БАКАЛАВРСЬКА РОБОТА

БР.КІ-05.00.00.000 ПЗ

Група КІ-22-1

Гулик Роман

2026

Івано-Франківський національний технічний університет нафти і газу
Факультет інформаційних технологій
Кафедра комп'ютерних систем і мереж

Гулик Роман Олегович

УДК 004.02

БАКАЛАВРСЬКА РОБОТА

Розробка web-додатку підтримки функціонування месенджер-бота сервісу розкладу занять ІФНТУНГ на базі Redis

Комп'ютерна інженерія
(назва освітньої програми)

123 - Комп'ютерна інженерія
(шифр і назва спеціальності)

Робота містить результати власних досліджень, використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело:

Здобувач освітнього ступеня _____ Гулик Р. О.
(підпис, прізвище та ініціали здобувача)

Науковий керівник _____ Слабінога М. О., к. т. н., доцент
(підпис, прізвище, ім'я, по батькові, науковий ступінь, вчене звання керівника)

Допущено до захисту

Завідувач кафедри КСМ

д.т.н., професор _____ Мельничук С. І.
(посада) (підпис) (дата) (прізвище та ініціали)

м. Івано-Франківськ – 2026

Івано-Франківський національний технічний університет нафти і газу

Факультет інформаційних технологій

Кафедра комп'ютерних систем і мереж

Освітній рівень бакалавр

Спеціальність 123 Комп’ютерна інженерія

ЗАТВЕРДЖУЮ

Завідувач кафедри КСМ

Мельничук С. І.

" 21 " квітня 2026 року

З А В Д А Н Н Я

НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТОВІ

Гулику Роману Олеговичу

(прізвище, ім’я, по батькові)

1. Тема роботи Розробка web-додатку підтримки функціонування месенджер-бота сервісу розкладу занять ІФНТУНГ на базі Redis

керівник роботи Слабінога Мар’ян Остапович, к. т. н., доцент

затверджені наказом закладу вищої освіти від 21.04.2026 № 180/7

2. Термін подання студентом роботи 11 червня 2026 року

3. Вихідні дані до роботи Методичні вказівки, технічна література

4. Зміст пояснювальної записки: 1 Проблема доступу до даних зі сторонніх сервісів, аналіз рішень кешування зі сторонніх сервісів, постановка задачі. 2 Вимоги до програмного забезпечення, аналіз структури бази даних, сценарії роботи користувача. 3 Реалізація програмного забезпечення, розгортання та конфігурація середовища розробки проекту, демонстрація роботи ПЗ на прикладі месенджера Telegram

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

6. Консультанти розділів роботи

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

7. Дата видачі завдання – 02 лютого 2026 року

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Термін виконання етапів роботи	Примітка
1	Підготовка та вибір літератури для написання бакалаврської роботи	Лютий 2026р	виконав
2	Опрацювання предметної області та аналіз аналогів сервісів з доступом до сторонніх сервісів	Березень 2026р	виконав
3	Розробка архітектури проекту, структури бази даних, діаграм	Квітень 2026р	виконав
4	Розробка коду для доступу до сторонніх сервісів і зберігання	Травень 2026р	виконав
5	Оформлення пояснювальної записки	Червень 2026р	виконав

Студент _____ Гулик Р. О.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Слабінога М. О.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Основною метою роботи є розробка веб-додатку та серверної частини месенджер-бота для автоматизації доступу до розкладу занять ІФНТУНГ. Для забезпечення високої продуктивності системи та швидкого опрацювання запитів користувачів впроваджено систему керування базами даних Redis, що виконує роль високошвидкісного сховища для кешування актуальних даних розкладу.

У ході дослідження було спроектовано структуру взаємодії між веб-сервером, месенджер-ботом та зовнішніми джерелами даних. Реалізовано алгоритми синхронізації інформації, механізми обробки запитів від студентів та викладачів, а також інтерфейс адміністратора для моніторингу працездатності сервісу. Результатом роботи є стабільний веб-додаток, який суттєво знижує навантаження на інформаційну систему університету та забезпечує миттєве отримання розкладу через мобільні месенджери.

Ключові слова: redis, messenger bot, розклад занять, web-додаток, кешування даних, інформаційна система.

ANNOTATION

The main objective of this work is to develop a web application and server-side logic to ensure the stable operation of a messenger bot that provides access to the class schedule for students and faculty of IFNTUOG. To achieve high data access speed and optimize the load on the primary information system, the Redis data structure store was utilized.

The tasks include designing the architecture for the bot's interaction with schedule data sources, implementing request caching mechanisms, and developing an administrative interface for monitoring the service status. The result of the work is a functional web application that allows users to receive up-to-date schedules in a convenient messenger format with minimal latency. The developed system meets modern requirements for high-performance services and can be expanded with additional features to improve user engagement.

Keywords**:** redis, messenger bot, class schedule, web application, data caching, information system.

ЗМІСТ

ВСТУП

Актуальність обраної теми. У сучасних умовах цифровізації освіти швидкий доступ до розкладу занять є критично важливим для забезпечення ефективного навчального процесу. Традиційні методи перегляду розкладу через вебсайти часто є незручними для мобільних користувачів або страждають від надмірного навантаження на сервери під час пікових запитів. Використання месенджер-ботів у поєднанні з технологіями швидкого кешування даних дозволяє забезпечити миттєвий відгук системи та зручність користування. Тому розробка веб-додатку для підтримки функціонування месенджер-бота на базі Redis є важливою прикладною задачею для покращення цифрової інфраструктури університету.

Об’єкт та предмет дослідження. Об’єктом дослідження є процес надання інформаційних послуг щодо розкладу занять студентам та викладачам університету. Предметом дослідження є методи та засоби оптимізації доставки даних через месенджери з використанням нереляційних сховищ даних.

Мета та завдання роботи. Метою роботи є розробка веб-додатка та серверної логіки месенджер-бота, що забезпечує стабільний та швидкий доступ до розкладу занять ІФНТУНГ. Для досягнення мети поставлено такі завдання:

- спроектувати архітектуру взаємодії бота із зовнішніми джерелами даних розкладу;
- впровадити базу даних Redis для кешування результатів запитів та зниження навантаження на основні сервери;
- реалізувати алгоритми обробки текстових та командних запитів користувачів у месенджері;
- розробити веб-інтерфейс для моніторингу та управління параметрами роботи сервісу;
- провести тестування працездатності системи при різних сценаріях навантаження.

Методи дослідження. Методи дослідження включають аналіз архітектурних підходів до побудови чат-ботів та систем реального часу. Важливим етапом є вивчення документації Redis щодо структур даних "ключ-значення" та методів їх ефективного використання для кешування. Також застосовано методи об’єктно-орієнтованого програмування для створення модульної структури додатка та проектування API-запитів для взаємодії з мессенджерами.

Практичне значення отриманих результатів. Практичне значення полягає у створенні діючого сервісу, який автоматизує процес отримання розкладу занять та забезпечує високу швидкість обробки запитів навіть при слабкому інтернет-з'єднанні у користувачів. Результати роботи можуть бути інтегровані в єдине інформаційне середовище університету для покращення взаємодії зі студентами.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧ

1.1 Проблема доступу до даних зі сторонніх сервісів

У сучасних інформаційних системах, зокрема в контексті розробки месенджер-ботів для освітніх установ, важливим аспектом є організація ефективного доступу до даних, що розміщені у сторонніх сервісах. У випадку сервісу розкладу занять ІФНТУНГ, джерелом даних виступає веб-ресурс університету, який не завжди надає спеціалізований програмний інтерфейс (API), що змушує застосовувати механізми парсингу HTML-сторінок. Такий підхід, попри свою універсальність, створює низку технічних викликів, пов’язаних із продуктивністю, надійністю та масштабованістю системи.

Однією з ключових проблем при безпосередньому доступі до сторонніх сервісів є затримка (latency), яка виникає під час виконання кожного HTTP-запиту. У випадку месенджер-бота, який обслуговує запити студентів, кожен користувацький запит може ініціювати окремий запит до сервера університету. Така модель взаємодії є неефективною з кількох причин.

По-перше, сторонній сервер може мати обмежену пропускну здатність і не бути оптимізованим для обробки великої кількості автоматизованих запитів. Це призводить до збільшення часу відповіді (response time), що негативно впливає на користувацький досвід. У месенджер-середовищі, де очікується майже миттєва відповідь, навіть затримка у кілька секунд може вважатися критичною.

По-друге, існує ризик блокування IP-адреси клієнта. Багато веб-серверів застосовують механізми захисту від надмірної кількості запитів (наприклад, anti-DDoS або rate limiting), що можуть ідентифікувати активність бота як підозрілу. У результаті сервер може тимчасово або повністю заблокувати доступ, що призведе до недоступності сервісу для всіх користувачів [1].

По-третє, нестабільність мережевого з’єднання також впливає на надійність системи. Запити до стороннього сервера можуть завершуватися помилками через тайм-аути, втрату пакетів або технічні роботи на стороні

університету. У такому випадку бот не зможе надати відповідь користувачу, що знижує довіру до системи.

Таким чином, пряма залежність від стороннього сервісу при кожному запиті створює вузьке місце в архітектурі системи та обмежує її масштабованість.

Ще однією важливою проблемою є обмеження, які накладаються сторонніми сервісами, зокрема обмеження частоти запитів (rate limiting). Навіть якщо офіційний API відсутній, веб-сервери зазвичай мають механізми контролю навантаження.

Розглянемо типовий сценарій: перед початком першої пари (наприклад, о 8:30 ранку) значна кількість студентів одночасно звертається до месенджер-бота для перевірки розкладу. Якщо припустити, що 1000 студентів надсилають запит протягом короткого проміжку часу, система, яка не використовує кешування, згенерує 1000 запитів до сервера університету.

У такій ситуації можливі наступні наслідки:

- перевищення допустимого ліміту запитів, що призведе до отримання помилок (наприклад, HTTP 429 Too Many Requests);
- значне збільшення часу обробки запитів через перевантаження сервера;
- повна відмова сервера у відповідь на запити;
- блокування IP-адреси сервісу, що обслуговує бот.

Особливо критичною ця проблема стає під час сесії, коли студенти часто перевіряють зміни в розкладі іспитів, консультацій та перескладань. Пікові навантаження у такі періоди можуть значно перевищувати середньодобові показники, що робить систему без кешування практично непридатною для використання.

Незважаючи на зазначені проблеми, доступ до сторонніх сервісів є необхідним, оскільки саме вони є джерелом актуальної інформації. У випадку ІФНТУНГ, веб-сайт розкладу виступає єдиним джерелом даних про заняття, аудиторії та викладачів.

За відсутності офіційного API, отримання даних здійснюється шляхом парсингу веб-сторінок. Парсинг передбачає:

- виконання HTTP-запиту до веб-сторінки;
- отримання HTML-респонсу;
- аналіз структури документа (DOM);
- витяг необхідних даних (розклад, групи, час занять);
- перетворення даних у структурований формат (наприклад, JSON) [2].

Такий підхід є гнучким, але водночас чутливим до змін у структурі сайту. Будь-яке оновлення HTML-розмітки може призвести до некоректної роботи парсера. Крім того, парсинг є ресурсомістким процесом і збільшує загальний час обробки запиту.

Для вирішення описаних проблем доцільно застосовувати механізм кешування. Кешування передбачає збереження результатів попередніх запитів у швидкодоступному сховищі з метою повторного використання без необхідності звернення до зовнішнього джерела.

У контексті розкладу занять важливою є властивість "умовної статичності" даних. Розклад:

- змінюється відносно рідко (зазвичай кілька разів на день або навіть рідше);
- водночас запитується дуже часто (особливо у пікові години).

Це робить його ідеальним кандидатом для кешування.

Робота системи з кешем може бути описана наступним чином:

1. При надходженні запиту від користувача система спочатку перевіряє наявність даних у кеші.

2. Якщо дані знайдено (cache hit), вони одразу повертаються користувачу з мінімальною затримкою.

3. Якщо дані відсутні (cache miss), система виконує запит до стороннього сервісу, обробляє респонс, зберігає результат у кеші та повертає його користувачу.

Такий підхід дозволяє значно зменшити кількість запитів до стороннього сервера, знизити latency та підвищити пропускну здатність системи.

Для реалізації механізму кешування доцільно використовувати спеціалізовані сховища даних типу In-memory, одним із найпоширеніших представників яких є Redis. Redis є NoSQL-системою керування базами даних, яка зберігає дані в оперативній пам’яті, що забезпечує надзвичайно швидкий доступ до них.

Основні переваги Redis у контексті даного проєкту:

1. Мінімальний час відгуку**.** Операції читання та запису в Redis виконуються за час менше ніж 1 мс, що значно швидше порівняно з традиційними реляційними базами даних або запитами до зовнішніх веб-сервісів.

2. Підвищення пропускної здатності**.** Завдяки використанню кешу більшість запитів обробляється локально, без звернення до стороннього сервера, що дозволяє обслуговувати велику кількість користувачів одночасно.

3. Підтримка різних структур даних. Redis підтримує такі типи даних, як:

- String - для збереження JSON-респонсів розкладу;
- Hash - для збереження профілів користувачів (наприклад, обрана група);
- List та Set - для організації допоміжних структур (черги оновлення, списки активних користувачів).

4. Простота інтеграції**.** Redis легко інтегрується з популярними мовами програмування та бібліотеками (наприклад, redis-py), що спрощує реалізацію.

5. Підтримка механізмів TTL. Вбудована можливість встановлення часу життя ключів дозволяє автоматично видаляти застарілі дані [3].

Одним із ключових аспектів кешування є визначення часу, протягом якого дані вважаються актуальними. Для цього використовується механізм TTL (Time-to-Live).

TTL визначає, скільки часу запис у кеші залишається дійсним. Після закінчення цього часу запис автоматично видаляється або вважається застарілим.

У контексті розкладу занять можливі такі підходи:

- встановлення TTL на рівні 5–30 хвилин для звичайного періоду;

– зменшення TTL у періоди підвищеної ймовірності змін (наприклад, під час сесії);

– примусове оновлення кешу за розкладом (cron-завдання).

Баланс між актуальністю даних і продуктивністю системи є критичним. Занадто великий TTL може призвести до відображення застарілої інформації, тоді як занадто малий - зменшить ефективність кешування.

Оскільки розроблюваний веб-додаток орієнтований на підтримку функціонування месенджер-бота, важливою складовою є моніторинг роботи кешу. Зокрема, система повинна надавати інформацію про:

– кількість кеш-попадань (cache hit);

– кількість кеш-промахів (cache miss);

– співвідношення запитів до Redis і до стороннього сервера;

– середній час відповіді.

Ці метрики дозволяють оцінити ефективність кешування, виявляти вузькі місця та оптимізувати параметри системи (наприклад, TTL).

Отже, прямий доступ до сторонніх сервісів для отримання даних розкладу є неефективним через високу затримку, обмеження на кількість запитів та нестабільність з’єднання. Особливо це проявляється в умовах пікових навантажень, характерних для освітнього середовища.

Використання механізмів кешування, зокрема на базі Redis, дозволяє значно підвищити продуктивність системи, зменшити навантаження на сторонній сервер та забезпечити швидкий доступ до даних. Завдяки властивості умовної статичності розкладу, кешування є ефективним і доцільним рішенням.

Таким чином, впровадження Redis як In-memory кешу є обґрунтованим архітектурним вибором для розробки масштабованого та надійного сервісу підтримки месенджер-бота розкладу занять ІФНТУНГ.

1.2 Аналіз рішень кешування зі сторонніх сервісів

Сучасні месенджер-боти розкладу навчальних занять є поширеним інструментом автоматизації доступу до навчальної інформації в університетах. Такі рішення активно використовуються в ПНУ імені Василя Стефаника, КПІ ім. Ігоря Сікорського, ХНУРЕ, Львівська політехніка та ЛНУ ім. Івана Франка.

Інтерфейс такого сервсісу ПНУ наведено на рисунку 1.1 [4].

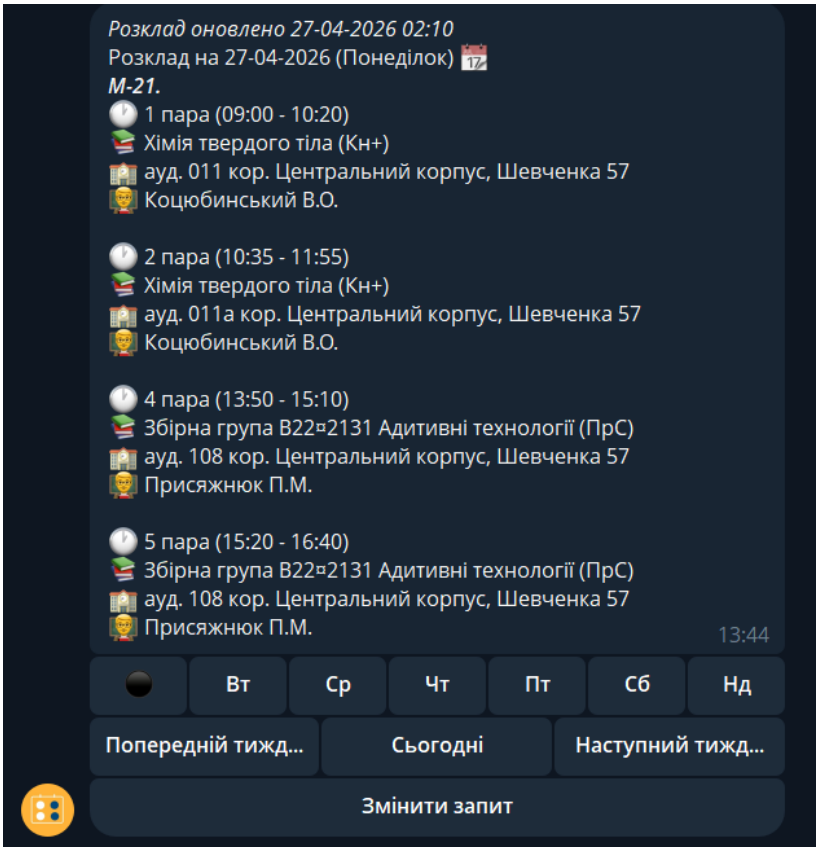


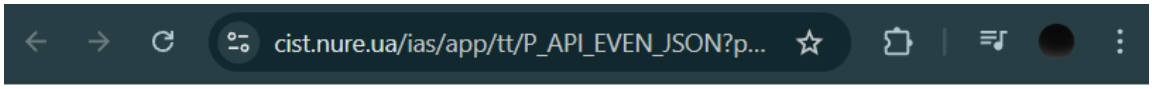
Рисунок 1.1 – Інтерфейс бота для розкладу ПНУ

Незважаючи на однакову функціональну мету - надання розкладу занять у зручному вигляді - архітектурні підходи до реалізації таких систем суттєво відрізняються.

Першочерговим аспектом, який визначає поведінку таких систем, є спосіб отримання даних. Саме він формує подальшу архітектуру: швидкість відповіді, стабільність роботи та залежність від зовнішніх систем.

У практиці університетських рішень можна виділити два основні підходи: використання API та HTML-парсинг.

У випадку ХНУРЕ застосовується централізований сервіс CIST API, який надає доступ до розкладу у структурованому форматі. Це означає, що дані вже мають визначену структуру (JSON), і бот працює не з візуальним представленням сторінки, а з формалізованою моделлю даних.



{ "university": "Kharkiv National University of Radio Electronics", "events": [{ "subject_id": 5234567, "start_time": 1715061000, "end_time": 1715066400, "type": 10, "number_pair": 1, "auditory": "301н", "teachers": [12345], "groups": [7035545] }], "groups": [{ "id": 7035545, "name": "ІТШІ-22-1" }], "teachers": [{ "id": 12345, "short_name": "Іванов І.І." }], "subjects": [{ "id": 5234567, "brief": "Програмування" }] }

Рисунок 1.2 - Приклад відповіді CIST API (ХНУРЕ)

На рисунку 1.2 ми бачимо, що API повертає структуровані об’єкти, що дозволяє уникнути складної логіки парсингу. З точки зору архітектури це зменшує кількість точок відмови: система не залежить від HTML-верстки, а працює через стандартизований інтерфейс [5].

Однак важливо зазначити, що навіть API-рішення не є повністю автономними. У випадку недоступності сервера або перевантаження API, бот втрачає можливість отримання актуальних даних.

У більшості інших університетів, зокрема Львівська політехніка та ЛНУ ім. Івана Франка, відкритий API або відсутній, або не використовується у сторонніх ботах. Це змушує розробників застосовувати HTML-парсинг як основний механізм отримання даних.

У такій архітектурі кожен запит користувача проходить повний цикл:

- HTTP-запит до сторінки розкладу;
- отримання HTML-документа;
- аналіз DOM-структури;
- витяг потрібних елементів;
- формування відповіді.

```
      <div class="view-header">
      <p>2025/2026 навчальний рік</p>
<p>весняний семестр</p>

    </div>
        <div class="view-filters form-group">
      <form class="views-exposed-form" data-drupal-selector="views-exposed-form-students-schedule-new-students-schedule"
  <div class="form--inline form-inline clearfix">
  <div class="form-item js-form-item form-type-textfield js-form-type-textfield form-item-studygroup-abbrname js-form-it
      <label for="edit-studygroup-abbrname" class="control-label js-form-required form-required">Група</label>


  <input data-drupal-selector="edit-studygroup-abbrname" class="form-text required form-control" type="text" id="edit-st



  </div>
<div class="form-item js-form-item form-type-select js-form-type-select form-item-semestr js-form-item-semestr form-grou
      <label for="edit-semestr" class="control-label">Семестр</label>


  <div class="select-wrapper"><select data-drupal-selector="edit-semestr" class="form-select form-control" id="edit-seme
            value="1" selected="selected">1 семестр</option><option
            value="2">2 семестр</option></select></div>



  </div>
<div class="form-item js-form-item form-type-select js-form-type-select form-item-semestrduration js-form-item-semestrdu
      <label for="edit-semestrduration" class="control-label">Половина семестру</label>
```

Рисунок 1.3 - Фрагмент HTML-структури розкладу

Як видно з рисунка, структура сторінки є вкладеною та не має чіткої стандартизації. Це означає, що логіка парсингу напряму залежить від розмітки сайту. Будь-яка зміна класів, тегів або структури таблиць призводить до необхідності оновлення логіки бота [6].

Таким чином, парсинг створює жорстку залежність від візуального шару системи, який не призначений для машинної інтеграції.

Ключовою проблемою обох підходів є залежність від першоджерела даних. У випадку перевантаження серверів університету або їх тимчасової недоступності (що характерно під час сесії), бот або значно збільшує час відповіді, або повністю перестає функціонувати.

У випадку парсингу ця залежність є прямою і критичною, оскільки відсутній будь-який проміжний шар зберігання або резервування даних. У випадку API, як у ХНУРЕ, ситуація дещо краща, однак залежність від доступності сервісу зберігається.

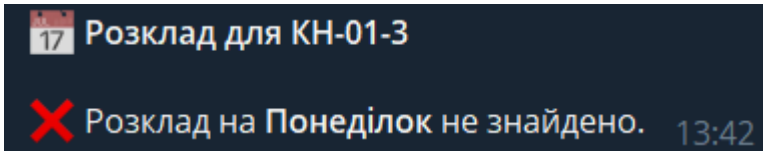


Рисунок 1.4 – Приклад затримки або помилки відповіді при недоступності джерела

Це підтверджує, що основним обмеженням є не сам бот, а зовнішня інфраструктура, на яку він повністю спирається.

Окремим прикладом більш розвиненої архітектури є бот KPI6, що використовується в КПІ ім. Ігоря Сікорського. У цьому рішенні застосовується проміжний шар обробки даних та кешування.

Його основна ідея полягає в тому, що:

- дані розкладу не запитуються щоразу з джерела;
- вони зберігаються у швидкому сховищі;
- користувач отримує відповідь із кешу.

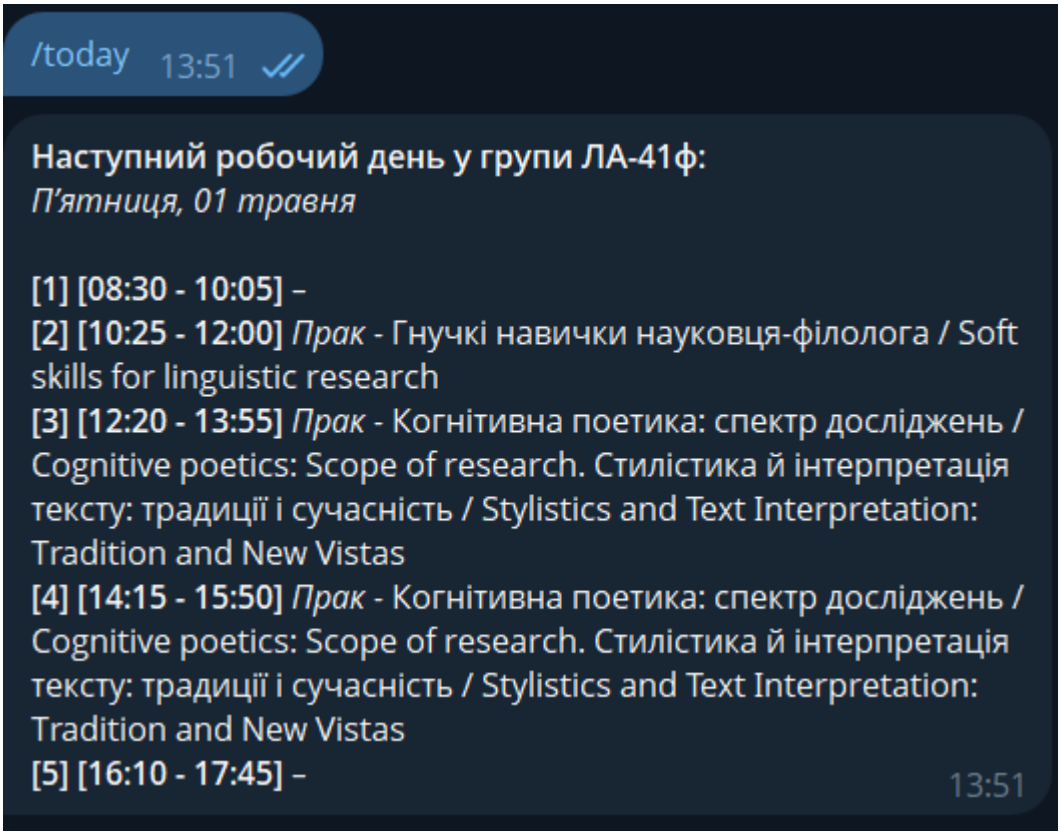


Рисунок 1.5 – Приклад відповіді KPI6 (отримання з кешу)

Завдяки цьому значно зменшується час відповіді та навантаження на зовнішні системи [7]. Однак навіть у цій архітектурі залишається проблема синхронізації даних із першоджерелом, що може призводити до тимчасової неактуальності інформації.

Другим важливим критерієм є швидкість отримання відповіді. У системах, що працюють через парсинг у реальному часі, кожен запит проходить повний цикл обробки, включаючи мережеву взаємодію та аналіз HTML.

Це призводить до середнього часу відповіді на рівні 2–3 секунд. У реальних умовах для ботів Львівська політехніка цей показник може бути ще вищим через навантаження на сервери.

З точки зору користувацького досвіду така затримка є критичною, оскільки взаємодія з месенджером очікується як миттєва.

У системах із кешуванням ситуація суттєво відрізняється. Попередньо збережені дані дозволяють уникати звернення до зовнішнього джерела, що знижує час відповіді до значень менше 0.1 секунди.

Однак це створює іншу проблему — ризик застарівання даних. Якщо оновлення кешу відбувається недостатньо часто, користувач отримує неактуальний розклад.

Таким чином формується базовий архітектурний компроміс:

- без кешу - актуально, але повільно;
- з кешем - швидко, але потенційно неактуально.

Для узагальнення підходів порівняємо їх у таблиці 1.1

Таблиця 1.1 – Порівняння підходів

Критерій	Парсинг HTML	API / кешування
Час відповіді	2–5 сек	<0.1 сек
Стабільність	Низька	Вища
Залежність	Повна	Часткова
Складність підтримки	Висока	Середня
Актуальність	Висока (при доступності)	Залежить від оновлення

Додатково слід врахувати системні обмеження, характерні для більшості існуючих рішень. До них належать:

- відсутність єдиної архітектури між університетами;
- слабка масштабованість при збільшенні кількості користувачів;
- відсутність механізмів контролю стану системи;

- відсутність оптимізації під пікові навантаження.

Це призводить до того, що більшість ботів стабільно працюють лише в умовах стандартного навантаження, але деградують при різкому збільшенні запитів.

Проведений аналіз показує, що існуючі університетські бот-системи вирішують задачу надання доступу до розкладу, однак їх архітектура має суттєві обмеження.

Основні проблеми можна узагальнити таким чином:

- залежність від зовнішніх джерел даних;
- нестабільність при перевантаженні систем;
- компроміс між швидкістю та актуальністю;
- відсутність єдиного ефективного підходу до обробки та доставки даних.

При цьому жоден із розглянутих підходів окремо не забезпечує одночасно високої продуктивності, стабільності та актуальності інформації.

Саме тому найбільш ефективним напрямком розвитку подібних систем є комбінування підходів: використання структурованих джерел даних (API), доповнених механізмами кешування та оптимізації обробки запитів, що дозволяє зменшити залежність від зовнішніх систем та підвищити загальну швидкодію.

1.3 Постановка задачі

Потрібно розробити web-додаток для отримання та обробки розкладу занять ІФНТУНГ, який забезпечує користувачам швидкий доступ до актуальної інформації з офіційного вебресурсу університету. Основною задачею системи є організація ефективного механізму отримання даних про розклад за назвою навчальної групи з мінімальними затримками при обробці запитів, а також забезпечення стабільної роботи у випадку тимчасової недоступності джерела даних.

Система повинна поєднувати дві ключові вимоги: високу швидкість відповіді та актуальність інформації. Для цього передбачається використання

кешування на основі Redis, що дозволяє зменшити кількість звернень до сайту університету. У випадку відсутності актуальної інформації в кеші виконується парсинг вебсторінки «на льоту», що забезпечує отримання свіжих даних.

Важливим аспектом є також відмовостійкість системи: у разі недоступності вебсайту університету користувач повинен отримувати останні збережені (навіть застарілі) дані з Redis, що дозволяє уникнути повної втрати функціональності сервісу.

Вимоги до інформаційної системи:

Функціональні можливості:

- реалізація пошуку розкладу занять за назвою навчальної групи;
- збереження обраної користувачем групи для подальшого швидкого доступу до розкладу;
- забезпечення механізму кешування даних розкладу з використанням Redis у форматі JSON-рядків;
- реалізація логіки пріоритетного доступу до кешу з можливістю автоматичного звернення до вебсайту у разі його відсутності або застарілості даних;
- реалізація функції примусового оновлення даних, яка дозволяє ігнорувати кеш і отримувати актуальну інформацію з джерела;
- забезпечення відмовостійкої роботи системи шляхом використання резервних (застарілих) даних у випадку недоступності зовнішнього ресурсу;
- організація взаємодії користувача з ботом через керування станами діалогу (FSM).

Архітектура програмного забезпечення:

- асинхронна обробка запитів: використання бібліотеки *aiogram* для побудови телеграм-бота, що забезпечує неблокуючу обробку повідомлень користувачів [8];
- мережевий рівень: застосування httpx для виконання асинхронних HTTP-запитів до вебсайту університету та отримання HTML-даних для подальшого опрацювання [9];

– модуль кешування: використання Redis для зберігання розкладу у вигляді серіалізованих JSON-рядків із метою підвищення продуктивності системи;

– логіка доступу до даних: реалізація принципу «спочатку кеш — потім джерело», що дозволяє зменшити навантаження на зовнішній сайт та прискорити відповідь користувачу;

– керування станами (FSM): застосування кінцевого автомата станів для організації багатокрокової взаємодії з користувачем, включаючи вибір та збереження навчальної групи;

– механізм оновлення даних: окремий сценарій отримання актуальної інформації з ігноруванням кешу за ініціативою користувача.

Перевірка працездатності розробленої системи:

– перевірка коректності роботи телеграм-бота шляхом тестування основних сценаріїв взаємодії користувача (вибір групи, отримання розкладу, оновлення даних);

– перевірка механізму кешування шляхом аналізу часу відповіді при повторних запитах до однакових даних;

– перевірка роботи відмовостійкого режиму у випадку імітації недоступності вебсайту університету з контролем повернення даних із Redis;

– тестування логіки FSM для коректного збереження та відновлення станів користувача;

– перевірка функціоналу примусового оновлення через сценарії, у яких кеш ігнорується та виконується запит до зовнішнього ресурсу.

Таким чином, розроблювана система забезпечує ефективну взаємодію з вебресурсом університету, поєднуючи асинхронну обробку запитів, кешування та механізми відмовостійкості. Це дозволяє досягти стабільної роботи сервісу з оптимальним балансом між швидкістю отримання відповіді та актуальністю даних.

2 АНАЛІЗ АРХІТЕКТУРИ СИСТЕМИ

2.1 Вимоги до програмного забезпечення

Для розробки web-додатку підтримки функціонування месенджер-бота сервісу розкладу занять ІФНТУНГ необхідно обрати програмне забезпечення та технології, які забезпечать стабільну роботу системи, швидку обробку запитів користувачів та можливість подальшого розширення функціоналу інформаційної системи.

Для створення серверної частини інформаційної системи та реалізації месенджер-бота було обрано мову програмування Python.

Фактори, які вплинули на вибір мови програмування Python:

– простота та зрозумілість синтаксису, що дозволяє швидко реалізовувати необхідний функціонал;

– наявність великої кількості бібліотек для веб-розробки, роботи з мережею та створення Telegram-ботів;

– підтримка асинхронного програмування для одночасної обробки великої кількості запитів;

– кросплатформність та активний розвиток мови програмування [10].

Для реалізації Telegram-бота використовується бібліотека aiogram, яка забезпечує взаємодію з Telegram Bot API та підтримує асинхронну обробку повідомлень.

Переваги бібліотеки aiogram:

– висока швидкість обробки повідомлень користувачів;

– підтримка FSM для організації сценаріїв взаємодії з користувачем;

– зручна структура побудови Telegram-ботів;

– простота інтеграції з іншими Python-бібліотеками [8].

Для отримання інформації з вебресурсу університету використовується бібліотека httpx.

Переваги бібліотеки httpx:

– підтримка асинхронних HTTP-запитів;

– висока швидкість виконання мережевих операцій;

– зручна обробка HTTP-відповідей [9].

Для реалізації механізму кешування даних використовується система керування базами даних Redis.

Переваги Redis:

– зберігання даних в оперативній пам’яті;

– мінімальний час виконання операцій читання та запису;

– підтримка механізму TTL для автоматичного видалення застарілих даних;

– зменшення навантаження на зовнішній вебресурс університету [3].

Для написання та тестування програмного коду було обрано редактор Visual Studio Code.

Переваги редактора коду Visual Studio Code:

– безкоштовність та доступність;

– простий та зрозумілий інтерфейс;

– наявність розширень для Python та Git;

– вбудований термінал та засоби налагодження програмного коду [9].

Переваги та доступність обраного програмного забезпечення створюють необхідні умови для розробки web-додатку підтримки функціонування месенджер-бота сервісу розкладу занять ІФНТУНГ. Використання Python, aiogram, Redis та httpx дозволяє забезпечити швидку обробку запитів, стабільність роботи системи та ефективне кешування даних розкладу занять.

2.2 Аналіз структури бази даних

Для організації ефективного збереження інформації, забезпечення мінімального часу відгуку системи (latency) та реалізації механізму

оперативного кешування розкладу занять у розроблюваному вебдодатку було обрано високопродуктивну NoSQL СКБД Redis. На відміну від класичних реляційних баз даних, які використовують табличну структуру із жорстко визначеними зв'язками, СКБД Redis функціонує на основі парадигми «ключ-значення» (Key-Value) [3]. У такій архітектурі кожному унікальному ключу-ідентифікатору відповідає певний об'єкт даних у пам'яті (In-memory сховище).

Для забезпечення повноцінного функціонування месенджер-бота сервісу розкладу занять ІФНТУНГ було спроєктовано логічну структуру сховища, яка розподілена на чотири функціональні сегменти ключів. Перший сегмент відповідає за мінімізацію повторних мережевих запитів до джерела даних шляхом збереження отриманих вебдокументів.

Архітектурний опис структури ключа, призначеного для оперативної фіксації та збереження повного вебдокумента розкладу занять конкретної академічної групи, наведено у таблиці 2.1.

Таблиця 2.1 – Сегмент кешу розкладу (Timetable Cache)

Параметр ключа / Поле	**Тип даних у Redis**	**Опис формату ключа / вмісту поля**	**Життєвий цикл (TTL)**	**Вміст та призначення**
Префікс ключа	String	timetable:{group}	259200 сек (3 дні)	Унікальний ідентифікатор кешу
Значення (Value)	String	Сирий HTML-код сторінки	Автоматичне видалення після завершення TTL	Повний HTML відповіді сервера «Деканат» для кешування

Оскільки збережений HTML-код є статичним зліпком даних на певний момент часу, система потребує ідентифікації користувачів для надання їм персоналізованого доступу. Наступний функціональний сегмент відповідає за збереження довготривалих налаштувань профілю, що дозволяє уникнути ручного введення назви групи при кожному новому зверненні до бота.

Специфікацію параметрів для збереження конфігурації та прив'язки студента до відповідної навчальної групи описано у таблиці 2.2.

Таблиця 2.2 – Сегмент профілю користувача (User Profile)

Параметр ключа / Поле	Тип даних у Redis	Опис формату ключа / вмісту поля	Життєвий цикл (TTL)	Вміст та призначення
Префікс ключа	String	user:{user_id}:group	Постійно зберігається (Без TTL)	{user_id} — унікальний цифровий ідентифікатор користувача у месенджері Telegram.
Значення (Value)	String	Текстовий рядок (напр., КІ-22-1)	Зберігається до примусової зміни користувачем	Закріплена академічна група для автоматичного швидкого виклику розкладу занять.

Окрім збереження базового профілю користувача, важливу роль у побудові якісного UI/UX інтерфейсу відіграє контроль над станом активного діалогу. Для того щоб інтерфейс месенджера не накопичував застарілі повідомлення з кнопками та не дезорієнтував користувача, у системі передбачено збереження динамічних технічних маркерів графічного вікна.

Технічні параметри збереження стану користувацького інтерфейсу, що необхідні програмному додатку для динамічної модифікації та вчасного видалення повідомлень у чаті, представлено у таблиці 2.3.

Таблиця 2.3 – Сегмент контролю інтерфейсу (User Interface State)

Параметр ключа / Поле	Тип даних у Redis	Опис формату ключа / вмісту поля	Життєвий цикл (TTL)	Вміст та призначення
Префікс ключа	String	user:{user_id}:last_msg_id	Постійно (динамічно оновлюється)	Зв'язок із поточним відкритим діалогом користувача.
Значення (Value)	Integer	Числовий ідентифікатор	-	ID останнього надісланого ботом інлайн-повідомлення для його модифікації.

Останнім невід'ємним компонентом сховища є координація кроків користувача за принципом кінцевого автомата. Оскільки процес взаємодії з ботом (наприклад, введення назви групи чи вибір дня тижня) є поетапним, додаток має чітко диференціювати, яку саме команду очікувати від користувача у конкретний момент часу. Для цього використовується спеціалізований механізм Hash-сховища.

Параметри контексту кінцевого автомата (FSM), який керує послідовністю кроків реєстрації користувача та поточним станом опитування, наведено у таблиці 2.4.

Таблиця 2.4 – Сегмент станів кінцевого автомата (Aiogram FSM Storage)

Назва поля (Ключ Hash)	Тип даних у Redis	Формат ключа системи FSM	Дозволяє NULL	Вміст та призначення
Ключ запису	Hash	fsm:records:{chat_id}:{user_id}:data	-	Головний структурований ключ стану сесії для конкретного чату та користувача..
state	String	Текстовий рядок (напр., Form:group_set)	-	Поточний зареєстрований крок алгоритму, у якому перебуває користувач.
data	JSON / String	Серіалізований JSON-об'єкт	+	Тимчасові дані контексту (проміжні змінні під час сесії введення назви групи).

Далі визначимо логічні зв’язки між описаними структурами даних NoSQL сховища та принципи організації зовнішнього інформаційного обміну.

Оскільки нереляційна СКБД Redis є розподіленим Key-Value сховищем, у ній на фізичному рівні відсутні класичні механізми підтримки реляційної цілісності даних, такі як каскадні обмеження зовнішніх ключів (Foreign Keys). Проте логічна зв'язність і несуперечливість даних повністю забезпечується безпосередньо архітектурою розробленого програмного додатка (на рівні обробників подій та проміжного ПЗ месенджер-бота).

Головним інтеграційним вузлом та унікальним ідентифікатором у системі виступає номер користувача Telegram (user_id). Цей ідентифікатор динамічно

інтерполюється у назви ключів сегментів профілю (user:{user_id}:group), контролю графічного інтерфейсу (user:{user_id}:last_msg_id) та стану кінцевого автомата (fsm:records:{chat_id}:{user_id}:data). Це утворює логічний зв’язок типу «один до одного» (1:1) між користувачем месенджера та його персональними конфігураційними записами в базі даних.

Наступний рівень абстракції реалізовано через використання назви академічної групи. Значення текстового рядка, що зберігається в профілі користувача, виступає логічним покажчиком (вказівником) на ключ кешу розкладу занять timetable:{group}. Оскільки ідентичний кеш розкладу окремої групи (наприклад, КІ-22-1) можуть одночасно запитувати та переглядати сотні студентів цієї групи, між сегментом персональних профілів та сегментом глобального кешу розкладу виникає логічне відношення типу «багато до одного» (M:1). Такий підхід дозволяє суттєво оптимізувати обсяг використовуваної оперативної пам'яті сервера, оскільки важкий HTML-код розкладу на весь тиждень зберігається в базі даних лише в одному екземпрілі для всього потоку студентів.

Логічну структуру ключів та схему їх взаємодії всередині NoSQL СКБД Redis представлено на рисунку 2.1.

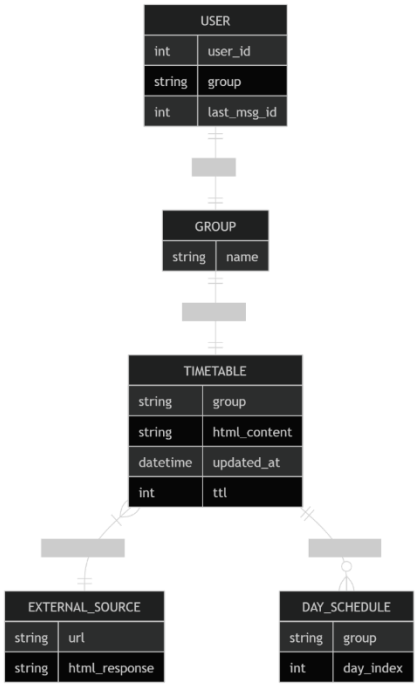


Рисунок 2.1 – Логічна структура ключів та зв’язків NoSQL СКБД Redis

Організація обміну даними зі стороннім сервісом (інформаційною системою розкладу «Деканат» ІФНТУНГ) побудована на базі асинхронного клієнта за неблокуючим принципом. Взаємодія з вебресурсом університету ускладнена тим, що його сервер використовує застаріле кодування тексту Windows-1251 (cp1251), тоді як внутрішня логіка бота оперує сучасним стандартом UTF-8.

Процес обміну даними містить такі кроки:

1. Клієнтський модуль TimetableDownloader ініціює первинний асинхронний GET-запит до скрипта timetable.cgi для встановлення сесії та отримання системних HTTP-cookies.

2. Програма здійснює примусове кодування назви академічної групи за стандартом cp1251 і перетворює її на безпечний URL-encoded рядок.

3. Формується POST-запит з MIME-типом application/x-www-form-urlencoded, де в тіло додаються зашифровані параметри запиту джерела розкладу.

4. Отримана відповідь декодується із кодування cp1251 назад в UTF-8 і надходить до синтаксичного аналізатора (парсера) BeautifulSoup4 для виділення занять на цільовий день тижня.

Спроєктована структура NoSQL бази даних та алгоритм асинхронного обміну даними повністю задовольняють технічні вимоги до розроблюваної інформаційної системи. Використання оперативної пам'яті як основного носія даних у поєднанні з механізмом TTL (Time To Live) тривалістю 3 доби гарантує миттєвий доступ до розкладу занять, високу актуальність інформації та зводить до мінімуму паразитне навантаження на сервери університету в пікові години.

2.3 Сценарії роботи користувача

Розглянемо процес взаємодії користувача з розроблюваним вебдодатком підтримки функціонування месенджер-бота сервісу розкладу занять ІФНТУНГ на базі Redis. Для цього побудуємо діаграму, що має на меті описати кожну

можливу дію користувача із компонентою та зв'язки між цими діями, щоб забезпечити повне розуміння роботи розроблюваної частини інформаційної системи. Вона має надавати всі можливості для повноцінної роботи із інформацією про розклад, такі як: первинна ініціалізація, автентифікація академічної групи, вибір дня тижня, асинхронне отримання даних, примусове оновлення кешу та зміна поточних налаштувань.

Встановлення та валідація назви академічної групи є першою умовою для доступу користувача до інформаційної системи розкладу. Після успішного проходження валідації та фіксації групи користувач матиме можливість виконувати маніпуляції із інформацією, яка зберігається в базі даних. Користувачу цієї інформаційної системи надаватиметься доступ до вибору дня тижня, перегляду занять на поточний або наступний тиждень, а також примусового оновлення кешованих даних.

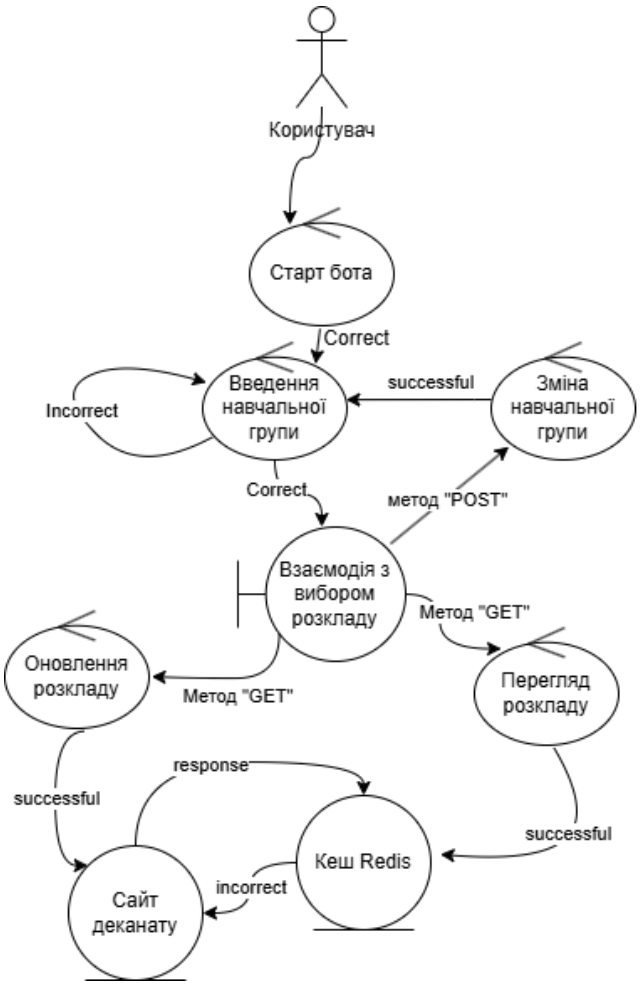


Рисунок 2.2 – Use-case діаграма роботи серверної частини інформаційної системи

На рисунку 2.2 можна побачити, що користувач не матиме доступу до інформаційної системи розкладу, поки скінченний автомат (FSM) не зафіксує і не валідує назву його академічної групи. Дальше з огляду на роботу бекенд частини цієї системи користувач, задіюючи асинхронні обробники подій (handlers) та callback-запити, виконує звернення до NoSQL бази даних Redis і маніпулює інформацією, яка в ній міститься.

Також додамо діаграму взаємодії із сервером, щоб показати як відбувається взаємодія сервера із базою даних:

На діаграмі відбуваються такі дії: користувач заходить у месенджер-бот, щоб переглянути інформацію, в нашому випадку про розклад занять викладачів та студентів, після цього сервер бота надсилає запит до NoSQL бази даних Redis, яка у відповідь користувачу повертає кешовані дані для перегляду (сценарій Cache Hit). Також користувач має можливість примусово оновити розклад або змінити поточну групу, в залежності від маніпуляцій із даними, які он бажає здійснити, сервер бота за допомогою HTTPX-клієнта надсилає запит до зовнішнього сервера «Деканат» університету (сценарій Cache Miss), записує оновлений результат у базу даних Redis із прапорцем TTL і після чого користувач може переглянути результат маніпуляцій з даними.

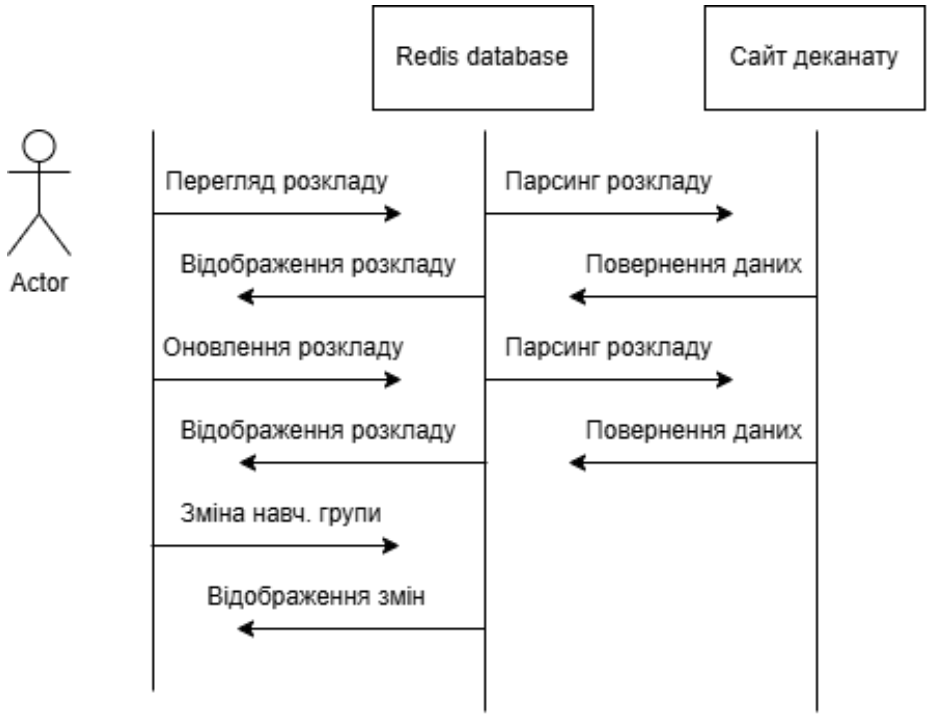


Рисунок 2.3 – Діаграма взаємодії із сервером

Іншими словами, діаграма послідовностей, яка зображена на рисунку 2.3, відображає часові особливості передачі і прийому повідомлень об'єктами.

Такий підхід до моделювання дозволяє чітко розділити внутрішні процеси обробки запитів та знизити навантаження на сервер університету завдяки використанню швидкого кешу. Побудовані діаграми повністю описують логіку роботи користувача з ботом та взаємодію між усіма компонентами програми. Вони наочно показують, як працюють стани бота та як відбувається збереження даних у Redis. Створені схеми є готовою основою для переходу до наступного етапу — написання програмного коду бота, реалізації асинхронних обробників та модулів парсингу.

3 РЕАЛІЗАЦІЯ І ДЕМОНСТРАЦІЯ СИСТЕМИ

3.1 Реалізація програмного забезпечення

Для забезпечення відмовостійкості та швидкої обробки запитів сервісу розкладу ІФНТУНГ в умовах пікових навантажень програмне забезпечення спроектовано за модульним принципом. Код розділено на ізольовані компоненти з чіткими зонами відповідальності: конфігурація, інфраструктурна взаємодія, бізнес-логіка парсингу та інтерфейсне представлення. Проект реалізовано мовою Python на базі асинхронного фреймворку aiogram 3 із застосуванням системи маршрутизації подій (роутерів), що дозволяє ізолювати обробку команд від шару мережевих запитів. Структура проекту наведена на рисунку 3.1.

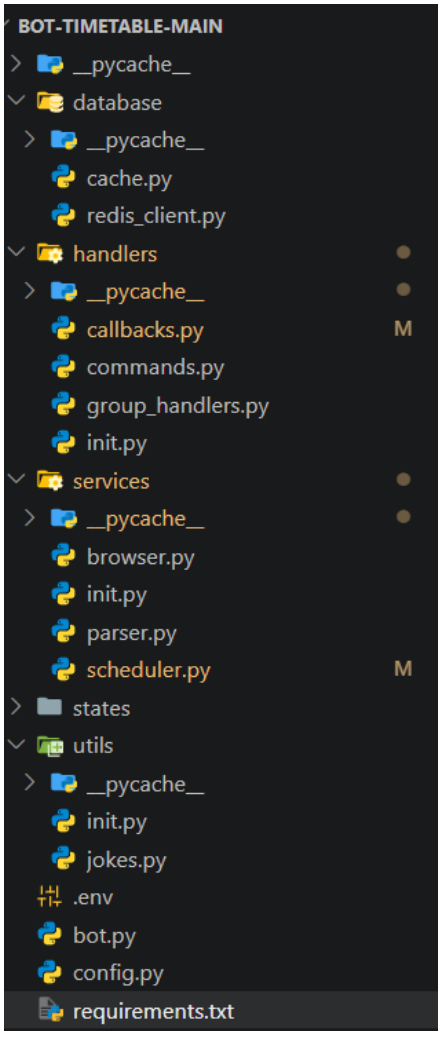


Рисунок 3.1 – Структура проекту

Першим важливим елементом системи є підсистема конфігурації. У будь-якій професійній розробці налаштування програми та секретні ключі доступу ніколи не записуються прямо всередині головного коду. Якщо вписати токен бота безпосередньо в код, то при публікації проекту або передачі його іншим розробникам ці конфіденційні дані стануть доступними стороннім особам, що скомпрометує безпеку всієї системи. Для запобігання цьому в проекті створено прихований файл .env. У ньому зберігаються змінні середовища, які містять токен бота, адресу підключення до бази даних Redis та базову веб-адресу сервера розкладу університету.

Процес зчитування цих даних, їх перевірки та централізованого розподілу по інших модулях програми покладено на файл config.py. Коли бот запускається, цей файл першим зчитує вміст прихованого документа за допомогою спеціальної бібліотеки. Початкові налаштування зчитуються таким чином:

```
import os
from dotenv import load_dotenv

load_dotenv()

BOT_TOKEN = os.getenv("BOT_TOKEN")
REDIS_URL = os.getenv("REDIS_URL", "redis://localhost:6379/0")
DEKANAT_URL = os.getenv("DEKANAT_URL",
"https://dekanat.nung.edu.ua/cgi-bin/timetable.cgi?n=700")
```

Окрім простого зчитування адрес та секретних ключів, модуль конфігурації виконує ще одну важливу функцію — він задає глобальні правила та обмеження для роботи всього алгоритму. Зокрема, тут визначається час, протягом якого завантажений розклад може вважатися свіжим і не потребує повторного оновлення з сайту. Також тут прописується спеціальний текстовий шаблон, який використовується для миттєвої перевірки назви академічної групи, яку вводить студент. Ці параметри записуються в коді так:

```
CACHE_TTL = 3 * 86400
GROUP_PATTERN = r'^[A-ZA-ЯҐЄІЇ]{1,5}-\d{1,3}-\d[A-ZA-ЯҐЄІЇ]{0,3}$'
```

Константа CACHE_TTL містить математичний вираз, який вираховує кількість секунд у трьох добах. Саме такий проміжок часу обрано як оптимальний для збереження інформації у внутрішній пам'яті. Шаблон GROUP_PATTERN представляє собою регулярний вираз. Оскільки в ІФНТУНГ назви всіх академічних груп підпорядковуються суворим стандартам, цей шаблон дозволяє відсікти будь-які випадкові або навмисні помилки введення з боку користувача ще до того, як програма почне витрачати ресурси на обробку запиту. Шаблон перевіряє, щоб спочатку йшли літери (назва спеціальності), потім дефіс, потім дві цифри року вступу, ще один дефіс і номер групи, за яким можуть слідувати додаткові літери за наявності таких груп.

Наступним критично важливим файлом у структурі є bot.py, який виступає головною точкою входу в програму. Його завдання — запустити асинхронний цикл подій, створити екземпляри головних системних класів та підготувати бота до приймання повідомлень. Асинхронність тут відіграє вирішальну роль. На відміну від класичних програм, які виконують кожну дію послідовно (поки один користувач чекає відповіді, інші стоять у черзі), асинхронний бот працює в режимі постійного перемикання завдань. Якщо один запит чекає завантаження сторінки з інтернету, програма не простоює, а обробляє повідомлення інших студентів. Початок ініціалізації в цьому файлі виглядає так:

```
async def main():
    redis = await redis_manager.get_redis()
    storage = RedisStorage(redis)

    bot = Bot(token=BOT_TOKEN)
    dp = Dispatcher(storage=storage)
```

У цьому фрагменті коду ми бачимо, як програма спочатку звертається до менеджера бази даних для отримання активного підключення до Redis. На основі цього підключення створюється спеціальне сховище станів RedisStorage. Воно необхідне диспетчеру (Dispatcher) — головному органу управління бота, який буде приймати всі вхідні події. Диспетчер використовує це сховище, щоб

пам'ятати, з яким саме користувачем він зараз спілкується та на якому етапі діалогу вони перебувають.

Після того як базові інструменти створені, файл bot.py виконує реєстрацію окремих маршрутизаторів (роутерів). Роутери — це спеціальні підмодулі, які дозволяють рознести логіку обробки різних типів повідомлень по окремих файлах. Процес підключення цих компонентів реалізовано наступним чином:

```
dp.include_router(commands.router)
  dp.include_router(group_handlers.router)
  dp.include_router(callbacks.router)

  dp.startup.register(on_startup)
  dp.shutdown.register(on_shutdown)
```

Завдяки такій структурі, диспетчер знає, що якщо користувач надіслав команду, її треба передати першому роутеру, якщо ввів назву групи — другому, а якщо натиснув кнопку на екрані — третьому. Також тут реєструються функції, які виконуються автоматично при старті програми (наприклад, надсилання сповіщення адміністратору про запуск) та при її вимкненні для безпечного закриття всіх з'єднань.

Завершальний етап роботи головного файлу полягає у запуску процесу безперервного сканування нових повідомлень від серверів Telegram. Цей процес організовано всередині захисного блоку, який гарантує, що за будь-яких умов ресурси комп'ютера будуть звільнені коректно:

```
try:
    await dp.start_polling(bot)
finally:
    await bot.session.close()
```

Метод start_polling переводить програму в режим постійного очікування. Бот робить швидкі запити до серверів Telegram, перевіряючи, чи не з'явилися нові повідомлення від студентів. Якщо програму зупиняє адміністратор, блок finally автоматично закриває відкриту сесію бота, запобігаючи «зависанню» мережевих портів сервера.

Окреме і дуже важливе місце в структурі проекту займає інфраструктурний шар взаємодії з базою даних Redis, який реалізовано у файлі redis_client.py. Оскільки база даних Redis працює безпосередньо з оперативною пам'яттю, вона здатна віддавати та записувати інформацію за лічені мілісекунди. Однак, якщо кожен модуль бота буде самостійно відкривати нове підключення до бази даних при кожному запиті, оперативна пам'ять сервера швидко переповниться відкритими мережевими сокетами, і система заблокує сама себе.

Щоб вирішити цю проблему на архітектурному рівні, було використано відомий патерн проектування під назвою Singleton (або Одинак). Суть цього підходу полягає в тому, що у всій програмі може існувати лише один єдиний об'єкт класу управління базою даних. Створення цього механізму виглядає так:

```
class RedisManager:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
            cls._instance.redis = None
        return cls._instance
```

Магічний метод __new__ перехоплює спробу створення нового об'єкта класу. Якщо об'єкт ще ні разу не створювався (змінна _instance дорівнює None), програма створює його і запам'ятовує. При всіх наступних спробах створити новий екземпляр у будь-якому іншому файлі проекту, система буде просто повертати посилання на той самий перший об'єкт. Це гарантує стабільність використання пам'яті.

Для безпосереднього керування асинхронним з'єднанням всередині цього ж класу реалізовано функцію, яка створює клієнт бази даних безпосередньо в момент першого реального звернення:

```
    async def get_redis(self):
        if not self.redis:
            self.redis = await aioredis.from_url(REDIS_URL)
        return self.redis
```

Такий метод називається лінивою ініціалізацією. Бот не витрачає час на відкриття сокетів під час старту, а підключається до Redis лише тоді, коли з'являється перший реальний запит від користувача.

На основі цього єдиного підключення функціонує підсистема кешування, описана у файлі cache.py. Зазвичай розклад занять у межах одного тижня змінюється вкрай рідко, тому завантажувати сторінку з сайту університету при кожній вимозі студента — це неефективно. Це створює величезне навантаження на сервер деканату і змушує користувача чекати по кілька секунд.

У розробленому ПЗ застосовано стратегію розумного кешування: ми зберігаємо у пам'яті Redis не окремі пари чи дні, а весь сирий HTML-код сторінки розкладу для конкретної групи під унікальним текстовим ключем. Функція для перевірки наявності сторінки у пам'яті виглядає наступним чином:

```
class TimetableCache:
    async def get(self, group: str, day_idx: int, target_date: str)
-> str | None:
        cache_key = f"timetable:{group}"
        redis = await redis_manager.get_redis()

        html = await redis.get(cache_key)
        if html:
            return html.decode('utf-8')
        return None
```

Коли студент просить розклад, програма спочатку формує унікальний ключ проекту виду timetable:назва_групи та перевіряє, чи є такий запис у пам'яті Redis. Якщо запис знайдено, він миттєво зчитується, перетворюється з байтового формату назад у звичайний текст і повертається для подальшого аналізу. Якщо ж пам'ять порожня, бот завантажує сторінку з інтернету і записує її в базу даних за допомогою такої функції:

```
    async def set(self, group: str, html: str):
        cache_key = f"timetable:{group}"
        redis = await redis_manager.get_redis()
        await redis.setex(cache_key, 259200, html)
```

Тут використовується спеціальна команда setex. Вона не просто записує рядок у базу даних, а одночасно встановлює для нього таймер самознищення. Параметр 259200 вказує базі даних, що рівно через три доби цей запис потрібно автоматично видалити з пам'яті. Це дозволяє підтримувати баланс: оперативна пам'ять сервера не засмічується застарілою інформацією, а розклад оновлюється автоматично кожні три дні, що гарантує його актуальність для студентів.

Наступна важлива ланка системи — модуль асинхронного завантаження інформації з мережі, який знаходиться у файлі browser.py. Офіційний веб-сервер розкладу нашого університету побудований на базі застарілих скриптів CGI. Головна проблема цього сервера полягає в тому, що він працює виключно зі старим кодуванням тексту Windows-1251 (CP1251) та очікує отримання параметрів через POST-запити у певному форматі. Якщо сучасна програма спробує відправити туди назву групи у звичайному форматі UTF-8, сервер університету або поверне порожню сторінку, або видасть помилку.

Для розв'язання цієї проблеми в класі TimetableDownloader було створено механізм ручного побайтового кодування даних. Конструктор цього класу налаштовує мережевий клієнт із жорсткими обмеженнями:

```
class TimetableDownloader:
    def __init__(self):
        self.client = httpx.AsyncClient(
            timeout=15.0,
            follow_redirects=True,
            verify=False,
            limits=httpx.Limits(max_connections=5)
        )
```

Встановлення параметру max_connections=5 є важливим інженерним рішенням. Воно обмежує максимальну кількість одночасних з'єднань з сервером університету до п'яти штук. Це повністю захищає сервер деканату від перевантаження або випадкової DOS-атаки з боку бота, якщо сотні студентів одночасно натиснуть кнопки оновлення.

Сам процес отримання сторінки розкладу реалізовано у функції fetch_timetable. На першому етапі бот робить швидкий запит до головної сторінки, щоб імітувати роботу реального браузера, після чого переводить назву групи у потрібне для сервера кодування:

```
async def fetch_timetable(self, group: str) -> str:
    try:
        await  self.client.get("https://dekanat.nung.edu.ua/cgi-bin/timetable.cgi?n=700")

        encoded_group = urllib.parse.quote(group.encode("cp1251"))
        encoded_sbtn = urllib.parse.quote("Переглянути".encode("cp1251"))
        payload_body = f"n=700&obj=group&group={encoded_group}&tw=0&sbtn={encoded_sbtn}"
```

Функція urllib.parse.quote бере текстовий рядок, переведений у байти кодування cp1251, і перетворює його на спеціальний набір символів, який зрозумілий старому веб-серверу. Після формування тіла запиту програма виконує безпосередню відправку даних:

```
        response = await self.client.post(
            DEKANAT_URL,
            content=payload_body.encode("ascii"),
            headers={"Content-Type":    "application/x-www-form-urlencoded"}
        )

        if response.status_code == 200:
            response.encoding = 'cp1251'
            return response.text
        return ""
    except:
        return ""
```

Бот відправляє сформовані дані за допомогою POST-запиту, вказуючи у заголовках, що це стандартна форма введення. Після отримання відповіді від

сервера з кодом 200 (що означає успішне виконання), програма примусово вказує, що отриманий текст потрібно читати в кодуванні cp1251. Це дозволяє уникнути появи незрозумілих символів («ієрогліфів») у назвах предметів та прізвищах викладачів.

Після того як сирий HTML-код сторінки успішно отримано (з бази даних або безпосередньо з мережі), його необхідно розібрати та перетворити на гарне повідомлення для месенджера. Цю функцію виконує синтаксичний аналізатор, розташований у файлі parser.py. Сторінка сайту університету представляє собою велику таблицю з багатьма тегами. Крім того, верстка сайту має певні дефекти: наприклад, текст дати часто злипається з назвою дня тижня, а якщо в якийсь день у студентів немає пар, таблиця може зміщуватися.

Клас TimetableParser використовує бібліотеку BeautifulSoup4 для побудови об'єктного дерева сторінки. Спочатку функція аналізу шукає на сторінці тег заголовка <h4>, який відповідає обраному користувачем дню тижня:

```
class TimetableParser:
    @staticmethod
    def parse(html: str, day_idx: int) -> str:
        soup = BeautifulSoup(html, "html.parser")
        day_names = ["Понеділок", "Вівторок", "Середа", "Четвер",
"П'ятниця", "Субота"]
        target_day_name = day_names[day_idx]

        target_h4 = None
        for h4 in soup.find_all("h4"):
            if target_day_name.lower() in h4.get_text().lower():
                target_h4 = h4
                break
```

Програма перебирає всі заголовки на сторінці і порівнює їх текст із назвою дня, який вибрав студент (наприклад, "Вівторок"). Якщо такий заголовок не знайдено, бот відразу повертає повідомлення про те, що розкладу на цей день немає. Якщо заголовок знайдено, запускається алгоритм очищення тексту та збору рядків таблиці:

```
        if not target_h4:
                return f"❌ Розклад на <b>{target_day_name}</b> не
знайдено."

            raw_title = target_h4.get_text(strip=True)
            formatted_title = re.sub(r'(\d{2}\.\d{2}\.\d{4})([А-Яа-я])',
r'\1 \2', raw_title)

            rows = []
            current_element = target_h4
            while True:
                current_element = current_element.find_next(["tr",
"h4"])
                if not current_element or current_element.name == "h4":
                    break
                if current_element.name == "tr":
                    rows.append(current_element)
```

За допомогою регулярного виразу re.sub програма знаходить місце, де цифри дати злипаються з літерами дня тижня, і автоматично вставляє туди пробіл для гарного відображення. Далі запускається цикл while, який рухається по сторінці вниз від знайденого заголовка. Він збирає всі рядки таблиці <tr>, які відносяться до цього конкретного дня, і зупиняється, як тільки зустрічає заголовок наступного дня тижня (<h4>) або кінець документа. Зібрані рядки потім детально розбираються: програма витягує час пари, назву дисципліни, тип заняття (лекція чи практика) та кабінет. Якщо всередині тексту знайдено посилання на системи дистанційного навчання, бот автоматично оформлює їх як клікабельні гіперпосилання.

Для забезпечення чіткого порядку у спілкуванні бота з користувачем використовується механізм керування станами діалогу (кінцевий автомат). Він описаний у файлі form_states.py. Без цього механізму бот не знав би, як реагувати на звичайний текст: чи це студент вводить назву своєї групи, чи це він просто пише якесь питання. Опис станів виглядає дуже просто і лаконічно:

```
from aiogram.fsm.state import StatesGroup, State

class Form(StatesGroup):
    waiting_for_group = State()
    group_set = State()
```

Ці два стани працюють як перемикачі режиму роботи. Стан waiting_for_group означає, що бот зараз чекає від користувача введення назви групи. Стан group_set вмикається тоді, коли назва групи успішно перевірена, і тепер користувач може обирати дні тижня та переглядати розклад.

Первинна взаємодія з користувачем починається з обробки системних команд, яка зібрана у файлі commands.py. Коли користувач вперше знаходить бота і натискає кнопку запуску, програма отримує команду /start. Бот повинен скинути всі попередні записи про цього користувача і перевести його в режим першого стану:

```
@router.message(Command("start"))
async def cmd_start(message: types.Message, state: FSMContext):
    await message.answer(
        "👋 Введіть назву вашої групи (наприклад, КІ-22-1): ",
        reply_markup=types.ReplyKeyboardRemove()
    )
    await state.set_state(Form.waiting_for_group)
```

Бот надсилає текстове запрошення та примусово видаляє з екрана будь-які старі кнопки за допомогою інструменту ReplyKeyboardRemove. Після цього рядок set_state вмикає стан очікування групи. Тепер будь-яке наступне текстове повідомлення від цього користувача буде оброблятися спеціальним модулем перевірки.

Цей модуль перевірки та реєстрації знаходиться у файлі group_handlers.py. Він перехоплює текст, який надіслав студент, переводить його у верхній регістр (робить літери великими) та виконує порівняння з конфігураційним шаблоном:

```
@router.message(Form.waiting_for_group)
async def set_group(message: types.Message, state: FSMContext):
    group = message.text.strip().upper()
```

```
        if not re.fullmatch(GROUP_PATTERN, group):
            await message.answer("✗ Неправильний формат групи. Спробуйте
ще раз.")
            return

        await state.update_data(group=group)
```

Якщо введений текст не відповідає правилам регулярного виразу (наприклад, студент зробив помилку або написав слово «привіт»), бот надсилає повідомлення про помилку і перериває виконання функції за допомогою команди return. Користувач залишається у поточному стані очікування. Якщо ж перевірка пройшла успішно, назва групи записується у внутрішню пам'ять поточного стану.

Після успішного збереження назви групи бот повинен перебудувати інтерфейс чату для користувача: надіслати йому постійні кнопки меню та динамічні інлайн-кнопки для вибору днів тижня. Також бот виконує важливу дію — фіксує ідентифікатори у базі даних Redis:

```
        await message.answer(
            f"✅ Групу <b>{group}</b> встановлено!",
            reply_markup=ScheduleKeyboard.get_main_reply_keyboard(),
            parse_mode="HTML"
        )

        msg = await message.answer(
            "Оберіть день тижня:",
            reply_markup=ScheduleKeyboard.generate_days_keyboard(),
            parse_mode="HTML"
        )

        redis = await redis_manager.get_redis()
        await redis.set(f"user:{message.from_user.id}:group", group)
        await      redis.set(f"user:{message.from_user.id}:last_msg_id",
msg.message_id)
        await state.set_state(Form.group_set)
```

Запис значення msg.message_id під ключем last_msg_id в базу даних Redis є унікальним рішенням для покращення зручності використання бота. Коли студент буде натискати на кнопки різних днів тижня, бот не буде надсилати щоразу нові повідомлення (що швидко засмітило б весь екран і змусило б постійно гортати історію чату вгору-вниз). Замість цього бот знає номер першого повідомлення і буде просто редагувати текст всередині нього. Наприкінці цієї функції автомат перемикається у фінальний стан group_set.

Основна повсякденна логіка роботи користувача з ботом зібрана у файлі callbacks.py. Цей модуль обробляє події, які виникають при натисканні на інлайн-кнопки (кнопки, які закріплені безпосередньо під текстом повідомлення). Коли користувач натискає на кнопку певного дня тижня, програма зчитує його поточні дані:

```
@router.callback_query(Form.group_set)
async def process_day_selection(callback: types.CallbackQuery, state: FSMContext):
    user_data = await state.get_data()
    group = user_data.get("group")

    if callback.data.startswith("day_"):
        day_idx = int(callback.data.split("_")[1])
        await callback.message.edit_text(f" Отримую розклад для {group}...")
```

Програма дізнається назву групи, яку вибрав цей студент, та визначає індекс дня тижня (від 0 для понеділка до 5 для суботи), який зашифрований у самій кнопці. Після цього на екрані миттєво з'являється напис про те, що дані завантажуються, щоб користувач бачив реакцію бота на свій клік. Далі запускається основний алгоритм пошуку та відображення розкладу:

```
        try:
            html = await cache.get(group, day_idx, "")
            if not html:
                html = await playwright_mgr.fetch_timetable(group)
                await cache.set(group, html)
```

```
                timetable = parser.parse(html, day_idx)
                await callback.message.edit_text(
                    f"🗓 <b>Розклад для {group}</b>\n\n{timetable}",

reply_markup=ScheduleKeyboard.generate_days_keyboard(),
                    parse_mode="HTML"
                )
```

Цей блок коду повністю об'єднує роботу всіх раніше описаних модулів. Він звертається до об'єкта кешу cache. Якщо сторінка для цієї групи вже завантажувалася протягом останніх трьох днів, вона миттєво дістається з оперативної пам'яті Redis. Якщо ж кеш порожній, викликається мережевий менеджер завантаження, який робить POST-запит до сайту університету та зберігає отриманий результат у пам'ять на майбутнє. Після цього сирий код сторінки передається в аналізатор parser.parse, який формує чистий текстовий розклад. Наприкінці метод edit_text замінює поточний текст повідомлення на готовий розклад, залишаючи ті самі кнопки днів тижня знизу для можливості подальшого вибору.

Для того щоб унеможливити поломку програми при виникненні будь-яких непередбачуваних помилок (наприклад, повна відсутність інтернету на сервері або раптова зміна структури коду на сайті університету), завершення обробки подій кнопками завжди захищається спеціальними блоками:

```
            except Exception as e:
                logger.error(f"Error: {e}")
                await callback.message.edit_text("⚠ Помилка
завантаження.",

reply_markup=ScheduleKeyboard.generate_days_keyboard())
            finally:
                await callback.answer()
```

Якщо стається збій, програма записує детальну технічну інформацію в журнал логів для адміністратора, а користувачу замість незрозумілої системної помилки виводить акуратне попередження про тимчасові технічні проблеми.

Блок finally виконується у будь-якому випадку, навіть якщо сталася критична помилка. Функція callback.answer() є обов'язковою для Telegram API — вона дає сигнал додатку на телефоні чи комп'ютері студента, що натискання кнопки успішно зафіксоване, завдяки чому на кнопці зникає кругова анімація завантаження.

За створення та оформлення самих кнопок у проекті відповідає окремий сервісний файл scheduler.py. Його головна інженерна особливість полягає в тому, що він працює повністю динамічно і підлаштовується під реальний час. Клас ScheduleKeyboard використовує бібліотеку pytz для точної синхронізації з часовим поясом України (Europe/Kyiv). Коли програма генерує клавіатуру для вибору днів, вона не просто пише на кнопках "Понеділок" чи "Вівторок". Бот бере поточну дату сервера, за допомогою математичного інструменту timedelta вираховує календарне число для кожного робочого дня поточного тижня і додає його безпосередньо у назву кнопки. Наприклад, студент бачить кнопки у форматі "Понеділок 18.05", "Вівторок 19.05" тощо. Це повністю виключає плутанину, якщо студент користується ботом на вихідних або під час переходу між різними навчальними семестрами.

Такий детально продуманий розподіл всієї програми на окремі невеликі та незалежні модулі дозволив створити дуже надійне програмне забезпечення для підтримки функціонування месенджер-бота.

3.2 Розгортання та конфігурація середовища розробки проекту

Практична реалізація та відлагодження компонентів асинхронного месенджер-бота виконується у спеціалізованому інтегрованому середовищі розробки (IDE). Конфігурація середовища передбачає підготовку ізольованого системного оточення проекту, підключення внутрішніх інструментів аналізу коду, інтеграцію залежностей через менеджер пакетів, а також запуск локальних портів СКБД Redis для забезпечення безперебійної перевірки транзакцій.

На першому етапі в середовищі розробки ініціалізується структура проекту та перевіряється базовий інтерпретатор мови Python, версія якого для повної підтримки асинхронних декораторів aiogram 3.x має бути не нижче 3.10.

Рисунок 3.2 – Вікно конфігурації інтерпретатора Python

Для забезпечення повторюваності розгортання програмного комплексу на будь-якому сервері, всі архітектурні залежності та сторонні бібліотеки суворо зафіксовані у конфігураційному файлі requirements.txt. Його структура містить перелік сумісних версій модулів:

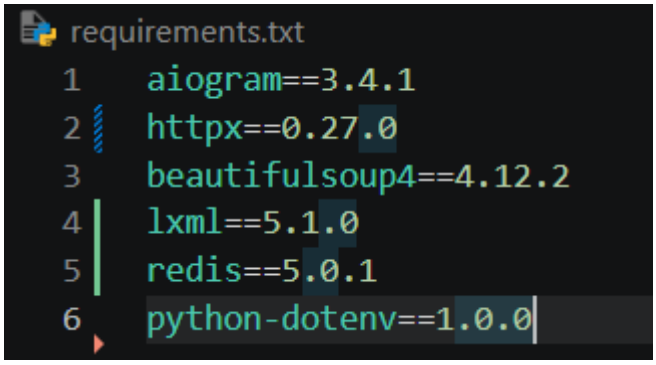


Рисунок 3.3 – Структура конфігураційного файлу

Встановлення всього зазначеного програмного стеку та бінарних розширень (зокрема, швидкого C-парсера lxml) виконується за допомогою вбудованого терміналу середовища розробки через єдину команду менеджера пакетів: `pip install -r requirements.txt`

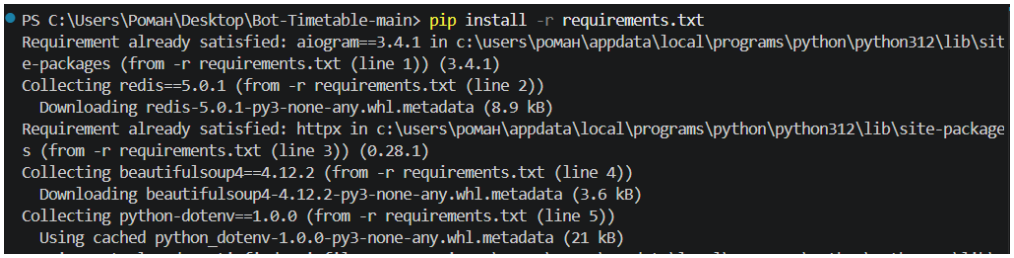


Рисунок 3.4 – Консоль IDE під час автоматичного завантаження та інсталяції залежностей проекту

Наступним кроком є підготовка інфраструктурного шару для роботи системи кешування та збереження проміжних станів (FSM). Для цього запускається локальний сервер СКБД Redis. Конфігурація здійснюється через виконання бінарного файлу із жорстким закріпленням стандартного мережевого порту: `redis-server.exe --port 6380`

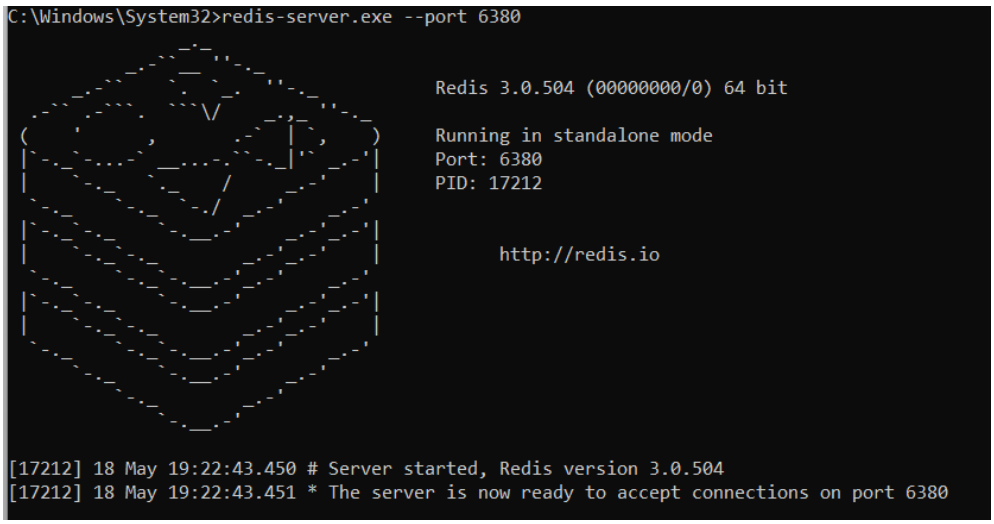


Рисунок 3.5 – Термінал запуску локального сервера Redis

Для перевірки активності сокетів бази даних та успішності встановлення каналу зв'язку всередині IDE відкривається додаткова сесія терміналу, де викликається діагностична утиліта командного рядка:

```
redis-cli.exe -p 6379
```

Всередині інтерактивного середовища тестування відправляється базова низькорівнева команда перевірки зв'язку, яка має повернути стандартну текстову відповідь:

Рисунок 3.6 – Інтерфейс утиліти redis-cli із результатом виконання діагностичної команди

Після підтвердження готовності бази даних до приймання транзакцій, у кореневому каталозі проекту створюється файл конфігурації змінних оточення .env. Цей документ ізолює конфіденційні токени та локальні адреси:

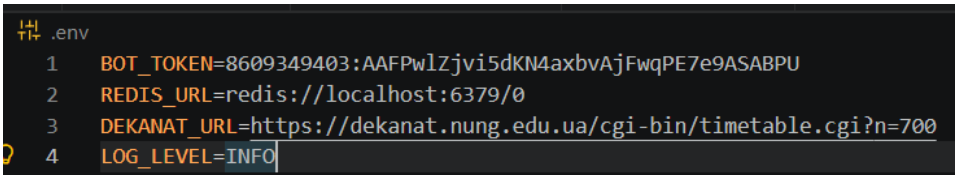


Рисунок 3.7 – Відображення структури файлу .env

Параметр BOT_TOKEN містить унікальний ключ авторизації Telegram API, а нуль наприкінці рядка REDIS_URL вказує системі на використання нульового логічного індексу бази даних всередині оперативної пам'яті Redis.

Для проведення первинного відлагодження коду та фінальної перевірки взаємодії всіх модулів, у терміналі середовища запускається виконання головного керуючого файлу програми:

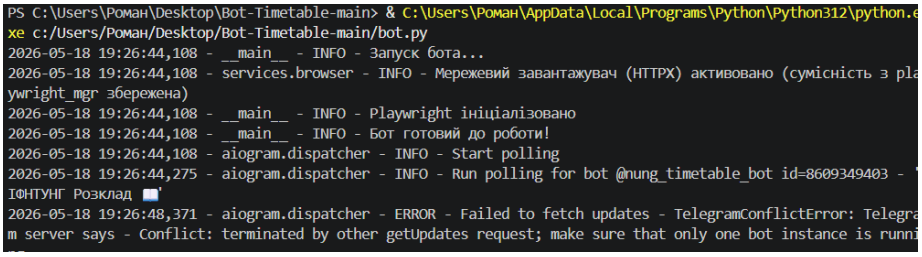


Рисунок 3.8 – Консоль IDE з логами успішного запуску бота

При старті додаток зчитує змінні з конфігураційного файлу, підключає єдиний пул з'єднань до Redis за патерном Singleton, реєструє асинхронні роутери

обробки повідомлень та переходить у режим безперервного сканування подій Telegram, що свідчить про успішне розгортання робочого середовища.

3.3 Демонстрація роботи ПЗ на прикладі месенджера Telegram

Для підтвердження працездатності, коректності роботи з базою даних Redis та стабільності взаємодії розробленого програмного комплексу необхідно провести тестування основних користувацьких сценаріїв безпосередньо в інтерфейсі клієнтського додатка Telegram. Тестування охоплює процеси першого запуску, вибору днів тижня, примусового оновлення даних та зміни поточної академічної групи.

Сценарій взаємодії починається з первинного пошуку месенджер-бота в системі та виклику базової керуючої команди. До моменту першого запуску або після використання системного скидання діалогове вікно бота є порожнім, а клавіатура прихована. Процес ініціалізації сервісу запускається натисканням кнопки «Старт» або введенням команди /start.

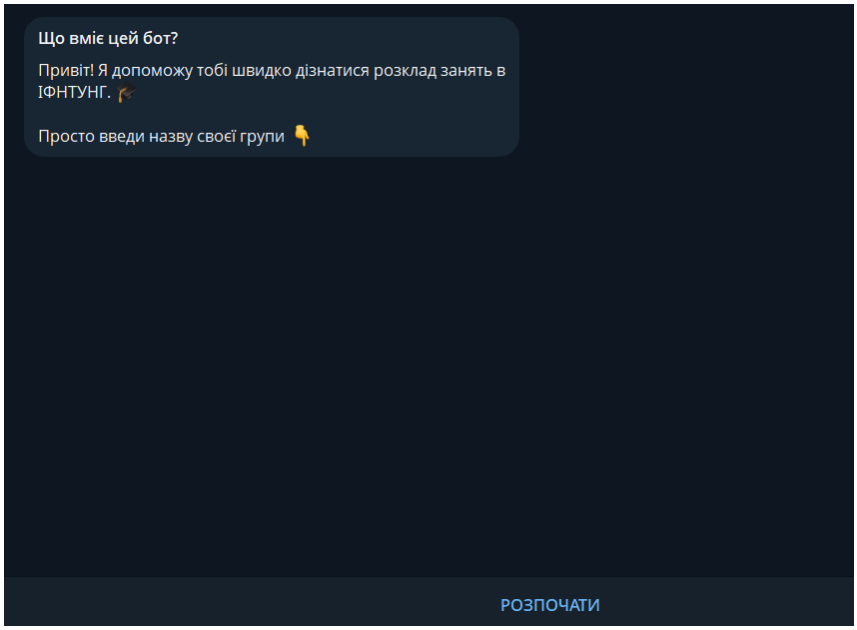


Рисунок 3.9 – Інтерфейс месенджер-бота перед первинним запуском

Після активації команди /start диспетчер подій (Dispatcher) викликає обробник із модуля commands.py. Бот переводить поточну сесію користувача у

стан Form.waiting_for_group, реєструє це значення у сховищі станів RedisStorage та надсилає текстове повідомлення-інструкцію, яка просить студента ввести назву його академічної групи.

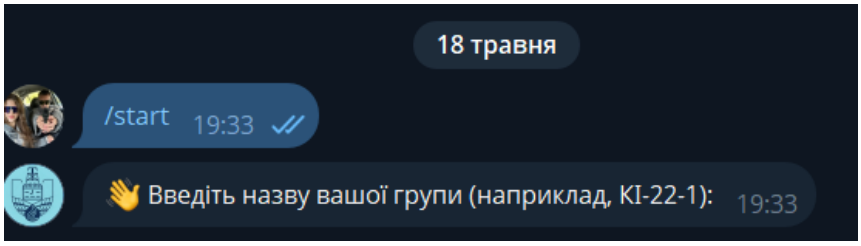


Рисунок 3.10 – Відповідь системи на команду запуску та перехід у стан очікування групи

Наступним кроком є перевірка стійкості системи до некоректного введення даних (валідація). Якщо користувач припускається помилки, вводить неіснуючий формат або сторонній текст (наприклад, «Привіт»), вбудований регулярний вираз GROUP_PATTERN у модулі group_handlers.py блокує запит. Бот надсилає сповіщення про помилку і залишає користувача в поточному стані очікування до моменту введення правильного шифру.

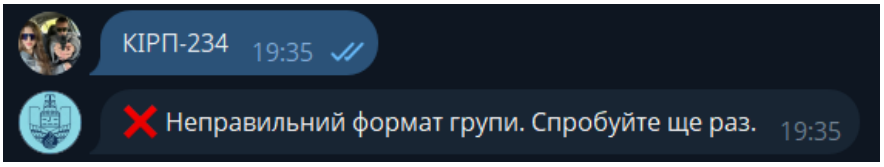


Рисунок 3.11 – Спрацювання алгоритму валідації при спробі введення некоректної назви групи

Коли студент вводить валідну назву групи згідно зі стандартами університету (наприклад, КІ-22-1), додаток успішно проходить перевірку re.fullmatch. Програма переводить текст у верхній регістр, записує назву групи у довготривалу пам'ять Redis під ключем user:id:group та змінює прапорець кінцевого автомата на Form.group_set. Користувач отримує повідомлення про успішне налаштування, головне меню знизу екрана та динамічну інлайн-клавіатуру для вибору днів тижня.

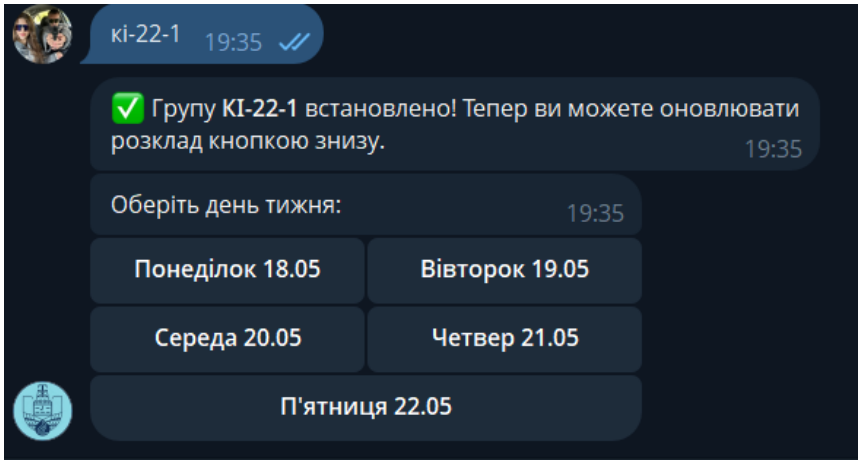


Рисунок 3.12 – Успішна фіксація академічної групи та генерація інлайн-клавіатури днів

При натисканні на кнопку будь-якого дня тижня (наприклад, «Вівторок»), тригер генерує подійний сигнал CallbackQuery, який перехоплюється модулем callbacks.py. Бот зчитує з Redis назву групи, формує унікальний ключ кешу timetable:КІ-22-1 і перевіряє наявність даних. Якщо запит здійснюється вперше, бот надсилає швидкий POST-запит через асинхронний клієнт httpx до сервера деканату, завантажує сторінку, очищає її через BeautifulSoup4 та виводить структурований розклад у чат.

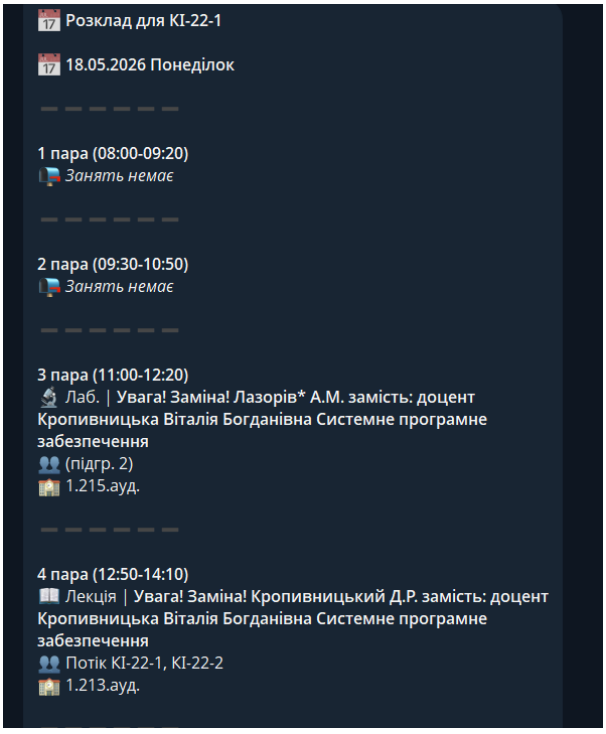


Рисунок 3.13 – Виведення розкладу занять на обраний день

Виведення інформації реалізовано за допомогою технології редагування поточного повідомлення (edit_text). Завдяки цьому інтерфейс не створює нових реплік у чаті, а оновлює вміст поточного вікна, зберігаючи інлайн-кнопки знизу тексту. Це дозволяє студенту швидко перемикатися між іншими днями тижня в один клік, отримуючи миттєву реакцію інтерфейсу без мережевих затримок за рахунок зчитування з кешу Redis.

Важливою функцією є можливість примусового оновлення розкладу занять у реальному часі. Для цього в нижньому головному меню передбачена кнопка «Оновити розклад». При її натисканні програма повністю ігнорує поточний збережений кеш в оперативній пам'яті сервера, повторно звертається до веб-ресурсу університету, завантажує свіжі дані та оновлює повідомлення на екрані, виводячи плашку із часом останнього синхрону.

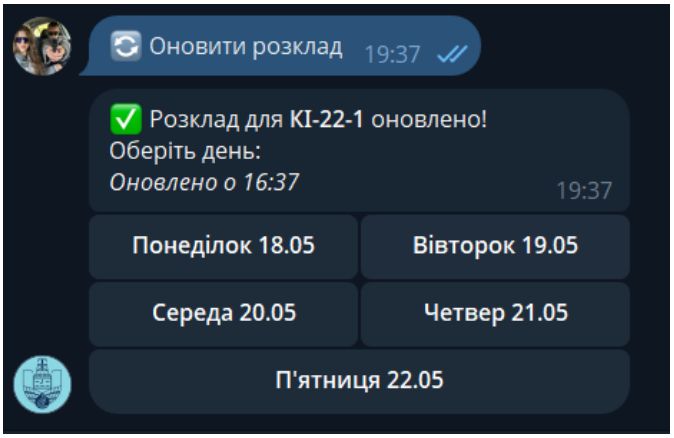


Рисунок 3.14 – Оновлення текстового блоку розкладу після натискання кнопки синхронізації

Для зміни поточної академічної групи (наприклад, якщо студент перевівся або хоче переглянути розклад іншої групи) у нижньому головному меню реалізована кнопка «Змінити групу». При її натисканні бот скидає поточний стан Form.group_set, перемикає автомат назад на крок Form.waiting_for_group і знову просить користувача ввести новий шифр, очищаючи попередні прив'язки в базі даних.

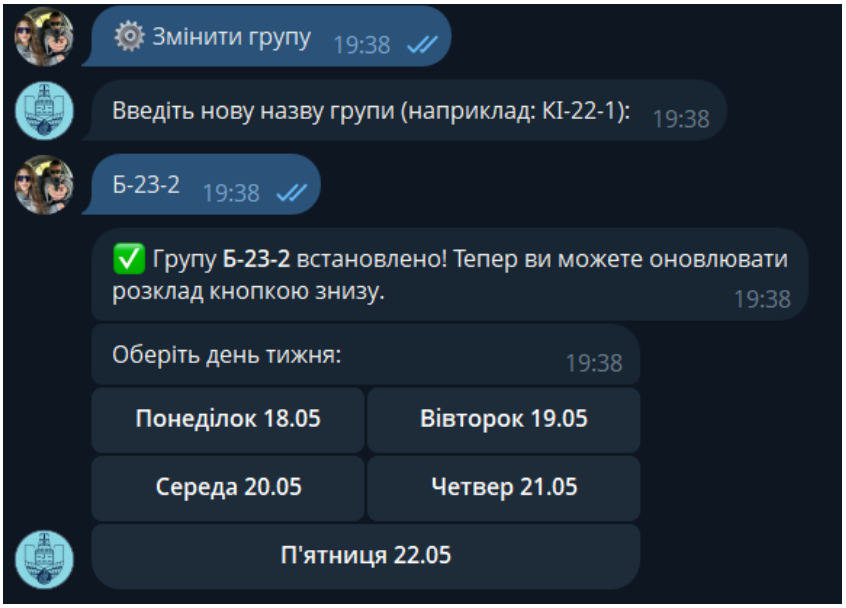


Рисунок 3.15 – Скидання стану та повернення до кроку вибору групи

Проведене тестування доводить, що розроблене програмне забезпечення повністю виконує поставлені завдання: успішно взаємодіє з Telegram API, коректно обробляє вхідні стани користувачів через кінцевий автомат, ізолює помилки введення, миттєво реагує на натискання кнопок та забезпечує швидку динамічну зміну контенту й налаштувань користувача.

ВИСНОВКИ

В ході виконання цієї роботи було розроблено вебдодаток підтримки функціонування месенджер-бота для сервісу розкладу занять ІФНТУНГ. Ця компонента відслідковує зміни та надає оперативний доступ до даних про навчальний процес університету. Вона охоплює інформацію про академічні групи, актуальний розклад на тиждень, поточні стани взаємодії з користувачами та параметри кешування даних.

Проведено аналіз сервісів-аналогів та інформації, яку вони надають своїм користувачам, визначено переваги та недоліки цих рішень для покращення власної системи. Визначено ключові атрибути та способи представлення даних, які були використані при проектуванні бази даних. Налаштування цих параметрів та часових обмежень дозволяє користувачу миттєво отримувати детальну інформацію про розклад занять університету.

Було розроблено структуру бази даних на основі сховища Redis, яка містить інформацію про розклад занять та профілі користувачів. Ця структура була створена з урахуванням сучасних вимог до швидкодії інформаційних систем та специфіки збереження тимчасових даних великого обсягу.

Розроблений вебдодаток може бути корисним для освітніх закладів для оптимізації доступу до навчальних планів і розкладу, а також для зниження навантаження на основні сервери під час пікових запитів студентів.

Таким чином, завдяки виконаній роботі досягнута мета бакалаврської роботи – розробка вебдодатку підтримки функціонування месенджер-бота, який надійно зберігає дані про розклад занять університету та надає швидкий доступ до них студентам, викладачам та іншим користувачам системи.

ПЕРЕЛІК ПОСИЛАНЬ НА ДЖЕРЕЛА

1. Шнієр Б. Прикладна криптографія та безпека інформаційних систем і мережевих протоколів. Харків: Гімназія, 2019. 420 с.

2. ECMAScript Context: ECMA-404 The JSON Data Interchange Standard. URL: https://www.json.org/json-en.html (дата звернення: 18.05.2026).

3. Redis Documentation. Redis In-Memory Data Store Architecture. URL: https://redis.io/docs/ (дата звернення: 18.05.2026).

4. Офіційний сервіс розкладу занять Прикарпатського національного університету імені Василя Стефаника. URL: https://pnu.edu.ua/ (дата звернення: 18.05.2026).

5. Документація автоматизованої системи керування навчальним процесом ХНУРЕ «CIST API». URL: https://cist.nure.ua/ (дата звернення: 18.05.2026).

6. Інформаційна система «Розклад занять» Національного університету «Львівська політехніка». URL: https://lpnu.ua/ (дата звернення: 18.05.2026).

7. Студентське конструкторське бюро КПІ ім. Ігоря Сікорського. Архітектура сервісу «KPI6 Bot». URL: https://kpi.ua/ (дата звернення: 18.05.2026).

8. Aiogram Framework Documentation. Asynchronous Telegram Bot API framework for Python. URL: https://docs.aiogram.dev/ (дата звернення: 18.05.2026).

9. HTTPX Client Documentation. Next generation HTTP client for Python. URL: https://www.python-httpx.org/ (дата звернення: 18.05.2026).

10. Python Software Foundation. Python Language Reference. URL: https://docs.python.org/3/ (дата звернення: 18.05.2026).

11. Фаулер М. Архітектура корпоративних програмних додатків. Харків: Фабула, 2020. 544 с.

12. Лучіано Р. Клієнт-серверна архітектура та проектування розподілених систем за допомогою Python. Київ: Діалектика, 2022. 380 с.

13. Форда А. Довідник з веб-безпеки та захисту бездротових мережевих протоколів. Львів: Видавництво Львівської політехніки, 2021. 296 с.

14. Docker Inc. Docker Documentation. Containerization and microservices architecture. URL: https://docs.docker.com/ (дата звернення: 18.05.2026).

15. FastAPI Framework Documentation. Modern, fast (high-performance), web framework for building APIs with Python. URL: https://fastapi.tiangolo.com/ (дата звернення: 18.05.2026).

16. Pydantic Services. Data validation and settings management using Python type hinting. URL: https://docs.pydantic.dev/ (дата` звернення: 18.05.2026).

17. SQLAlchemy Authors. Database Object Relational Mapper for Python and SQL. URL: https://www.sqlalchemy.org/ (дата звернення: 18.05.2026).

18. Chodorow K. MongoDB: The Definitive Guide: Powerful and Scalable Data Storage. Sebastopol: O'Reilly Media, 2020. 514 p.

19. Git SCM. Distributed Version Control System Reference Manual. URL: https://git-scm.com/doc/ (дата звернення: 18.05.2026).

20. Офіційний веб-портал Івано-Франківського національного технічного університету нафти і газу. Інформаційна екосистема закладу освіти. URL: https://nung.edu.ua/ (дата звернення: 18.05.2026).

ДОДАТКИ

ДОДАТОК А

Код web-додатку месенджер бота сервісу розкладу занять ІФНТУНГ

cache.py

```
import asyncio
from database.redis_client import redis_manager

class TimetableCache:
    async def get(self, group: str, day_idx: int, target_date: str) -> str
| None:
        cache_key = f"timetable:{group}"
        redis = await redis_manager.get_redis()

        html = await redis.get(cache_key)
        if html:
            return html.decode('utf-8')
        return None

    async def set(self, group: str, html: str):
        cache_key = f"timetable:{group}"
        redis = await redis_manager.get_redis()
        await redis.setex(cache_key, 259200, html)

cache = TimetableCache()
```

redis_client.py

```
from redis import asyncio as aioredis
from config import REDIS_URL

class RedisManager:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
            cls._instance.redis = None
        return cls._instance

    async def get_redis(self):
        if not self.redis:
            self.redis = await aioredis.from_url(REDIS_URL)
        return self.redis

    async def close(self):
        if self.redis:
            await self.redis.close()

redis_manager = RedisManager()
```

callbacks.py

```
import logging
from datetime import datetime
from aiogram import Router, types, F
from aiogram.fsm.context import FSMContext
from states.form_states import Form
from services.browser import playwright_mgr
from services.parser import parser
from services.scheduler import ScheduleKeyboard
from database.cache import cache
from database.redis_client import redis_manager
import pytz

logger = logging.getLogger(__name__)
router = Router()

@router.message(F.text == "🔄 Оновити розклад")
async def handle_refresh_button(message: types.Message, state:
FSMContext):
    user_data = await state.get_data()
    group = user_data.get("group")

    if not group:
        await message.answer("Групу не знайдено. Введіть назву групи:")
        await state.set_state(Form.waiting_for_group)
        return

    redis = await redis_manager.get_redis()
    last_id = await redis.get(f"user:{message.from_user.id}:last_msg_id")
    if last_id:
        try:
            await message.bot.delete_message(message.chat.id,
int(last_id))
        except:
            pass
```

```
    kyiv_tz = pytz.timezone('Europe/Kyiv')
    now = datetime.now(kyiv_tz).strftime("%H:%M")
    msg = await message.answer(
        f"✅ Розклад для <b>{group}</b> оновлено!\nОберіть
день:\n<i>Оновлено о {now}</i>",
        reply_markup=ScheduleKeyboard.generate_days_keyboard(),
        parse_mode="HTML"
    )
    await redis.set(f"user:{message.from_user.id}:last_msg_id",
msg.message_id)

@router.message(F.text == "⚙ Змінити групу")
async def handle_change_group_text(message: types.Message, state:
FSMContext):
    await message.answer(
        "Введіть нову назву групи (наприклад: КІ-22-1):",
        reply_markup=types.ReplyKeyboardRemove()
    )
    await state.set_state(Form.waiting_for_group)

@router.callback_query(F.data == "change_group")
async def handle_change_group_callback(callback: types.CallbackQuery,
state: FSMContext):
    await callback.message.answer(
        "Введіть нову назву групи:",
        reply_markup=types.ReplyKeyboardRemove()
    )
    await state.set_state(Form.waiting_for_group)
    await callback.answer()

@router.callback_query(Form.group_set)
async def process_day_selection(callback: types.CallbackQuery, state:
FSMContext):
    user_data = await state.get_data()
    group = user_data.get("group")
```

```
if callback.data.startswith("day_"):
        day_idx = int(callback.data.split("_")[1])
        await callback.message.edit_text(f"□ Отримую розклад для
{group}...")

        try:
            html = await cache.get(group, day_idx, "")
            if not html:
                html = await playwright_mgr.fetch_timetable(group)
                await cache.set(group, html)

            timetable = parser.parse(html, day_idx)
            await callback.message.edit_text(
                f"▦ <b>Розклад для {group}</b>\n\n{timetable}",
                reply_markup=ScheduleKeyboard.generate_days_keyboard(),
                parse_mode="HTML"
            )
        except Exception as e:
            logger.error(f"Error: {e}")
            await callback.message.edit_text("⚠ Помилка завантаження.",

reply_markup=ScheduleKeyboard.generate_days_keyboard())
        finally:
            await callback.answer()
```

commands.py

```
from aiogram import Router, types
from aiogram.filters import Command
from aiogram.fsm.context import FSMContext
from states.form_states import Form
from services.scheduler import ScheduleKeyboard
from database.redis_client import redis_manager

router = Router()

@router.message(Command("start"))
```

```
async def cmd_start(message: types.Message, state: FSMContext):
    await message.answer(
        "👋 Введіть назву вашої групи (наприклад, КІ-22-1): ",
        reply_markup=types.ReplyKeyboardRemove()
    )
    await state.set_state(Form.waiting_for_group)

@router.message(Command("help"))
async def cmd_help(message: types.Message):
    help_text = (
        "👋 Привіт! Я — твій бот-помічник, який допоможе не пропустити пари.\n\n"
        "Щоб дізнатися свій розклад, просто введи назву своєї групи (наприклад, КІ-22-1), "
        "і я відправлю тобі розклад на вибраний день.\n\n"
        "Ось що ти можеш зробити:\n"
        "1. Введи назву своєї групи, і я надам тобі можливість вибрати день тижня.\n"
        "2. Обери день, і я покажу тобі розклад на цей день.\n"
        "3. Якщо потрібно, можеш змінити групу чи вибір дня.\n\n"
        "Ти завжди можеш звернутися до мене, і я надам актуальний розклад."
    )
    await message.answer(help_text, parse_mode="HTML")

@router.message(Command("go"))
async def cmd_go(message: types.Message, state: FSMContext):
    user_data = await state.get_data()
    group = user_data.get("group")

    if not group:
        await message.answer("ℹ️ У вас ще не обрана група. Введіть /start для початку.")
        return

    msg = await message.answer(
```

```
    f"✅ Обрана група: <b>{group}</b>\nОберіть день:",
        reply_markup=ScheduleKeyboard.generate_days_keyboard(),
        parse_mode="HTML"
    )

    redis = await redis_manager.get_redis()
    await redis.set(f"user:{message.from_user.id}:group", group)
    await           redis.set(f"user:{message.from_user.id}:last_msg_id",
msg.message_id)

    await state.set_state(Form.group_set)
```

group_handlers.py

```
import re
from aiogram import Router, types
from aiogram.fsm.context import FSMContext
from states.form_states import Form
from services.scheduler import ScheduleKeyboard
from database.redis_client import redis_manager
from config import GROUP_PATTERN

router = Router()

@router.message(Form.waiting_for_group)
async def set_group(message: types.Message, state: FSMContext):
    group = message.text.strip().upper()

    if not re.fullmatch(GROUP_PATTERN, group):
        await message.answer("❌ Неправильний формат групи. Спробуйте ще
раз.")
        return

    await state.update_data(group=group)

    await message.answer(
        f"✅ Групу <b>{group}</b> встановлено! Тепер ви можете оновлювати
```

```
розклад кнопкою знизу.",
        reply_markup=ScheduleKeyboard.get_main_reply_keyboard(),
        parse_mode="HTML"
    )

    msg = await message.answer(
        "Оберіть день тижня:",
        reply_markup=ScheduleKeyboard.generate_days_keyboard(),
        parse_mode="HTML"
    )

    redis = await redis_manager.get_redis()
    await redis.set(f"user:{message.from_user.id}:group", group)
    await           redis.set(f"user:{message.from_user.id}:last_msg_id",
msg.message_id)

    await state.set_state(Form.group_set)
```

browser.py

```
import httpx
import logging
import urllib.parse
from config import DEKANAT_URL

logger = logging.getLogger(__name__)

class TimetableDownloader:
    def __init__(self):
        self.client = httpx.AsyncClient(
            timeout=15.0,
            follow_redirects=True,
            verify=False,
            limits=httpx.Limits(max_connections=5)
        )

    async def initialize(self):
        logger.info("Мережевий завантажувач (HTTPX) активовано
```

```
(сумісність з playwright_mgr збережена)")

    async def close(self):
        await self.client.aclose()

    async def fetch_timetable(self, group: str) -> str:
        try:
            await       self.client.get("https://dekanat.nung.edu.ua/cgi-
bin/timetable.cgi?n=700")

            encoded_group = urllib.parse.quote(group.encode("cp1251"))
            encoded_sbtn                                                =
urllib.parse.quote("Переглянути".encode("cp1251"))

            payload_body                                                =
f"n=700&obj=group&group={encoded_group}&tw=0&sbtn={encoded_sbtn}"

            headers = {
                "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/120.0.0.0 Safari/537.36",
                "Content-Type": "application/x-www-form-urlencoded",
                "Referer":              "https://dekanat.nung.edu.ua/cgi-
bin/timetable.cgi?n=700",
                "Origin": "https://dekanat.nung.edu.ua",
                "Accept":
"text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/
webp,*/*;q=0.8"
            }

            response = await self.client.post(
                DEKANAT_URL,
                content=payload_body.encode("ascii"),
                headers=headers
            )

            if response.status_code == 200:
```

```
 response.encoding = 'cp1251'
                return response.text
            else:
                logger.error(f"Помилка сервера деканату: {response.status_code}")
                return ""

        except Exception as e:
            logger.error(f"Помилка при завантаженні розкладу: {e}")
            return ""

playwright_mgr = TimetableDownloader()
```

parser.py

```
import re
from datetime import datetime, timedelta
from bs4 import BeautifulSoup

class TimetableParser:
    @staticmethod
    def parse(html: str, day_idx: int) -> str:
        if not html:
            return "❌ Помилка: HTML порожній."

        soup = BeautifulSoup(html, "html.parser")
        day_names = ["Понеділок", "Вівторок", "Середа", "Четвер", "П'ятниця", "Субота"]
        target_day_name = day_names[day_idx]

        target_h4 = None
        for h4 in soup.find_all("h4"):
            if target_day_name.lower() in h4.get_text().lower():
                target_h4 = h4
                break

        if not target_h4:
```

```
            return f"❌ Розклад на <b>{target_day_name}</b> не знайдено."

        raw_title = target_h4.get_text(strip=True)
        formatted_title = re.sub(r'(\d{2}\.\d{2}\.\d{4})([А-Яа-я])', r'\1
\2', raw_title)

        rows = []
        current_element = target_h4
        while True:
            current_element = current_element.find_next(["tr", "h4"])
            if not current_element or current_element.name == "h4":
                break
            if current_element.name == "tr":
                rows.append(current_element)

        result = [f"<b>🗓 {formatted_title}</b>"]
        found_lessons = 0

        for row in rows:
            cells = row.find_all("td")
            if len(cells) < 3:
                continue

            pair_num = cells[0].get_text(strip=True)
            time_raw = cells[1].get_text(" ", strip=True).replace("\n", "
")
            time_parts = re.findall(r'\d{2}:\d{2}', time_raw)
            time_str     =     f"{time_parts[0]}-{time_parts[1]}"     if
len(time_parts) >= 2 else time_raw

            content_td = cells[2]

            for div in content_td.find_all("div", class_="link"):
                div.decompose()
```

```
   is_remote = bool(content_td.find("span", class_="remote_work") or
                            content_td.find("img", src=lambda x: x and
"home-16" in x))

            lines = [l.strip() for l in
content_td.get_text("\n").split("\n")
                     if l.strip() and l.strip().lower() != "дистанційно"]

            if not lines:
                result.append(f"<b>{pair_num} пара ({time_str})</b>\n
<i>Занять немає</i>")
                continue

            found_lessons += 1
            links = []
            for a in content_td.find_all("a", href=True):
                url = a["href"]
                label = "Meet" if "meet" in url else "Moodle" if "dn.nung"
in url else "🔗"
                links.append(f"<a href='{url}'>{label}</a>")

            title = lines[0]
            icon = "🏫"
            if "(Л)" in title: icon = "📖 Лекція |"; title =
title.replace("(Л)", "")
            elif "(Пр)" in title: icon = "🔧 Практ. |"; title =
title.replace("(Пр)", "")
            elif "(Лаб)" in title: icon = "🧪 Лаб. |"; title =
title.replace("(Лаб)", "")
            elif "Військова" in title: icon = "🎖"; title = "Військова
підготовка"

            details = []
            for line in lines[1:]:
```

```
   if "ауд." in line: details.append(f"📍 {line}")
                elif any(x in line for x in ["група", "підгр", "Збірна",
"Потік"]): details.append(f"👥 {line}")
                else: details.append(f"🤷📍 {line}")

            remote_label = "💻 <b>ДИСТАНЦІЙНО</b>\n" if is_remote else ""
            formatted_pair = f"<b>{pair_num} пара
({time_str})</b>\n{remote_label}{icon} <b>{title.strip()}</b>"

            if details: formatted_pair += "\n" + "\n".join(details)
            if links: formatted_pair += "\n🔗 " + " | ".join(links)

            result.append(formatted_pair)

        return "\n\n━━━━━━\n\n".join(result)

parser = TimetableParser()
```

scheduler.py

```
from datetime import datetime, timedelta
from aiogram.types import InlineKeyboardMarkup, InlineKeyboardButton,
ReplyKeyboardMarkup, KeyboardButton
import pytz

class ScheduleKeyboard:
    @staticmethod
    def get_main_reply_keyboard():
        return ReplyKeyboardMarkup(
            keyboard=[
                [KeyboardButton(text="🔄 Оновити розклад")],
                [KeyboardButton(text="⚙ Змінити групу")]
            ],
            resize_keyboard=True,
            input_field_placeholder="Оберіть дію..."
        )
```

```
    @staticmethod
    def generate_days_keyboard():
        kyiv_tz = pytz.timezone('Europe/Kyiv')
        today = datetime.now(kyiv_tz)
        days = ["Понеділок", "Вівторок", "Середа", "Четвер", "П'ятниця"]

        buttons = []
        for i, day in enumerate(days):
            days_ahead = i - today.weekday()
            if days_ahead < 0:
                days_ahead += 7
            target_date = today + timedelta(days=days_ahead)
            date_str = target_date.strftime("%d.%m")

            buttons.append(
                InlineKeyboardButton(
                    text=f"{day} {date_str}",
                    callback_data=f"day_{i}"
                )
            )

        return InlineKeyboardMarkup(inline_keyboard=[
            buttons[:2],
            buttons[2:4],
            [buttons[4]]
        ])
```

bot.py

```
import asyncio
import logging
from aiogram import Bot, Dispatcher
from aiogram.fsm.storage.redis import RedisStorage
from config import BOT_TOKEN, REDIS_URL, LOGGING_CONFIG
from database.redis_client import redis_manager
from handlers import commands, group_handlers, callbacks
from services.browser import playwright_mgr
```

```
logging.basicConfig(**LOGGING_CONFIG)
logger = logging.getLogger(__name__)

async def on_startup():
    logger.info("Запуск бота...")
    await playwright_mgr.initialize()
    logger.info("Playwright ініціалізовано")
    logger.info("Бот готовий до роботи!")

async def on_shutdown():
    logger.info("Зупинка бота...")
    await playwright_mgr.close()
    await redis_manager.close()
    logger.info("Ресурси звільнено")

async def main():
    redis = await redis_manager.get_redis()
    storage = RedisStorage(redis)

    bot = Bot(token=BOT_TOKEN)
    dp = Dispatcher(storage=storage)

    dp.include_router(commands.router)
    dp.include_router(group_handlers.router)
    dp.include_router(callbacks.router)

    dp.startup.register(on_startup)
    dp.shutdown.register(on_shutdown)

    try:
        await dp.start_polling(bot)
    finally:
        await bot.session.close()

if __name__ == "__main__":
    asyncio.run(main())
```

config.py

```
import os
from dotenv import load_dotenv

load_dotenv()

BOT_TOKEN = os.getenv("BOT_TOKEN")
REDIS_URL = os.getenv("REDIS_URL", "redis://localhost:6379/0")
DEKANAT_URL = os.getenv("DEKANAT_URL", "https://dekanat.nung.edu.ua/cgi-
bin/timetable.cgi?n=700")

CACHE_TTL = 3 * 86400
GROUP_PATTERN = r'^[A-ZA-ЯҐЄІЇ]{1,5}-\d{1,3}-\d[A-ZA-ЯҐЄІЇ]{0,3}$'

LOGGING_CONFIG = {
    'level': os.getenv('LOG_LEVEL', 'INFO'),
    'format': '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
}
```

БІБЛІОГРАФІЧНА ДОВІДКА

Тема бакалаврської роботи: **Розробка web-додатку підтримки функціонування месенджер-бота сервісу розкладу занять ІФНТУНГ на базі Redis**

Обсяг пояснювальної записки 54 аркушів:

5 таблиць;

23 рисунка;

1 додаток.

Дата завершення роботи: 11 червня 2026р.

Підпис студента- _____Гулик Р.О..