. Файл *registration_handlers.py* містить логіку процесу реєстрації та оновлення профілів користувачів. Модулі *schedule_handlers.py* та *extra_schedule_handlers.py* відповідають за обробку запитів

на отримання різних типів розкладу. Файли *additional_handlers.py* та *options_handlers.py* містять обробники для додаткових функцій та налаштувань. Адміністративні функції винесені в окремий модуль *management_handlers.py*.

Каталог *middlewares* містить компоненти антиспам-фільтра, що запобігає зловживанням з боку користувачів. Каталог *services* об'єднує сервіси для обробки різних типів сповіщень від зовнішніх систем. Каталог *utils* містить допоміжні утиліти для форматування даних, генерації клавіатур, налаштування логування та інших допоміжних функцій.

2.2.2 Проєктування та реалізація механізмів взаємодії з *API*

Основним принципом проєктування механізмів взаємодії з *API* стало забезпечення неблокуючої роботи бота. Це означає, що під час виконання запитів до зовнішнього *API* сервісу основний потік виконання бота не припиняє свою роботу і може продовжувати обробляти запити від інших користувачів. Для досягнення цієї мети було обрано асинхронний підхід до програмування з використанням можливостей *Python asyncio* та спеціалізованих бібліотек для *HTTP*-комунікації.

Для взаємодії з *API* є клас *ApiClient*, розташований у модулі *api_client.py*. Цей клас приховує складність формування *HTTP*-запитів та обробки відповідей за простим та зрозумілим інтерфейсом. Для виконання асинхронних *HTTP*-запитів *ApiClient* використовує бібліотеку *httpx*, яка є сучасною альтернативою популярної бібліотеки *requests* з повною підтримкою асинхронних операцій.

Архітектура *ApiClient* побудована навколо приватного методу *_get*, який служить універсальним механізмом для виконання *GET*-запитів до будь-якої кінцевої точки *API*. Цей метод приймає як параметри відносний шлях до кінцевої точки та словник параметрів запиту, автоматично формуючи повний *URL* та виконуючи *HTTP*-запит з необхідними заголовками та налаштуваннями таймауту.

Кожен тип даних, що отримується з *API*, має відповідний спеціалізований метод в класі *ApiClient*. Клас *ApiClient* для асинхронної роботи із зовнішнім *API* показано на рисунку 2.2. Метод *get_schedule* забезпечує отримання розкладу занять

для конкретної групи або викладача на певний день та тип тижня. Метод *get_substitutions* дозволяє запитувати інформацію про заміни в розкладі з можливістю фільтрації за різними критеріями. Методи *get_practice* та *get_exams* призначені для отримання розкладів практик та іспитів відповідно.

Важливим аспектом реалізації *ApiClient* є використання патерна асинхронного контекстного менеджера, коли клас реалізує методи *aenter* та *aexit*, що дозволяє використовувати його в конструкції *async with*.

```
     class ApiClient:
         def __init__(self, base_url: str = API_URL, timeout: int =
10):
             self.base_url = base_url
             self.timeout = timeout
            self.client = httpx.AsyncClient(base_url=self.base_url,
timeout=self.timeout)

         async def __aenter__(self):
             return self

         async def __aexit__(self, exc_type, exc, tb):
             await self.close()

         async def close(self):
             """Коректно закриває клієнт (звільняє ресурси)."""
             await self.client.aclose()

         async def _get(self, endpoint: str, params: dict[str, Any]
| None = None) -> dict[str, Any]:
             """
             Внутрішній метод для відправлення GET-запитів.
                  """
             try:
                       response  =  await  self.client.get(endpoint,
params=params)
                 response.raise_for_status()
                 return response.json()
             except httpx.HTTPError as e:
                  logger.error(f"Помилка при запиті до {endpoint}:
{str(e)}")
                raise Exception(f"Помилка при запиті до {endpoint}:
{str(e)}")
```

Рисунок 2.2 – Клас *ApiClient* для асинхронної роботи із зовнішнім *API*

Інтеграція механізмів взаємодії з *API* в обробники команд бота здійснюється через прямий виклик методів *ApiClient* або *APICache* в залежності від типу необхідних даних та вимог до їх актуальності.

2.2.3 Розробка сценаріїв взаємодії та обробників команд

Процес первинної реєстрації та налаштування користувача запускається командою /*start* або натисканням кнопки “Змінити вибір” в головному меню. Цей сценарій розроблений як багатоетапний діалог, що проводить користувача через послідовність налаштувань для персоналізації роботи з системою. Перший етап передбачає вибір ролі користувача в навчальному процесі – здобувач освіти або викладач. Цей вибір критично впливає на подальший хід діалогу та доступні функції системи. Початок реєстрації показано на рисунку 2.3.

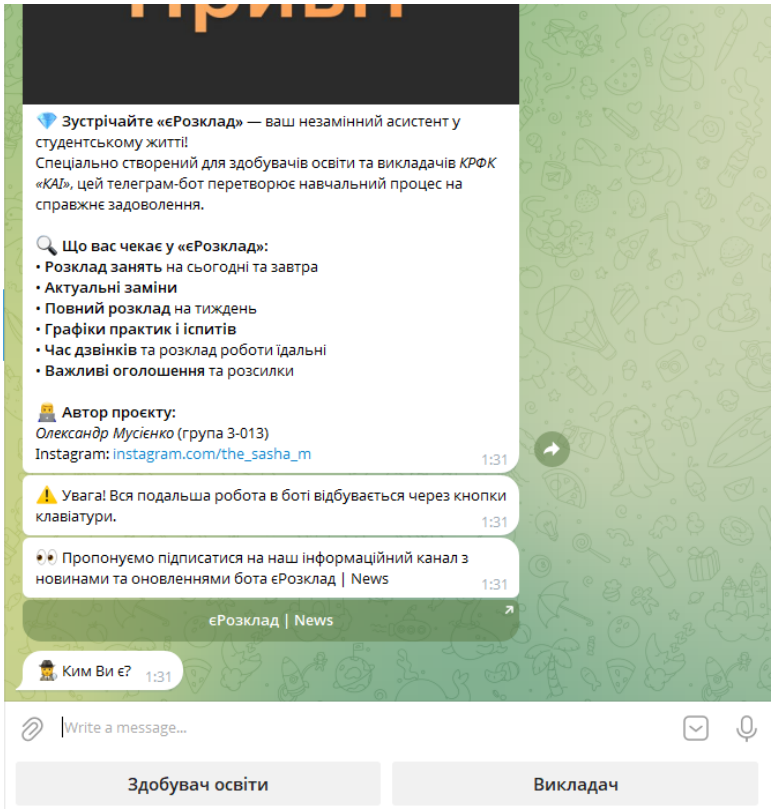

Рисунок 2.3 – Початок реєстрації

Для користувачів зі статусом здобувача освіти наступний етап включає вибір навчального відділення з динамічно сформованого списку, отриманого через *APICache*. Вибір групи показано на рисунку 2.4. Після вибору відділення система надає список доступних навчальних груп для цього відділення, дозволяючи користувачеві обрати свою конкретну групу. Вибір групи показано на рисунку 2.5. Для викладачів процес спрощується до введення особистих ініціалів.

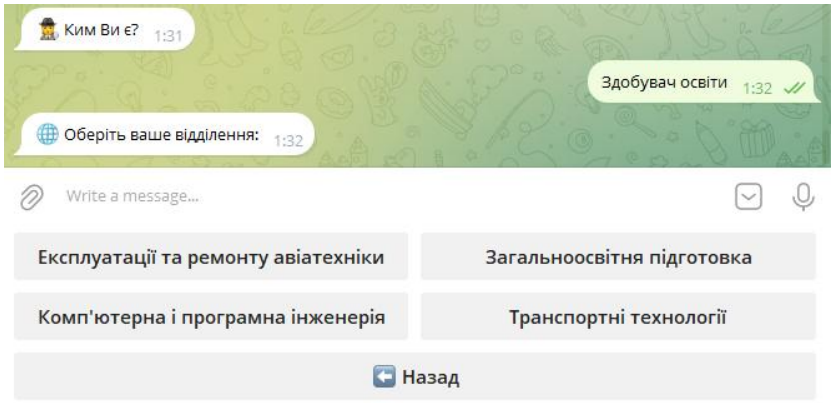

Рисунок 2.4 – Вибір відділення

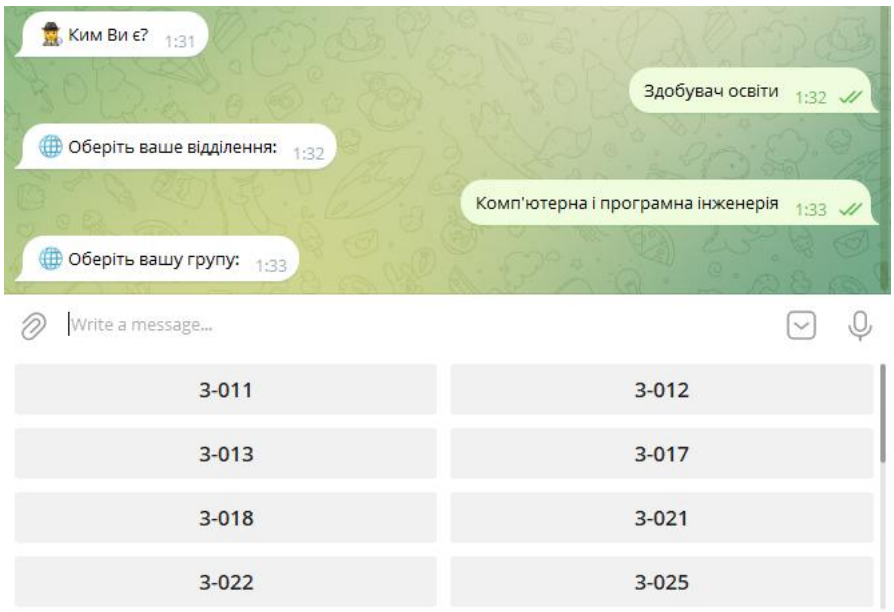

Рисунок 2.5 – Вибір групи

Завершальні етапи сценарію реєстрації включають налаштування персональних параметрів роботи системи. Користувач може вибрати, чи хоче він отримувати автоматичні сповіщення про зміни в розкладі, та визначити формат відображення імен викладачів – повні імена або тільки ініціали. Всі налаштування зберігаються в базі даних та використовуються для персоналізації подальшої взаємодії з системою. Меню налаштувань користувача показано на рисунку 2.6.

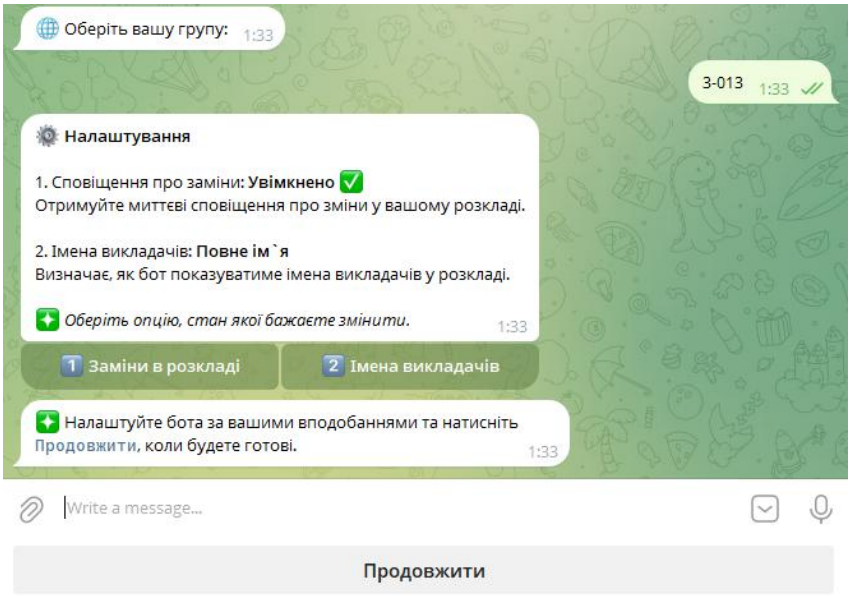

Рисунок 2.6 – Меню налаштувань користувача

Базовий сценарій отримання розкладу на поточний або наступний день активується натисканням відповідних кнопок в головному меню або введенням текстових команд. Система автоматично визначає поточну дату, тип навчального тижня та отримує збережені налаштування користувача для формування персоналізованого запиту до *API* сервісу. Розклад на поточний день із замінами показано на рисунку 2.7.

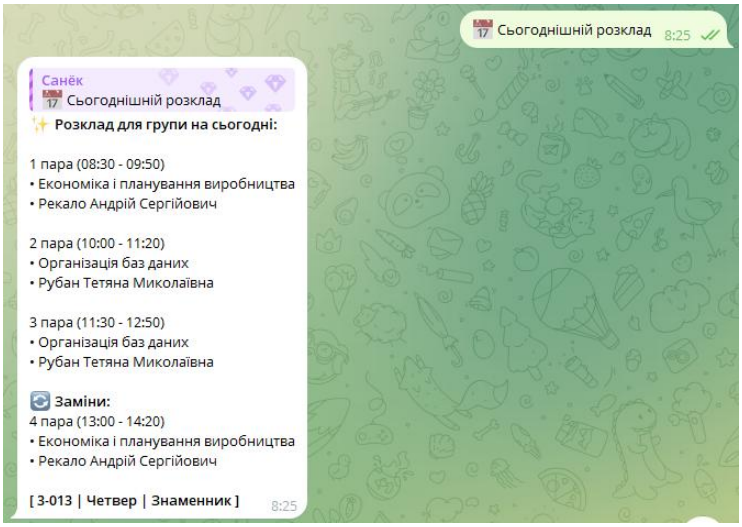

Рисунок 2.7 – Розклад на поточний день із замінами

Розширений сценарій перегляду повного тижневого розкладу забезпечує інтерактивну навігацію між днями тижня та типами тижнів за допомогою *inline-*

клавіатури. Відображення повного розкладу групи показано на рисунку 2.8. Користувач може легко переключатися між різними днями.

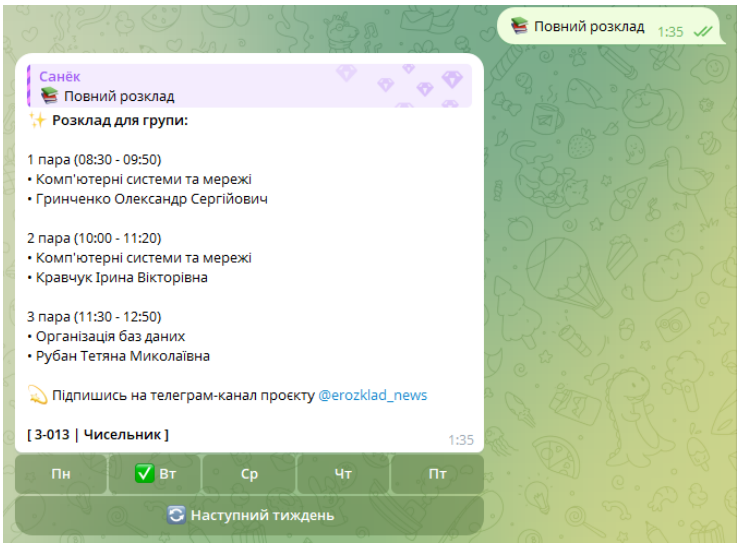

Рисунок 2.8 – Відображення повного розкладу групи

Функції пошуку розкладу реалізують складніші сценарії взаємодії, що включають кілька етапів вводу та валідації даних. Користувач спочатку обирає тип пошуку – за навчальною групою або за викладачем. Після цього система переходить в режим очікування вводу конкретного ідентифікатора та валідує введені дані. У випадку успішної валідації система відображає знайдений розклад з можливістю інтерактивної навігації.

Реалізація обробників команд та повідомлень здійснюється через систему декораторів *aiogram*, що дозволяє прив’язувати функції обробки до конкретних типів подій або користувацьких дій.

Обробники повідомлень використовують систему фільтрів для визначення, на які саме повідомлення вони повинні реагувати. Фільтри можуть базуватися на тексті повідомлення, типі команди, поточному стані користувача в *FSM* або комбінації цих критеріїв. Наприклад, обробник команди /*start* активується як при введенні текстової команди, так і при натисканні відповідної кнопки в меню.

Обробники *callback*-запитів спеціалізуються на роботі з *inline*-клавіатурами та інтерактивними елементами інтерфейсу. Кожна кнопка *inline*-клавіатури має унікальний ідентифікатор, що передається в *callback_data* при натисканні.

Обробники аналізують цей ідентифікатор для визначення необхідної дії та можуть динамічно змінювати вміст повідомлення або клавіатури без створення нових повідомлень.

2.2.4 Система отримання та обробки сповіщень

Система сповіщень у *Telegram*-боті є критично важливим компонентом для забезпечення своєчасного інформування користувачів про зміни в розкладі, початок практичних занять та інші академічні події. Архітектура системи побудована на принципах асинхронного обміну повідомленнями через *NATS* та використанні асинхронних черг для надійного розподілу навантаження при масових розсилках.

Центральним елементом системи є *NATS* слухач, реалізований у модулі *src/config/nats.py*. Цей компонент забезпечує постійне підключення до сервера *NATS*, адреса якого конфігурується через змінну оточення *NATS_SERVER_URL*. Функція *nats_listener* ініціалізує клієнт *NATS* та створює підписки на специфічні теми для отримання різноманітних типів сповіщень.

Бот підписується на дві основні теми *NATS*. Перша тема, визначена константою *SUBSTITUTIONS_NATS_SUBJECT*, призначена для отримання повідомлень про нові заміни в розкладі від *API* сервісу. Ці повідомлення обробляються спеціалізованою функцією *substitutions_message_handler* з модуля *src/services/substitutions_notifier.py*.

Друга тема, *GMAIL_CONNECTOR_NATS_SUBJECT*, використовується для отримання сповіщень від сервісу інтеграції з *Gmail* про надходження нових листів, що потенційно містять інформацію про заміни. Сповіщення про виявлення нових замін у розкладі показано на рисунку 2.9. Обробка цих повідомлень здійснюється функцією *gmail_substitutions_handler* з модуля *src/services/gmail_notifier.py*.

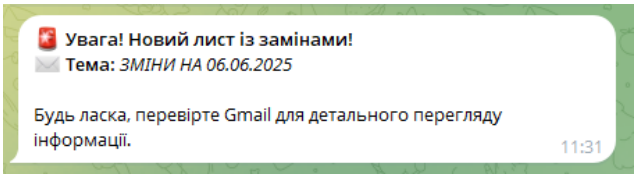

Рисунок 2.9 – Сповіщення про виявлення нових замін у розкладі

Обробка сповіщень про заміни в розкладі реалізована як багатоетапний процес з обов'язковим підтвердженням від розробника системи. Коли *API* сервіс додає нові заміни до бази даних, він формує повідомлення у форматі *JSON*, яке містить списки груп та викладачів, які потребують інформування про зміни. Функція *substitutions_message_handler* отримує це повідомлення, декодує його та виконує запит до бази даних через метод *DatabaseHandler.get_users_for_substitutions*(*groups*, *teachers*) для визначення користувачів, які мають отримати сповіщення.

Критично важливим елементом процесу є механізм підтвердження розсилки розробником. Система формує детальне повідомлення для розробника, ідентифікатор якого зберігається у константі *DEVELOPER_ID*, з інформацією про кількість потенційних отримувачів та перелік груп і викладачів, які будуть проінформовані. Для забезпечення безпеки та контролю над розсилками генерується унікальний ідентифікатор *broadcast_id*, а всі дані, необхідні для розсилки, тимчасово зберігаються в *Redis* з коротким терміном життя у п'ять хвилин. Розробнику надсилається повідомлення з *inline*-клавіатурою, що містить кнопки підтвердження та відхилення розсилки. Звіт про успішне виконання розсилки замін користувачам показано на рисунку 2.10.

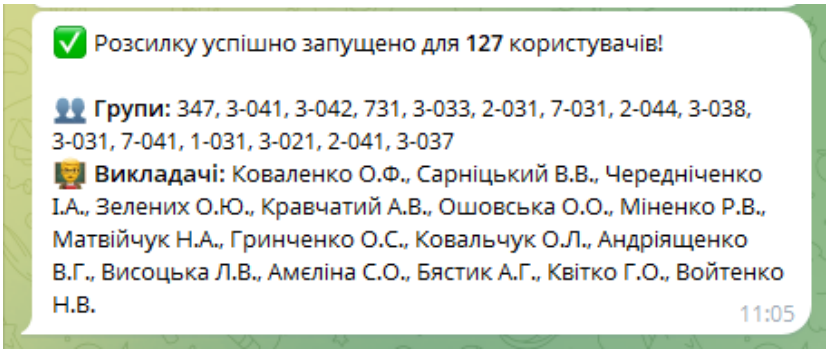

Рисунок 2.10 – Звіт про успішне виконання розсилки замін користувачам

У разі підтвердження розсилки *callback*-обробник *process_confirm_broadcast* витягує збережені дані з *Redis* та для кожного користувача з переліку додає завдання до асинхронної черги *notification_queue*. Повідомлення, що надсилається

користувачам, формується на основі шаблону *SUBSTITUTIONS_NOTIFICATION* з підтримкою *HTML*-розмітки. Після успішного запуску розсилки дані видаляються з *Redis*, а повідомлення розробнику оновлюється інформацією про статус операції. Функція скасування *process_cancel_broadcast* дозволяє розробнику відхилити розсилку, при цьому всі тимчасові дані також видаляються з кешу.

Обробка повідомлень від *Gmail* реалізована значно простіше, оскільки ці сповіщення мають інформаційний характер. Коли сервіс інтеграції з *Gmail* виявляє новий лист, що потенційно містить заміни, функція *gmail_substitutions_handler* декодує отримане *JSON*-повідомлення та витягує тему листа. Розробнику системи надсилається коротке інформаційне повідомлення про надходження нового листа з замінами, що служить сигналом для запуску процедури обробки замін. Сповіщення про заміни в розкладі показано на рисунку 2.11.

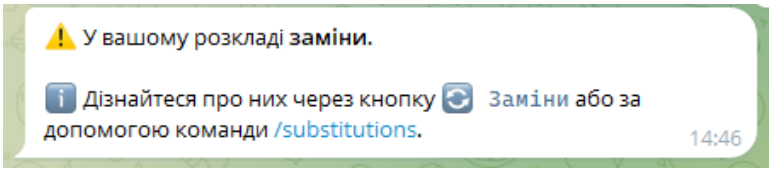

Рисунок 2.11 – Сповіщення про заміни в розкладі

Система сповіщень про початок практики працює за розкладом та використовує планувальник завдань *APScheduler*. Функція *start_practice_notifications* налаштована на виконання щочетверга о 10:00 ранку та здійснює запит до кінцевої точки */api/practice/next-week* для отримання актуальної інформації про групи та викладачів, у яких наступного тижня починається практика. Після отримання даних використовується метод *DatabaseHandler.get_users_for_practice_notifications*(*groups*, *teachers*) для визначення користувачів, які мають бути проінформовані про початок практичних занять. Для кожного знайденого користувача формується завдання на надсилання стандартного повідомлення з шаблону *EVENT_MESSAGES* з *HTML*-розміткою, яке додається до черги сповіщень.

Центральним компонентом надійної доставки сповіщень є асинхронна черга та диспетчер, реалізовані у модулі *src/utils/notification_dispatcher.py*. Система

використовує стандартну асинхронну чергу *asyncio.Queue* під назвою *notification_queue*, яка забезпечує неблокуючу обробку завдань на надсилання повідомлень.

Функція *process_notification_queue* працює у нескінченному асинхронному циклі, постійно дістаючи завдання з черги та обробляючи їх. Система підтримує надсилання як звичайних текстових повідомлень, так і медіагруп з фото або відео. Якщо список медіафайлів не порожній, формується медіагрупа з об'єктів *InputMediaPhoto* або *InputMediaVideo*, при цьому текст повідомлення додається як підпис до першого елемента групи. Надсилання здійснюється через метод *bot.send_media_group*(), після чого користувачу також надсилається головна клавіатура бота. Для звичайних текстових повідомлень використовується метод *bot.send_message*() з відповідним режимом парсингу та головною клавіатурою.

2.2.5 Реалізація антиспам-фільтра

Middleware-компонент *AntiSpamMiddleware* інтегрується в процес обробки повідомлень бібліотеки *aiogram* як проміжний шар між отриманням повідомлення від *Telegram API* та його передачею до відповідного обробника. Цей підхід забезпечує централізовану фільтрацію без необхідності модифікації існуючих обробників команд та повідомлень, що підтримує принцип розділення відповідальностей та спрощує підтримку коду.

Клас *AntiSpamMiddleware* ініціалізується трьома ключовими параметрами конфігурації, які визначають поведінку антиспам-системи. Параметр *limit* встановлює максимально допустиму кількість повідомлень від одного користувача протягом певного часу. Параметр *period* визначає тривалість часового вікна в секундах. Параметр *block_time* встановлює тривалість блокування користувача в секундах у разі виявлення спаму або флуду.

Словник *self.blocked_users* відстежує статус блокування користувачів, зберігаючи часову мітку моменту блокування або значення *False* для незаблокованих користувачів. Код фільтра з алгоритмом блокування користувачів показано на рисунку 2.12.

Основний алгоритм фільтрації реалізований у методі *on_process_message*, який викликається для кожного вхідного повідомлення. Спочатку з об'єкта повідомлення витягується унікальний ідентифікатор користувача та фіксується поточний час. Система перевіряє, чи знаходиться користувач у списку заблокованих, і якщо так, то обчислює час, що минув з моменту блокування. Інтеграція *middleware* в систему бота здійснюється в головному файлі *main.py* під час ініціалізації диспетчера *aiogram*.

```
    class AntiSpamMiddleware(BaseMiddleware):
        def __init__(self, limit, period, block_time):
            self.limit = limit
            self.period = period
            self.block_time = block_time
            self.user_data = defaultdict(list)
            self.blocked_users = defaultdict(lambda: False)
            super().__init__()

        async def on_process_message(self, message: types.Message,
data: dict):
            user_id = message.from_user.id
            now = datetime.now()

            # Перевіряємо, чи не заблокований користувач
            if self.blocked_users[user_id]:
                        if  now  -  self.blocked_users[user_id]  <
timedelta(seconds=self.block_time):
                    raise CancelHandler()
                else:
                    self.blocked_users[user_id] = False

            # Додаємо поточне повідомлення до історії користувача
            self.user_data[user_id].append(now)

            # Видаляємо старі повідомлення
            self.user_data[user_id] = [
                msg_time for msg_time in self.user_data[user_id] if
now - msg_time <= timedelta(seconds=self.period)
            ]

            # Перевіряємо на флуд
            if len(self.user_data[user_id]) > self.limit:
                if not self.blocked_users[user_id]:
                     await message.reply("🚫 Увага! Виявлено спам!
Будь ласка, не надсилайте команди так часто.")
                    self.blocked_users[user_id] = now
                    raise CancelHandler()
```

Рисунок 2.12 – Код фільтра з алгоритмом блокування користувачів

РОЗДІЛ 3
РОЗГОРТАННЯ ТА БЕЗПЕРЕРВНА ІНТЕГРАЦІЯ

3.1 Створення *Dockerfile* для кожного сервісу

Контейнеризація є критично важливим аспектом сучасної розробки програмного забезпечення, особливо для мікросервісних архітектур. У системі управління навчальним розкладом кожен компонент потребує індивідуального підходу до створення *Docker*-образів, що забезпечує їх портативність, ізоляцію та відтворюваність незалежно від інфраструктури розгортання.

API сервіс системи управління розкладом є найбільш критичним компонентом, що обслуговує основну бізнес-логіку та надає *REST API* для взаємодії з іншими сервісами та клієнтськими додатками. *Dockerfile* для цього сервісу побудований з акцентом на продуктивність та стабільність роботи під навантаженням.

Як базовий образ обрано *python*:3.11-*slim*, який представляє компромісне рішення між розміром образу та функціональністю. На відміну від повних образів *Python*, *slim*-варіант містить лише найнеобхідніші системні компоненти, що значно зменшує загальний розмір образу. Водночас, на відміну від *Alpine*-образів, *slim*-варіант базується на *Debian*, що забезпечує кращу сумісність з бінарними залежностями *Python*-пакетів, особливо тими, що містять C-розширення.

Для управління залежностями використовується сучасний інструмент *uv*, який значно перевершує традиційний *pip* за швидкістю встановлення пакетів та вирішення конфліктів залежностей. Бінарні файли *uv* копіюються безпосередньо з офіційного образу *ghcr*.*io*/*astral*-*sh*/*uv*:0.6.6, що гарантує використання конкретної версії інструменту та уникає потреби в його встановленні через пакетний менеджер системи.

Структура *Dockerfile* передбачає багатоетапну збірку, де перший етап (*builder*) відповідає за встановлення всіх залежностей проєкту. Команда *uv sync –frozen –no-install-project –no-editable –no-cache* виконує синхронізацію залежностей відповідно до *lockfile*, забезпечуючи детерміністичність збірки. Прапор *–frozen* запобігає будь-яким змінам у версіях залежностей порівняно з зафіксованими у *uv.lock*, що критично важливо для відтворюваності збірок у різних середовищах.

На фінальному етапі збірки копіюється лише готове віртуальне середовище та код додатку, виключаючи всі допоміжні інструменти та файли. Змінна середовища *PATH* модифікується для включення шляху до бінарних файлів віртуального середовища, що дозволяє викликати команди безпосередньо без активації середовища.

Запуск *API* здійснюється через *uvicorn* - високопродуктивний *ASGI*-сервер, оптимізований для асинхронних *Python*-додатків. Параметри *–host* 0.0.0.0 та *–port* 80 забезпечують прослуховування всіх мережевих інтерфейсів контейнера на стандартному *HTTP*-порту, що спрощує налаштування мережі в оркестраторах контейнерів.

Для *Telegram*-бота обрано базовий образ *python*:3.11-*alpine*, що базується на *Alpine Linux* - надзвичайно мінімалістичному дистрибутиві *Linux*. *Alpine* використовує *musl libc* замість *glibc* та *BusyBox* замість стандартних *GNU* утиліт, що дозволяє досягти значно менших розмірів образів. Для *Telegram*-бота це рішення є оптимальним, оскільки більшість його залежностей є чистими *Python*-пакетами без складних нативних розширень. *Dockerfile* сервісу *e_schedule_bot* показано на рисунку 3.1.

```
    # Використовуємо офіційний образ Python 3.11 alpine як базу для
етапу збірки
    FROM python:3.11-alpine AS builder

    # Копіюємо бінарні файли uv (uv, uvx та допоміжні виконувані
файли) із зазначеного образу uv
    COPY --from=ghcr.io/astral-sh/uv:0.6.6 /uv /uvx /bin/

    # Встановлюємо робочу директорію для етапу збірки
    WORKDIR /app

    # Копіюємо файли залежностей проекту в етап збірки
    COPY pyproject.toml uv.lock ./

    # Синхронізуємо залежності використовуючи інструмент uv з
суворими прапорами для відтворюваності
    RUN uv sync --frozen --no-install-project --no-editable --no-
cache

    # Використовуємо офіційний образ Python 3.11 alpine для
фінального етапу виконання
    FROM python:3.11-alpine

    # Встановлюємо робочу директорію для етапу виконання
    WORKDIR /app

    # Копіюємо віртуальне середовище, зібране на етапі збірки, в
етап виконання
    COPY --from=builder /app/.venv /app/.venv

    # Оновлюємо PATH, щоб включити директорію з бінарними файлами
віртуального середовища
    ENV PATH="/app/.venv/bin:$PATH"

    # Копіюємо код додатку в контейнер
    COPY main.py ./
    COPY src/ ./src/

    # Визначаємо команду за замовчуванням для запуску бота з
використанням Python
    CMD ["python", "main.py"]
```

Рисунок 3.1 – *Dockerfile* сервісу *e_schedule_bot*

Архітектура *Dockerfile* залишається консистентною з *API* сервісом - використовується той самий підхід з багатоетапною збіркою та інструментом *uv* для управління залежностями. Це забезпечує уніфікованість процесу збірки та обслуговування всієї системи. Команда запуску *CMD* [“*python*”, “*main.py*”] є

стандартним способом запуску *Python*-скрипту, який ініціалізує бота та встановлює необхідні обробники подій.

Багатоетапна збірка є фундаментальною технікою оптимізації *Docker*-образів, що дозволяє значно зменшити їх фінальний розмір без компромісів у функціональності. Цей підхід особливо ефективний для мов програмування з інтерпретованими або віртуальними середовищами виконання, таких як *Python*.

3.2 Оркестрація сервісів

Для ефективного управління та взаємодії між різними компонентами системи “єРозклад” під час локальної розробки та тестування використовується інструмент оркестрації *Docker Compose*. Він дозволяє визначити та запустити багатоконтейнерний *Docker*-додаток за допомогою одного конфігураційного файлу *docker-compose.yml*. Цей файл описує сервіси, мережі та томи, необхідні для повноцінної роботи системи, забезпечуючи централізоване управління всіма компонентами розподіленої архітектури.

Кожен сервіс визначений з власними параметрами конфігурації. Для сервісу *postgres* вказано образ 17.4, ім’я контейнера *PostgreSQL*, змінні середовища для налаштування користувача та пароля, а також список баз даних для створення. Важливою частиною є секція *volumes*, яка монтує іменований том *e_schedule_postgres_data* для збереження даних *PostgreSQL* та скрипт *init-multiple-databases.sh* для ініціалізації декількох баз даних при першому запуску. Такий підхід забезпечує автоматичне створення всіх необхідних баз даних для *API* та бота під час першого розгортання системи.

Сервіси *API* та бота збираються з локальних *Dockerfile* за допомогою секції *build*, де вказано *context* (шлях до директорії сервісу) та *dockerfile* (ім’я *Dockerfile*). Це дозволяє автоматично збирати контейнери з найновішими змінами в коді під час запуску системи. Для кожного з цих сервісів також визначені змінні середовища, необхідні для їх роботи, такі як адреси баз даних, *Redis*, *NATS*, токени *API* тощо.

Для забезпечення взаємодії між контейнерами визначена користувацька мережа типу *bridge* під назвою *e_schedule_network*. Використання власної мережі

замість стандартної мережі *Docker* забезпечує ізоляцію від інших контейнерів, що можуть працювати на тій же хост-машині, та надає повний контроль над мережевими налаштуваннями. Усі сервіси підключені до цієї мережі, що дозволяє їм звертатися один до одного за іменами сервісів, визначеними у *docker-compose.yml*.

Наприклад, *API* сервіс може звернутися до бази даних *PostgreSQL* за адресою *postgres*:5432, а *Telegram*-бот може взаємодіяти з *API* за адресою *e_schedule_api*:80. Це спрощує конфігурацію та робить систему більш гнучкою, оскільки *IP*-адреси контейнерів можуть змінюватися при кожному перезапуску, але імена сервісів залишаються сталими в межах *Docker Compose* мережі.

Змінні середовища для конфігурації сервісів виносяться у файл *.env*, розташований у кореневій директорії проєкту. У файлі *docker-compose.yml* ці змінні використовуються через синтаксис ${*VARIABLE_NAME*}. Такий підхід дозволяє легко змінювати конфігурацію без модифікації самого *docker-compose.yml* та запобігає випадковому потраплянню чутливих даних у систему контролю версій, оскільки файл *.env* зазвичай додається до *.gitignore*.

Файл *.env* містить всі критично важливі параметри конфігурації. Використання змінних середовища забезпечує безпеку системи, оскільки чутливі дані не зберігаються безпосередньо в конфігураційних файлах, що можуть потрапити в публічні репозиторії.

Для забезпечення персистентності даних, які не повинні втрачатися при перезапуску або видаленні контейнерів, використовуються *Docker*-томи. У проєкті визначено три іменованих томи: *e_schedule_postgres_data* для зберігання даних бази *PostgreSQL*, *e_schedule_redis_data* для зберігання даних кешу *Redis*. Іменовані томи керуються *Docker* і зберігаються у спеціальній директорії на хост-машині, що забезпечує їх збереження навіть після повного видалення контейнерів.

Окрім іменованих томів, використовуються прив’язані томи (*bind mounts*) для монтування директорій з хост-системи в контейнери. Скрипт ініціалізації баз даних *./init-multiple-databases.sh* монтується в контейнер *PostgreSQL* для автоматичного

створення декількох баз даних при першому запуску. Також директорії *data* кожного з прикладних сервісів монтуються з хост-системи.

Такий підхід дозволяє зберігати логи, файли токенів *OAuth*, тимчасові файли та інші дані, специфічні для додатків, на хост-машині, а також полегшує розробку, оскільки зміни в цих директоріях миттєво відображаються в контейнерах.

Для контролю стану критично важливих сервісів, таких як *PostgreSQL* та *Redis*, у *docker-compose.yml* визначені *healthcheck*. Перевірка стану *PostgreSQL* виконується командою *pg_isready -U* ${*POSTGRES_USER*}, яка перевіряє готовність сервера бази даних до прийняття з'єднань. Для *Redis* використовується команда *redis-cli ping*, яка повертає відповідь *PONG* у разі успішної роботи сервера кешування.

Політика перезапуску *restart*: *unless-stopped* застосована до всіх сервісів, що означає автоматичний перезапуск контейнерів у разі їх аварійного завершення, за винятком випадків, коли контейнер був зупинений вручну командою *docker stop*.

3.3 Впровадження системи безперервної інтеграції та доставки

Безперервна інтеграція та доставка (*Continuous Integration*/*Continuous Deployment*, *CI*/*CD*) є сучасною методологією розробки програмного забезпечення, яка передбачає автоматизацію процесів збірки, тестування та розгортання коду. Термін "безперервна інтеграція" означає практику регулярного об'єднання змін коду від різних розробників у спільний репозиторій з автоматичним запуском процесів перевірки та збірки. "Безперервна доставка" розширює цю концепцію, включаючи автоматизацію процесу розгортання програмного забезпечення у виробничому середовищі.

Основна мета *CI*/*CD* полягає у зменшенні ризиків, пов'язаних з випуском нових версій програмного забезпечення, прискоренні циклу розробки та підвищенні якості кінцевого продукту. Традиційно процес інтеграції коду та розгортання вимагав значних ручних зусиль, що призводило до помилок, затримок та непередбачуваних проблем у виробничому середовищі. *CI*/*CD* автоматизує ці

процеси, забезпечуючи швидку, надійну та повторювану доставку змін користувачам.

У сервісі “єРозклад” система *CI*/*CD* реалізована з використанням *GitHub Actions* [4] як основної платформи автоматизації. *GitHub Actions* є вбудованим сервісом *GitHub*, який дозволяє створювати *workflow* для автоматизації різноманітних задач розробки, включаючи збірку, тестування та розгортання коду. Вибір *GitHub Actions* обумовлений тісною інтеграцією з системою контролю версій *GitHub*.

Архітектура проєкту “єРозклад” базується на принципах мікросервісів, де кожен компонент системи (*API* сервіс, *Telegram*-бот, *Gmail*-коннектор, інструменти автоматизації) розміщено в окремих *GitHub* репозиторіях. Такий підхід дозволяє реалізувати незалежні *CI*/*CD pipeline* для кожного сервісу, що забезпечує можливість окремого розгортання компонентів без впливу на інші частини системи. Кожен репозиторій містить власний *workflow* файл у директорії *.github*/*workflows*/, який визначає специфічну логіку розгортання для відповідного сервісу.

Тригери *CI*/*CD pipeline* налаштовано таким чином, щоб автоматично реагувати на зміни в коді. Основним тригером є *push* до гілки *main*, що означає автоматичний запуск процесу розгортання при внесенні будь-яких змін у головну гілку розробки. Додатково реалізовано можливість ручного запуску *workflow* через тригер *workflow_dispatch*, що дозволяє адміністраторам системи ініціювати розгортання за потреби без внесення змін у код. Такий підхід забезпечує гнучкість управління процесом розгортання та можливість швидкого реагування на критичні ситуації. Конфігурацію тригерів для автоматичного запуску *CI*/*CD* показано на рисунку 3.2.

```
name: Розгортання API єРозклад
on:
push:
branches:
- main
workflow_dispatch:
```

Рисунок 3.2 – Конфігурація тригерів для автоматичного запуску *CI*/*CD*

Особливістю реалізації *CI*/*CD* у проєкті “єРозклад” є використання *self-hosted runner* замість стандартних *GitHub-hosted runner*. *Self-hosted runner* встановлено безпосередньо на сервері розгортання. Це дозволяє мати повний контроль над середовищем виконання, включаючи встановлене програмне забезпечення, мережеві налаштування та доступ до локальних ресурсів. Значно підвищується швидкість виконання операцій розгортання, оскільки не потрібно передавати артефакти між різними системами.

Процес розгортання складається з декількох послідовних етапів, кожен з яких виконує специфічну функцію у загальному процесі доставки коду. Перший етап включає оновлення локальної копії коду через виконання команди *git pull origin main* у відповідній директорії проєкту. Це забезпечує синхронізацію локального репозиторію з віддаленою версією та отримання всіх останніх змін. Етап оновлення коду *API* з віддаленого репозиторію показано на рисунку 3.3.

```
    - name: Оновлення коду API
      run: |
        cd /home/deploy/e_schedule/e_schedule_api
        git pull origin main
```

Рисунок 3.3 – Етап оновлення коду *API* з віддаленого репозиторію

Наступний етап передбачає перезапуск відповідного сервісу через *Docker Compose* з параметром *–build*, що забезпечує збірку нового *Docker* образу з урахуванням внесених змін. Команди перезапуску *API* сервісу через *Docker Compose* показано на рисунку 3.4. Використання параметра *-d* (*detached mode*) дозволяє запустити контейнер у фоновому режимі, не блокуючи виконання *workflow*. *Docker Compose* автоматично зупиняє попередню версію контейнера та запускає нову, забезпечуючи безперервність роботи сервісу.

```
    - name: Перезапуск служби API
      working-directory: /home/deploy/e_schedule
      run: |
        docker compose up -d --build e_schedule_api
```

Рисунок 3.4 – Команди перезапуску *API* сервісу через *Docker Compose*

Завершальний етап *pipeline* включає очищення *Docker*-ресурсів для підтримання чистоти системи та запобігання переповненню дискового простору. Виконуються команди для видалення невикористовуваних образів, контейнерів та мереж з фільтром за часом, що дозволяє зберегти нещодавно створені ресурси та видалити лише застарілі елементи. Автоматичне очищення невикористовуваних *Docker*-ресурсів показано на рисунку 3.5.

```
- name: Очищення Docker-ресурсів
run: |
docker image prune -af --filter "until=24h"
docker container prune -f
docker network prune -f
```

Рисунок 3.5 – Автоматичне очищення невикористовуваних *Docker*-ресурсів

ВИСНОВКИ

Кваліфікаційна робота присвячена розробці комплексної автоматизованої системи “єРозклад”, призначеної для ефективного управління навчальним розкладом, оперативного інформування здобувачів освіти та викладачів про зміни, а також автоматизації пов’язаних процесів у навчальних закладах.

Система включає *API* сервіс, *Telegram*-бот для взаємодії з користувачами та сервіс інтеграції з *Gmail* для автоматичної обробки повідомлень про заміни. У ході виконання цієї роботи було детально проаналізовано існуючі проблеми управління навчальним розкладом, такі як ручне введення даних, затримки з оновленнями, фрагментованість інформації та відсутність ефективних каналів сповіщення.

У даній роботі було запропоновано та реалізовано мікросервісну архітектуру, що забезпечує гнучкість, масштабованість та незалежність розробки окремих компонентів системи. Використання мови програмування *Python* з фреймворками *FastAPI* для розробки *API* сервісу та *Aiogram* для створення *Telegram*-бота, у поєднанні з системою управління базами даних *PostgreSQL*, системою кешування *Redis* та брокером повідомлень *NATS*, дозволило створити надійне та високопродуктивне рішення.

Розроблено функціонал для перегляду розкладу занять, замін, практик та іспитів, реалізовано систему персоналізованих автоматичних сповіщень, адміністративну панель для управління контентом та розсилками, а також механізм автоматичної обробки електронних листів із замінами. Впроваджено технології контейнеризації та оркестрації, а також систему безперервної інтеграції та доставки (*CI*/*CD*) за допомогою *GitHub Actions*. Таким чином, розроблена система “єРозклад” є значним кроком до модернізації процесів управління навчальним розкладом, підвищення їх ефективності, оперативності інформування та, як наслідок, покращення загальної якості освітнього процесу. Впровадження подібних систем дозволяє оптимізувати використання ресурсів навчального закладу та створити більш комфортні умови для всіх учасників освітнього процесу.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. *FastAPI* – офіційна документація. [Електронний ресурс] – Режим доступу: *https://fastapi.tiangolo.com/* (дата звернення: 23.05.2025)

2. *Aiogram* – офіційна документація. [Електронний ресурс] – Режим доступу: *https://docs.aiogram.dev/* (дата звернення: 19.05.2025)

3. *Docker* – офіційна документація. [Електронний ресурс] – Режим доступу: *https://docs.docker.com/* (дата звернення: 30.05.2025)

4. *GitHub Actions* – офіційна документація. [Електронний ресурс] – Режим доступу: *https://docs.github.com/en/actions* (дата звернення: 25.05.2025)

5. *NATS* – офіційна документація. [Електронний ресурс] – Режим доступу: *https://docs.nats.io/* (дата звернення: 11.05.2025)

6. *Redis* – офіційна документація. [Електронний ресурс] – Режим доступу: *https://redis.io/docs/latest/* (дата звернення: 25.05.2025)

7. *AWS* – офіційна документація. [Електронний ресурс] – Режим доступу: *https://docs.aws.amazon.com/* (дата звернення: 16.05.2025)

8. *Pydantic* – офіційна документація. [Електронний ресурс] – Режим доступу: *https://docs.pydantic.dev/latest/* (дата звернення: 20.05.2025)

9. *PostgreSQL* – офіційна документація. [Електронний ресурс] – Режим доступу: *https://www.postgresql.org/docs/* (дата звернення: 20.05.2025)

10. *System Design Primer – GitHub* репозиторій. [Електронний ресурс] – Режим доступу: *https://github.com/donnemartin/system-design-primer* (дата звернення: 21.05.2025)

11. *Microservices* – вікіпедія. [Електронний ресурс] – Режим доступу: *https://uk.wikipedia.org/wiki/Microservices* (дата звернення: 05.05.2025)

12. *ACID* – вікіпедія. [Електронний ресурс] – Режим доступу: *https://en.wikipedia.org/wiki/ACID* (дата звернення: 12.05.2025)

13. *Microservices Pattern* – архітектурні патерни мікросервісів. [Електронний ресурс] – Режим доступу: *https://microservices.io/patterns/microservices.html* (дата звернення: 22.05.2025)

14. *Microsoft Azure* – керівництво з архітектури мікросервісів. [Електронний ресурс] – Режим доступу: *https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices* (дата звернення: 19.05.2025)

15. *Google Cloud* – введення в мікросервіси. [Електронний ресурс] – Режим доступу: *https://cloud.google.com/architecture/microservices-architecture-introduction* (дата звернення: 24.05.2025)

16. *Telegram Bot API* – офіційна документація. [Електронний ресурс] – Режим доступу: *https://core.telegram.org/bots/api* (дата звернення: 21.05.2025)

17. *SQLAlchemy* – офіційна документація. [Електронний ресурс] – Режим доступу: *https://docs.sqlalchemy.org/* (дата звернення: 23.05.2025)

18. *Python 3.11* – офіційна документація. [Електронний ресурс] – Режим доступу: *https://docs.python.org/3.11/* (дата звернення: 22.05.2025)

19. *uv* – менеджер пакетів *Python*. [Електронний ресурс] – Режим доступу: *https://docs.astral.sh/uv/* (дата звернення: 24.05.2025)

ДОДАТОК А

Модуль кешування *API* даних

```
import json
import logging
from datetime import datetime
from zoneinfo import ZoneInfo

from src.api.api_client import ApiClient
from src.config.redis import redis_client

logger = logging.getLogger(__name__)

class APICache:
    """
    Клас для отримання даних з API через ApiClient та їх кешування в
Redis.
    Використовуються наступні endpoint'и:
      - /api/week-intervals
      - /api/groups
      - /api/teachers

    TTL кешу за замовчуванням – 7200 секунд.
    """

    _ttl = 7200  # час життя кешу в секундах

    @staticmethod
    async def _fetch_and_cache(cache_key: str, fetch_func, ttl: int =
_ttl):
        """
         Загальний метод для отримання даних з Redis (якщо вони вже
закешовані) або виклику функції fetch_func,
         яка має виконати запит до API (з використанням ApiClient).
Отримані дані серіалізуються в JSON
        та зберігаються в Redis із заданим часом життя (ttl).

        :param cache_key: Ключ для зберігання даних в Redis.
         :param fetch_func: Асинхронна функція, яка повертає дані з
API.
        :param ttl: Час життя кешу в секундах (за замовчуванням 7200).
        :return: Десеріалізовані дані (зазвичай словник).
        """
        try:
            cached = await redis_client.get(cache_key)
            if cached:
                try:
                    data = json.loads(cached)
                        # logger.info(f"Дані отримано з кешу (ключ:
{cache_key})")
                    return data
                except json.JSONDecodeError:
```

```
                    logger.error(f"Помилка JSON-декодування для ключа
{cache_key}")
            # Якщо даних немає в кеші — викликаємо функцію для отримання
даних з API
            data = await fetch_func()
            await redis_client.set(cache_key, json.dumps(data), ex=ttl)
             # logger.info(f"Дані отримано та збережено в кеші (ключ:
{cache_key})")
            return data
        except Exception as e:
                logger.error(f"Помилка при роботі з кешем для ключа
{cache_key}: {e}")
            raise

    @staticmethod
    async def _fetch_teachers_data():
        """
        Отримує дані про викладачів через ApiClient.
        """
        async with ApiClient() as client:
            return await client.get_teachers()

    @staticmethod
    async def _fetch_groups_data():
        """
        Отримує дані про відділення та групи через ApiClient.
        """
        async with ApiClient() as client:
            return await client.get_groups()

    @staticmethod
    async def _fetch_week_intervals_data():
        """
        Отримує дані про тижневі інтервали через ApiClient.
        """
        async with ApiClient() as client:
            return await client.get_week_intervals()

    @staticmethod
    async def get_teacher_initials() -> list:
        """
        Повертає список ініціалів усіх викладачів.

        :return: Список рядків з ініціалами.
        """
        cache_key = "api_cache:teachers"
               data   =   await   APICache._fetch_and_cache(cache_key,
APICache._fetch_teachers_data)
        if "teachers" in data:
                initials = [teacher.get("initials") for teacher in
data.get("teachers", [])]
            return initials
        else:
```

```
            logger.error("Ключ 'teachers' не знайдено в даних")
            return []

    @staticmethod
    async def get_department_names() -> list:
        """
        Повертає список назв усіх відділень.

        :return: Список назв відділень.
        """
        cache_key = "api_cache:groups"
               data   =   await   APICache._fetch_and_cache(cache_key,
APICache._fetch_groups_data)
        if "departments" in data:
             department_names = [dept.get("departmentName") for dept
in data.get("departments", [])]
            return department_names
        else:
            logger.error("Ключ 'departments' не знайдено в даних")
            return []

    @staticmethod
    async def get_groups_for_department(department_name: str) -> list:
        """
        Повертає список груп для вказаного відділення.

        :param department_name: Назва відділення.
         :return: Список груп для даного відділення (якщо відділення
не знайдено, повертається пустий список).
        """
        cache_key = "api_cache:groups"
               data   =   await   APICache._fetch_and_cache(cache_key,
APICache._fetch_groups_data)
        if "departments" in data:
            for dept in data.get("departments", []):
                        if  dept.get("departmentName",  "").lower()  ==
department_name.lower():
                    return dept.get("groups", [])
           logger.warning(f"Відділення '{department_name}' не знайдено
в даних")
            return []
        else:
            logger.error("Ключ 'departments' не знайдено в даних")
            return []

    @staticmethod
    async def get_all_groups() -> list:
        """
        Повертає список усіх груп незалежно від відділення.

        :return: Список груп.
        """
        cache_key = "api_cache:all_groups"
```

```
            data = await APICache._fetch_and_cache(cache_key,
APICache._fetch_groups_data)
        if "departments" in data:
            all_groups = []
            for dept in data.get("departments", []):
                all_groups.extend(dept.get("groups", []))
            return all_groups
        else:
            logger.error("Ключ 'departments' не знайдено в даних")
            return []

    @staticmethod
    async def get_current_and_next_week_type() -> tuple[str | None,
str | None]:
        """
         Повертає тип поточного тижня та наступного тижня на основі
даних з endpoint /api/week-intervals.
         Дані мають містити список інтервалів, де кожен інтервал має
поля:
            - startDate (наприклад, "2025-02-01")
            - endDate (наприклад, "2025-02-07")
            - weekType (наприклад, "Чисельник" або "Знаменник")

        Алгоритм:
          1. Отримуємо та парсимо інтервали, сортуємо їх за startDate.
          2. Порівнюємо поточну дату з діапазонами інтервалів:
            - Якщо сьогодні входить в інтервал, то це поточний тиждень,
                а наступним вважається інтервал з наступним індексом
(якщо він є).
            - Якщо сьогодні раніше першого інтервалу, то беремо перший
інтервал як поточний і другий як наступний.
            - Якщо сьогодні пізніше останнього інтервалу — повертається
останній interval як поточний.

          :return: Кортеж (current_week_type, next_week_type). Якщо
наступного тижня немає — next_week_type буде None.
        """
        cache_key = "api_cache:week_intervals"
            data = await APICache._fetch_and_cache(cache_key,
APICache._fetch_week_intervals_data)
        if "weekIntervals" not in data:
            logger.error("Ключ 'weekIntervals' не знайдено в даних")
            return None, None

        intervals = data.get("weekIntervals", [])
        if not intervals:
            logger.error("Список 'weekIntervals' порожній")
            return None, None

        # Парсимо інтервали та сортуємо за датою початку
        parsed_intervals = []
        for interval in intervals:
            try:
```

```
                                                    start_date      =
datetime.strptime(interval.get("startDate"), "%Y-%m-%d").date()
                end_date = datetime.strptime(interval.get("endDate"),
"%Y-%m-%d").date()
                parsed_intervals.append(
                    {
                        "id": interval.get("id"),
                        "start_date": start_date,
                        "end_date": end_date,
                        "weekType": interval.get("weekType"),
                    }
                )
            except Exception as e:
                  logger.error(f"Помилка парсингу дат для інтервалу
{interval}: {e}")
        parsed_intervals.sort(key=lambda x: x["start_date"])
        today = datetime.now(ZoneInfo("Europe/Kiev")).date()

        current_week_type = None
        next_week_type = None
        found = False

        for i, interval in enumerate(parsed_intervals):
           if interval["start_date"] <= today <= interval["end_date"]:
                current_week_type = interval["weekType"]
                found = True
                if i + 1 < len(parsed_intervals):
                  next_week_type = parsed_intervals[i + 1]["weekType"]
                break

        if not found:
            # Якщо поточна дата раніше першого інтервалу:
            if today < parsed_intervals[0]["start_date"]:
                current_week_type = parsed_intervals[0]["weekType"]
                if len(parsed_intervals) > 1:
                    next_week_type = parsed_intervals[1]["weekType"]
            # Якщо дата пізніше останнього інтервалу:
            elif today > parsed_intervals[-1]["end_date"]:
                current_week_type = parsed_intervals[-1]["weekType"]
                next_week_type = None
            else:
                # Якщо сьогодні не входить ні в один інтервал,
                 # шукаємо найближчий інтервал, що починається після
сьогодні
                for i, interval in enumerate(parsed_intervals):
                    if interval["start_date"] > today:
                        current_week_type = interval["weekType"]
                        if i + 1 < len(parsed_intervals):
                                next_week_type = parsed_intervals[i +
1]["weekType"]
                        break

        return current_week_type, next_week_type
```

```
    @staticmethod
    async def get_teacher_initials_by_id(teacher_id: int) -> str:
        """
        Повертає ініціали викладача за його ідентифікатором.

        :param teacher_id: Ідентифікатор викладача.
        :return: Рядок з ініціалами (наприклад, "Гринченко Я.П."), або
пустий рядок,
                 якщо викладача з таким id не знайдено.
        """
        cache_key = "api_cache:teachers"
              data   =   await   APICache._fetch_and_cache(cache_key,
APICache._fetch_teachers_data)
        if "teachers" in data:
            for teacher in data["teachers"]:
                if teacher.get("id") == teacher_id:
                    return teacher.get("initials", "")
            logger.warning(f"Викладач з id {teacher_id} не знайдений")
            return ""
        else:
            logger.error("Ключ 'teachers' не знайдено в даних")
            return ""

    @staticmethod
    async def get_teacher_id_by_initials(initials: str) -> int | None:
        """
        Повертає ідентифікатор викладача за його ініціалами.

         :param initials: Ініціали викладача (наприклад, "Гринченко
Я.П.").
        :return: Ідентифікатор викладача, або None, якщо викладача з
такими ініціалами не знайдено.
        """
        cache_key = "api_cache:teachers"
              data   =   await   APICache._fetch_and_cache(cache_key,
APICache._fetch_teachers_data)
        if "teachers" in data:
            for teacher in data["teachers"]:
                if teacher.get("initials") == initials:
                    return teacher.get("id")
            logger.warning(f"Викладач з ініціалами {initials} не
знайдений")
            return None
        else:
            logger.error("Ключ 'teachers' не знайдено в даних")
            return None

class ApiClient:
    """
    Асинхронний клієнт для взаємодії із зовнішнім API.
    """
```

```
    def __init__(self, base_url: str = API_URL, timeout: int = 10):
        """
        :param base_url: Базовий URL API (за замовчуванням береться з
налаштувань)
        :param timeout: Таймаут запитів у секундах
        """
        self.base_url = base_url
        self.timeout = timeout
            self.client  =  httpx.AsyncClient(base_url=self.base_url,
timeout=self.timeout)

    async def __aenter__(self):
        return self

    async def __aexit__(self, exc_type, exc, tb):
        await self.close()

    async def close(self):
        """Коректно закриває клієнт (звільняє ресурси)."""
        await self.client.aclose()

    async def _get(self, endpoint: str, params: dict[str, Any] | None
= None) -> dict[str, Any]:
        """
        Внутрішній метод для відправлення GET-запитів.

       :param endpoint: Шлях endpoint (наприклад, '/api/schedule/day')
        :param params: Параметри запиту
        :return: Розпарсена JSON-відповідь у вигляді словника
        :raises Exception: При помилці запиту
        """
        try:
            response = await self.client.get(endpoint, params=params)
            response.raise_for_status()
            return response.json()
        except httpx.HTTPError as e:
           logger.error(f"Помилка при запиті до {endpoint}: {str(e)}")
               raise  Exception(f"Помилка  при  запиті  до  {endpoint}:
{str(e)}")

    async def get_schedule(
        self,
        schedule_type: str,
        identifier: str,
        weekType: str,
        dayOfWeek: str,
    ) -> dict[str, Any]:
        """
        Отримує розклад для групи або викладача.

        :param schedule_type: 'group' або 'teacher'
        :param identifier: назва групи або ініціали викладача
        :param weekType: 'Чисельник' або 'Знаменник'
```

```
:param dayOfWeek: день тижня, наприклад, 'Понеділок'
"""
params = {
    "type": schedule_type,
    "identifier": identifier,
    "weekType": weekType,
}
if dayOfWeek:
    params["dayOfWeek"] = dayOfWeek
return await self._get("/api/schedule/day", params=params)
```

КРИВОРІЗЬКИЙ ФАХОВИЙ КОЛЕДЖ
ДЕРЖАВНОГО НЕКОМЕРЦІЙНОГО ПІДПРИЄМСТВА
«ДЕРЖАВНИЙ УНІВЕРСИТЕТ «КИЇВСЬКИЙ АВІАЦІЙНИЙ ІНСТИТУТ»

ВІДГУК
керівника кваліфікаційної роботи

випускника спеціальності: 123 «Комп'ютерна інженерія»

відділення: комп'ютерної та програмної інженерії

циклова комісія: комп'ютерних систем та мереж

Олександр МУСІЄНКО
(ім'я, прізвище)

1. *Кваліфікаційна робота присвячена створенню автоматизованої системи "єРозклад" для ефективного управління навчальним розкладом та оперативного інформування всіх учасників освітнього процесу. Розроблена система охоплює всі ключові аспекти сучасного IT-рішення: мікросервісна архітектура, використання актуальних фреймворків і технологій (FastAPI, Aiogram, PostgreSQL, Redis, NATS), інтеграція з поштовим сервісом Gmail, а також засоби автоматизації CI/CD. Особливої уваги заслуговує практична реалізація Telegram-бота та системи сповіщень, що забезпечує зручну і швидку комунікацію з користувачами.*
2. *В межах виконання роботи здобувач освіти продемонстрував високий рівень технічної підготовки, вміння аналізувати існуючі проблеми галузі та знаходити ефективні програмні рішення.*
3. *Здобувач освіти грамотно структурував роботу, обґрунтував вибір інструментів та рішень, продемонстрував здатність до самостійної розробки програмного продукту. Усі поставлені завдання виконані у повному обсязі.*
4. *Рівень виконаної кваліфікаційної роботи заслуговує оцінку «відмінно», відповідає набутих випускником знань, умінь та навичок, вимогам освітньої характеристики фахівця і можливість присвоєння йому кваліфікації фахівця освітньо-кваліфікаційного ступеню «Фаховий молодший бакалавр» спеціальності 123 «Комп'ютерна інженерія».*

Керівник кваліфікаційної роботи

«10» 06 2025 р. _____ Світлана ТЕРЬОШИНА
(підпис) (ім'я, прізвище)

КРИВОРІЗЬКИЙ ФАХОВИЙ КОЛЕДЖ
ДЕРЖАВНОГО НЕКОМЕРЦІЙНОГО ПІДПРИЄМСТВА
«ДЕРЖАВНИЙ УНІВЕРСИТЕТ «КИЇВСЬКИЙ АВІАЦІЙНИЙ ІНСТИТУТ»

РЕЦЕНЗІЯ
на кваліфікаційну роботу

випускника спеціальності: 123 «Комп'ютерна інженерія»

відділення: комп'ютерної та програмної інженерії

циклова комісія: комп'ютерних систем та мереж

Олександр МУСІЄНКО
(ім'я, прізвище)

1. *Кваліфікаційна робота присвячена актуальній темі — автоматизації управління навчальним розкладом у закладах освіти. У роботі глибоко проаналізовано проблеми ручного управління розкладом, фрагментації інформації та затримок із її оновленням, на основі чого запропоновано комплексне технічне рішення — систему "єРозклад".*
2. *Особливої уваги заслуговує впровадження CI/CD-практик, використання мікросервісної архітектури та застосування засобів інтеграції з месенджерами та поштовими сервісами. Усі компоненти системи працюють узгоджено та ефективно вирішують поставлені завдання.*
3. *Робота має як теоретичну, так і значну практичну цінність, демонструє реальні шляхи підвищення ефективності навчального процесу за рахунок цифровізації. Оформлення роботи відповідає вимогам, а структура є логічною та послідовною.*
4. *Кваліфікаційна робота заслуговує оцінку «відмінно».*

Рецензент
викладач
(науковий ступінь, посада)

«____»_____ 2025 р. _____ Тетяна НОВІК
(підпис) (ім'я, прізвище)

З рецензією ознайомлений _____ Олександр МУСІЄНКО
(підпис) (ім'я, прізвище)