ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНФОРМАЦІЙНИХ СИСТЕМ ТА ТЕХНОЛОГІЙ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «ПРОКСІ RESTFUL-СЕРВІС МОНІТОРИНГУ РОЗКЛАДУ ЗАНЯТЬ ДЛЯ СТУДЕНТІВ УНІВЕРСИТЕТІВ»

на здобуття освітнього ступеня бакалавра
зі спеціальності 126 Інформаційні системи та технології
освітньо-професійної програми Інформаційні системи та технології

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

_____ *Андрій ДАНИЛЕНКО*

Виконав:
здобувач вищої освіти
група ІСД-42 — Андрій ДАНИЛЕНКО

Керівник:
науковий ступінь, вчене звання — Оксана ТКАЛЕНКО, *к.т.н., доцент*

Рецензент:
науковий ступінь, вчене звання — _____
Ім’я, ПРІЗВИЩЕ

Київ – 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут Інформаційних технологій

Кафедра Інформаційних систем та технологій

Ступінь вищої освіти Бакалавр

Спеціальність Інформаційні системи та технології

Освітньо-професійна програма Інформаційні системи та технології

ЗАТВЕРДЖУЮ
Завідувач кафедрою ІСТ

_____Каміла СТОРЧАК
«____» _____20___ р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ
Даниленко Андрію Євгенійовичу
(прізвище, ім'я, по батькові здобувача)

1.Тема кваліфікаційної роботи: «проксі restful-сервіс моніторингу розкладу занять для студентів університетів»

керівник кваліфікаційної роботи Оксана ТКАЛЕНКО, к.т.н., доцент
(ім'я, ПРІЗВИЩЕ, науковий ступінь, вчене звання)

затверджені наказом вищого навчального закладу від «20» лютого 2026 року № 51.

2. Строк подання кваліфікаційної роботи: 01 червня 2026 року.

3. Вихідні дані до кваліфікаційної роботи: мова програмування Java, фреймворк Spring, Hibernate, бібліотеки Jackson, liquiBase, Apache Http.
науково-технічна література з питань, пов'язаних з WEB-технологіями.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1) Аналіз предметної області, існуючих рішень та вибір технологічної платформи для автоматизації моніторингу розкладу.

2) Проєктування моделі даних, збір функціональних та нефункціональних вимог до програмного продукту.
3) Архітектурне проєктування системи, розробка UML-моделей та підсистеми багаторівневого кешування.
4) Програмна реалізація серверної частини Telegram-бота, алгоритмів обробки запитів та взаємодії з API месенджера.

5. Перелік ілюстративного матеріалу: *презентація*

6. Дата видачі завдання: 20 лютого 2026 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз наявної науково-технічної літератури	23.02 – 27.02.26	
2	Аналіз предметної області та існуючих рішень для автоматизації моніторингу навчального розкладу	02.03 – 09.03.26	
3	Формування функціональних вимог та проєктування внутрішньої моделі даних	10.03 – 24.03.26	
4	Проєктування базової архітектури системи та створення алгоритмів парсингу JSON-даних	25.03 – 07.04.26	
5	Розробка сервісного шару та підсистеми багаторівневого кешування розкладів (рантайм, БД, файли)	08.04 – 23.04.26	
6	Імплементація бізнес-логіки та маршрутизатора команд на базі патернів «Стратегія» та «Фабрика»	24.04 – 04.05.26	
7	Оформлення роботи: вступ, висновки, реферат	05.05 – 15.05.26	
8	Розробка демонстраційних матеріалів	18.05 – 27.05.26	

Здобувач вищої освіти _____ Андрій ДАНИЛЕНКО
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

Керівник
кваліфікаційної роботи _____ Оксана ТКАЛЕНКО
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 64 стор., 48 рис., 3 табл., 22 джерела.

Мета роботи – розробка та впровадження автоматизованої інформаційної системи моніторингу навчального розкладу у вигляді Telegram-бота для підвищення ефективності та зручності доступу студентів до освітньої інформації.

Об'єкт дослідження – процес автоматизації отримання, обробки та розповсюдження інформації щодо розкладу навчальних занять у закладах вищої освіти.

Предмет дослідження – методи, архітектурні патерни та програмні засоби розробки надійних кросплатформних Telegram-ботів із використанням екосистеми мови програмування Java та фреймворку Spring.

Короткий зміст роботи: Досліджено предметну область, існуючі рішення для моніторингу навчальних розкладів та виявлено їхні недоліки при використанні з мобільних пристроїв. Здійснено проєктування масштабованої архітектури серверного додатка з використанням патернів «Стратегія» та «Фабрика». Обґрунтовано вибір технологічного стеку на базі Java, Spring та бази даних PostgreSQL. Розроблено підсистему парсингу та нормалізації JSON-документів із даними розкладу. Імплементовано відмовостійкий багаторівневий механізм кешування у оперативній пам’яті, базі даних та файлах. Створено користувацький інтерфейс на базі Telegram API із використанням inline-клавіатур.

КЛЮЧОВІ СЛОВА: TELEGRAM-БОТ, JAVA, SPRING, SPRING BOOT, РОЗКЛАД ЗАНЯТЬ, ПАРСИНГ ДАНИХ, НОРМАЛІЗАЦІЯ ДАНИХ, КЕШУВАННЯ, АРХІТЕКТУРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, ІНФОРМАЦІЙНА СИСТЕМА, API.

ЗМІСТ

ВСТУП

Актуальність теми. Швидка цифровізація усіх сфер життя та, особливо, освіти та перехід до змішаних форматів навчання супроводжується зростанням вимог до оперативності та зручності отримання академічної інформації. Сучасний освітній процес вимагає від студентів постійного контролю за змінами у розкладі занять, їх перенесенням та форматами проведення (онлайн або офлайн). За таких умов ефективна організація навчального часу неможлива без наявності надійного, мобільного та інтуїтивно зрозумілого інструменту доступу до актуальної інформації.

Традиційні методи поширення інформації, такі як статичні веб-портали університету або розсилка файлів у форматах електронних таблиць, встрачають свою ефективність. Вони часто не є адаптованими для зручного перегляду з мобильних пристроїв та вимагають від здобувача освіти постійної проактивної дії (ручного оновлення даних). Крім того, централізовані веб-сервери нерідко страждають від пікових навантажень у ранкові години, що призводить до тимчасової недоступності критично важливої інформації. Це зумовлює необхідність розробки нових підходів до автоматизації отримання даних, здатних працювати в умовах обмеженої доступності зовнішніх джерел даних. Перспективним напрямком вирішення цієї задачі є інтеграція сервісів отримання розкладу у популярні платформи-месенджери, якими студенти користуються щодня.

Використання чат-ботів із застосуванням сучасних технологічних стеків (Java та Spring) та алгоритмів багаторівневого кешування відкриває можливості для кардинального підвищення швидкості доступу до навчального розкладу. Це дозволяє забезпечити гарантовану видачу даних навіть при недоступності віддаленого сервера університету, оптимізувати мережевий трафік, запровадити персоналізовані автоматичні сповіщення та позбавити користувачів потреби

здійснювати ручний пошук. Крім того, такі підходи є надзвичайно важливою складовою побудови сучасного цифрового середовища закладів освіти.

У зв’язку з цим розробка та дослідження автоматизованої системи моніторингу навчального розкладу на базі архітектури Telegram-бота є надзвичайно актуальним науково-практичним завданням. Вирішення цього завдання сприятиме підвищенню якості та ефективності інформаційного супроводу освітнього процесу, повністю відповідаючи сучасним тенденціям мобільної цифровізації суспільства.

Метою роботи є розробка та впровадження автоматизованої інформаційної системи моніторингу навчального розкладу у вигляді Telegram-бота на основі сучасних архітектурних патернів для підвищення ефективності та надійності доступу для студентів навчальних закладів.

Для досягнення поставленої мети необхідно виконати наступні завдання:

- дослідити предметну область, існуючі програмні рішення та особливості автоматизації моніторингу розкладу занять;
- обґрунтувати вибір технологічного стека (Java, Spring, PostgreSQL) та архітектурних патернів проєктування для забезпечення масштабованості системи;
- розробити архітектуру системи, UML-моделі внутрішньої структури даних та алгоритми парсингу JSON-документів;
- реалізувати та дослідити відмовостійкий механізм багаторівневого кешування даних (рантайм кеш, БД, файлове кешування).
- виконати програмну реалізацію серверної частини бота, інтегрувати її з Telegram API.

Об'єкт дослідження – процес автоматизації отримання та розповсюдження інформації про розклад навчальних занять у закладах освіти.

Предметом дослідження є методи, архітектурні патерни та програмні засоби розробки відмовостійких кросплатформних Telegram-ботів з використанням екосистеми Java та Spring.

Методи дослідження: аналіз літературних джерел і огляд існуючих рішень; об’єктно-орієнтоване проєктування; системний аналіз; методи розробки програмного забезпечення за принципами SOLID

Наукова новизна отриманих результатів: запропоновано архітектурний підхід до побудови Telegram-бота на базі патернів «Стратегія» та «Фабрика» із впровадженням механізму динамічної маршрутизації станів та багаторівневого кешування, що забезпечує стабільну роботу системи в умовах нестабільного доступу до зовнішніх джерел інформації.

Практична значущість отриманих результатів полягає у можливості застосування розробленої системи у закладах вищої освіти для: автоматичної ідентифікації та парсингу даних розкладу у реальному часі; забезпечення миттєвого доступу до навчальних планів з мобільних пристроїв; гарантованої видачі інформації в режимі без доступу до зовнішнього джерела завдяки кешуванню; автоматизація вечірніх сповіщень студентів.

Апробація результатів та публікації:

Даниленко А. Є. «Використання штучного інтелекту для моніторингу мережі» Тези доповіді на II Всеукраїнській науково-технічній конференції «Технологічні горизонти: дослідження та застосування інформаційних технологій для технологічного прогресу України і світу». – Київ, 18 листопада 2024 року.

Даниленко А. Є. «Тенденції та технології сучасної розробки програмного забезпечення» Тези доповіді на III Всеукраїнській науково-технічній конференції «Технологічні горизонти: дослідження та застосування інформаційних технологій для технологічного прогресу України і світу». – Київ, 18 листопада 2025 року.

РОЗДІЛ 1. АНАЛІЗ ТЕХНОЛОГІЧНИХ ОСНОВ ТА ІСНУЮЧИХ РІШЕНЬ ДЛЯ АВТОМАТИЗАЦІЇ РОЗКЛАДУ НАВЧАНЬ

1.1 Огляд сучасних систем управління розкладом у закладах освіти

У сучасних умовах трансформації освітнього процесу у світі та Україні питання ефективного управління та розповсюдження навчальних розкладів стає дедалі важливішою. Розклад є одним з ключових інформаційних ресурсів навчальних закладів, оскільки він має вплив на організацію навчального процесу, розподіл часу студентів та викладачів, а також на ефективність освітньої діяльності в цілому.

Традиційні підходи до формування та розповсюдження розкладів, зокрема у фізичному вигляді (друкованих таблиць) або електронних статичних файлів, втрачають свою актуальність. Цю ситуацію погіршила війна, через яку динаміка змін у розкладі значно виросла, а також змінилися вимоги до зручності доступу до інформації.

Системи моніторингу електронних розкладів забезпечують доступ до централізовано збережених даних, перевірку консистетності даних, а також інформування студентів у разі зміни розкладу. Існує багато прикладів впровадження таких систем, які ми розглянемо. З основних видів систем електронних розкладів ми можемо виділити дві групи: веб-додатки та додатки, що встановлюються на пристрій користувача.

У цьому розділі будуть розглянуті основні типи систем моніторингу розкладів, що найчастіше використовуються у закладах освіти. Також буде розглянуто форми представлення та збереження розкладів для зручного зберігання та транспортування в межах мережі Інтернет.

Огляд онлайн-порталу із розкладом Skedy

Одним з найбільш поширених підходів до автоматизації задачі розповсюдження навчальних розкладів є використання веб-додатків. Веб-розклади зазвичай реалізуються у вигляді one-page сайтів, які надають доступ до актуальної інформації за допомогою браузеру.

Головним прикладом імплементації веб-додатку для моніторингу розкладів університету є Skedy [1]. Дружній до користувача інтерфейс дозволяє переглядати розклад шляхом вибору навчального закладу, факультету, курсу та групи студенту. Також він надає можливість зручної навігації та структуроване представлення інформації, що робить її зрозумілою для широкої аудиторії користувачів (рис. 1.1).

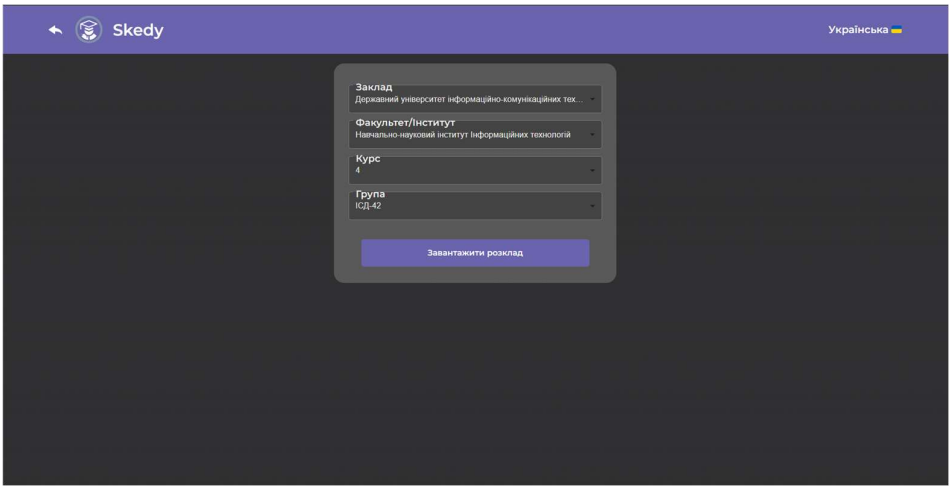


Рис. 1.1 Демонстрація вибору даних про групу студента

Веб-додаток Skedy [1] використовує структуру відображення розкладів у вигляді плиток. Воно побудоване на основі блоків інформації, де день представлений у вигляді окремого блоку з детальною інформацією про заняття (рис. 1.2). Також є можливість дивитися щотижневий розклад з функцією переходу між минулими, поточним та майбутніми тижнями (рис. 1.3).

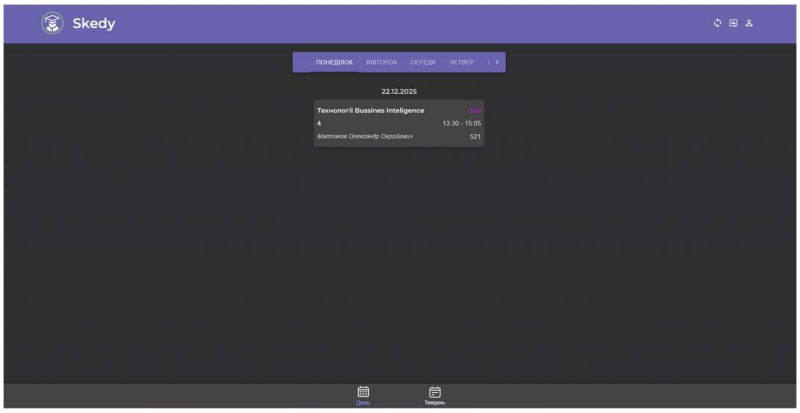


Рис. 1.2 Демонстрація плиткової структури щоденного розкладу

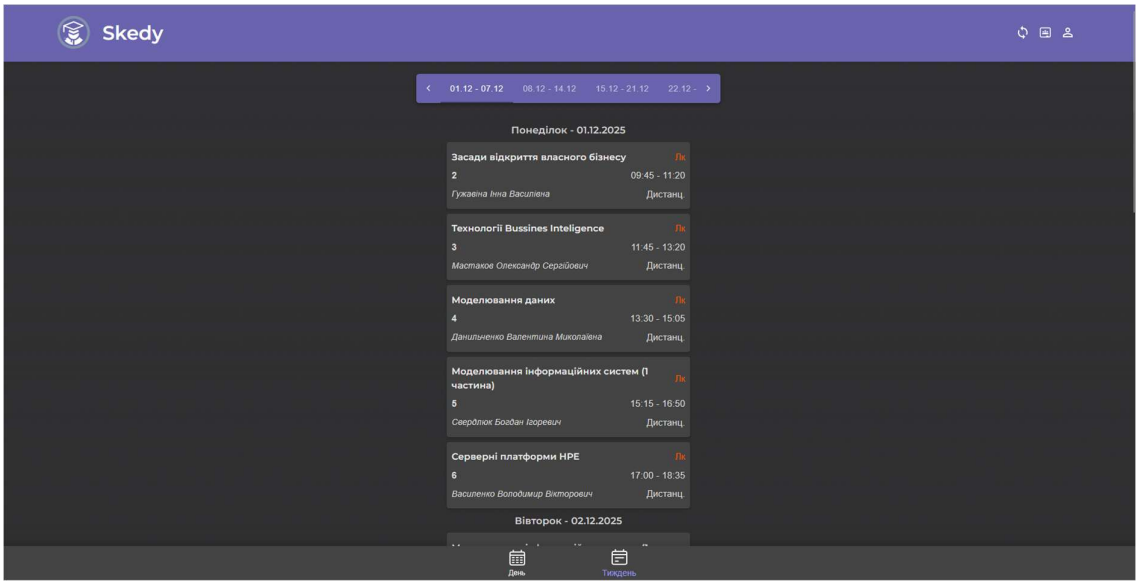


Рис. 1.3 Демонстрація плиткової структури щотижневого розкладу

Веб-додаток Skedy [1] безпосередньо інтегрований у внутрішні бази даних університетів або працюють з xlsx-таблицями. Це дозволяє автоматизувати процес оновлення інформації, а також динамічно реагувати на зміни у розкладі.

Разом із великою кількістю переваг, веб-розклади мають і певні недоліки. Основним із них є обмежена зручність використання на мобільних пристроях. Незважаючи на наявність адаптивного дизайну, веб-додатки зазвичай поступаються нативним застосункам за швидкістю роботи, зручністю інтерфейсу та взаємодії в умовах щоденного використання.

Аналіз технології чат-ботів, як основи для інформаційної системи

З метою підвищення зручності використання систем моніторингу розкладу все більшого розповсюдження набувають альтернативні підходи, зокрема мобільні додатки та чат-боти у месенджерах. На відміну від нативних додатків та веб-застосунків, чат-боти не потребують встановлення окремого програмного забезпечення, що знижує поріг входу для цільової аудиторії.

Однією з ключових переваг використання ботів у месенджерах є відсутність потреби займати додаткове місце у пам’яті мобільного пристрою, - ані постійної, ані оперативної. Навіть сучасні смартфони мають обмежений обсяг вільної пам’яті, тому студенти не завжди готові встановлювати окремі додатки для виконання вузькоспеціалізованих задач, таких як моніторинг розкладу навчального закладу. У випадку чат-бота достатньо мати встановлений месенджер, який студент уже використовує для повсякденного спілкування.

Крім означених вище переваг, чат-боти мають низку додаткових переваг:

1. Кросплатформеність - бот одночасно доступний на пристроях із різними ОС, на яких є Telegram (Windows, Android, IOS, MacOS, тощо), а усі складнощі підтримки окремих систем повністю лягають на розробника Telegram.
2. Відсутність оновлення на стороні користувача - усі зміни та покращення архітектури відбуваються на сервері, що гарантує миттєвий доступ до останньої версії сервісу за допомогою CI/CD інтеграції.
3. Швидкий доступ до інформації - взаємодія відбувається в окремому чат-інтерфейсі, який мінімізує кількість необхідних дій для отримання інформації.
4. Підтримка push-сповіщень - бот може автоматично інформувати про зміни у розкладі або будь-які інші важливі та просто цікаві новини.
5. Низькі вимоги до апаратних ресурсів - бот не створює додаткове навантаження на залізо пристрою. Усю роботу на себе бере віддалений сервер.

Проте нативні мобільні застосунки також мають певні переваги, наприклад більш багаті можливості графічного інтерфейсу та потенційну можливість працювати в офлайн-режимі. Однак для задачі оперативного доступу до інформації щодо розкладів фунціоналу чат-ботів у месенджерах цілком достатньо, а у багатьох

випадках - є більш доцільною з точки зору ефективності та зручності використання, особливо якщо додати стратегії кешування.

Таким чином, можна зазначити, що чат-боти можуть розглядатися як найбільш раціональне рішення для автоматизації перегляду навчальних розкладів, оскільки поєднують найважливіші для такої роботи характеристики: простоту використання, мінімальні вимоги до ресурсів системи та високий рівень доступності для широкої аудиторії.

Проте незалежно від обраного підходу реалізації системи моніторингу навчальних розкладів, однією з ключових проблем є забезпечення актуальності та доступності даних. Часті зміни у навчальному процесі, перенесення занять або зміни викладачів та аудиторій потребують своєчасного оновлення даних.

Аналіз існуючих рішень показує необхідність створення гнучких, масштабованих та, головне, зручних рішень автоматизації моніторингу розкладів, які поєднують переваги різних підходів та забезпечують високий рівень доступності для користувачів.

1.2 Аналіз форм представлення та зберігання розкладу

Існує багато форм представлення даних, які можуть бути використані для збереження і відображення розкладу навчальних дисциплін. Вибір конкретного формату безпосередньо впливає на зручність та швидкість обробки інформації, можливість інтеграції із зовнішніми системами, а відповідно й на складність у реалізації.

В контексті автоматизації отримання та представлення розкладу необхідно враховувати два аспекти:

1. Зручність для сприйняття інформації людиною;
2. Зручність для машинної обробки.

Одним з найпростіших форматів представлення розкладів є HTML-таблиці. Вони дозволяють відображати структуровані дані у звичному для користувача вигляді: у форматі рядків та стовпців, де кожен рядок може відповідати часовому

інтервалу, а стовпець - дню тижня. Прикладом такого відображення є розклад НАУ [2] (рис.1.4).

Розклад групи Б-G9-25-1-МК

Тиждень 1

	Понеділок	Вівторок	Середа	Четвер	П'ятниця	Субота	Неділя
1 08:00-09:35		Б-G12-25-1-ЛВ Б-G9-25-1-МК Бд-G12-25-1-ЛВ **Основи авіації** Практичне ауд. 11.230 *Каран Євген Валентинович*	Б-G11-25-1-ГУ Б-G7-25-1-РК Б-G9-25-1-МК Бд-G7-25-1-РК **Інтенсивний курс англійської мови** Практичне ауд. 1.337 *Ладогубець Наталія Віталіївна*				
2 09:50-11:25	**Матеріалознавство** Лабораторна ауд. 2.312 *Федорчук Світлана Володимирівна*	Б-G9-25-1-МК Б-J6-25-1-ЛЕ **Академічна та публічна комунікація українською мовою** Практичне ауд. 9.201 *Онуфрійчук Ганна Іванівна*	Б-G9-25-1-МК Б-J6-25-1-ЛЕ **Академічна та публічна комунікація українською мовою** Практичне ауд. 9.204 *Відоменко Аліна Романівна*	**Матеріалознавство** Лекція ауд. 2.312 *Федорчук Світлана Володимирівна*			
3 11:40-13:15	Б-G12-25-1-ЛВ Б-G9-25-1-МК Бд-G12-25-1-ЛВ **Теоретична механіка** Лекція ауд. 1.335 *Галембієвський Григорій Григорійович*	Б-G11-25-1-ГУ Б-G7-25-1-РК Б-G9-25-1-МК Бд-G7-25-1-РК **Інтенсивний курс англійської мови** Практичне ауд. 1.337 *Ладогубець Наталія Віталіївна*	**Вища математика** Практичне ауд. 10.212 *Горідько Руслана Володимирівна*	**Основи мехатроніки** Лабораторна ауд. 2.408 *Баштa Олександр Васильович*	**Основи мехатроніки** Лекція ауд. 2.406 *Баштa Олександр Васильович*		
4 13:30-15:05					**Інженерна та комп'ютерна графіка** Лабораторна ауд. 3.516 *Квач Юлія Миколаївна*		
5 15:20-16:55							
6 17:10-18:45							
7 19:00-20:35							

Рис. 1.4 Приклад HTML-таблиці із розкладом університету

HTML-таблиці мають низку переваг:

1. Наочність представлення інформації;
2. Простота публікації у веб-середовищі;
3. Універсальна підтримка всіма сучасними браузерами.

Проте незважаючи на зручність для людини, HTML-таблиці мають складності, пов’язані із автоматизацією обробки. HTML-код може дуже складну структуру (це може бути або надмірна вкладеність, або використання стилів, класів та скриптів). Але головна проблема HTML-таблиць полягає в тому, що часто візуальне представлення не відповідає логічній структурі, що призводить до наступних проблем:

1. Відсутність чіткої структури;
2. Залежність стратегії парсингу від конкретної верстки сторінки;
3. Висока чутливість до змін у HTML-коді сторінки;

4. Необхідність у використанні досить важких та дорогих у плані продукивності інструментів (CSS-селектори, Xpath або DOM-парсери).

Навіть дрібні зміни у коді сторінки (наприклад зміна назви класу таблиці) може призвести до некоректної роботи алгоритмів обробки. Таким чином HTML є зручним форматом для сприйняття людиною, але не є оптимальним з точки зору обробки автоматизованою системою.

Більш сучасним підходом до зберігання та обміну інформацією є JSON (JavaScript Object Notation) [3]. В останні роки цей формат де-факто став стандартом для зберігання та обміном даних не тільки у веб-середовищі (наприклад для комунікації REST-сервіси майже завжди використовують JSON заміть XML), а й у автономних додатках.

Основною перевагою JSON є його структурованість та логічна організація даних у вигляді об’єктів та масивів. Наприклад у JSON розклад може бути представлений, як масив дат, кожна з яких має масив занять із потрібними параметрами, наприклад: назва дисципліни, кабінет, викладач, час проведення, тощо. На рис. 5 показаний приклад об’єкта, що збережено у форматі JSON.

Рис. 1.5 Приклад об'єкта JSON

Можна виділити основні переваги використання JSON для обміну інформацією у додатку:

1. Чітка структура даних;
2. Зручність серіалізації та десеріалізації;
3. Легка інтеграція із веб- або мобільними застосунками через REST;
4. Незалежність від візуального представлення;
5. Простота передачі через HTTP-запити.

Незважаючи на те, що JSON є доволі зручним для сприйняття людиною, аналіз складних структур може вимагати додаткових інструментів.

Проте однією із з ключових проблем автоматизації розкладів є відсутність єдиного стандарту представлення даних у закладах освіти. Університети можуть використовувати різні способи для розповсюдження розкладів: статичні HTML-сторінки, Excel-таблиці, PDF-документи або навіть зображення.

Відсутність єдиного стандарту ускладнює процес автоматичного збору та парсингу інформації. Серед основних труднощів можна виділити наступні:

1. Відсутність стабільної структури документу;
2. Використання візуальних елементів замість семантичних (Дуже розповсюджене у PDF-таблицях, через що їх обробка може стати дуже неприємним заняттям);
3. Об’єднані комірки таблиць;
4. Різний формат представлення дат та часу.

У випадку HTML-документів необхідно реалізовувати алгоритми, які будуть враховувати структуру DOM-дерева [4], а для PDF-файлів дуже часто доводиться використовувати технологію OCR [5], яка дозволяє зчитувати дані з картинки, але це може призвести до втрати форматування або повністю порушити консистентність джерела даних (деякі дані з файлу можуть просто не зчитатися).

У зв’язку з цим доцільним є створення проміжного шару обробки даних (Наприклад, це можуть бути парсинг-делегати, які реалізують різні стратегії в залежності від джерела інформації), які перетворюють початкову інформацію до

стандартизованого вигляду (Наприклад, JSON), що надалі будуть використовуватися ботом або іншою інформаційною системою.

Підводячи підсумки підрозділу 1.2, аналіз різних форм представлення даних показує, що хоча HTML та інші формати зручні для візуального відображення, для автоматизації та інтеграції набагато більш ефективним є використання структурованих типів представлення інформації із чітко визначеною логікою організації даних (наприклад, JSON).

1.3 Технології та методи для побудови чат-ботів

Розвиток глобальної мережі Інтернет та поява персональних мобільних пристроїв посприяли появі універсальних платформ для спілкування та обміну інформацією, а відповідно й нових програмних систем - чат-ботів. Чат-боти у Telegram - це спеціалізовані програми, що можуть взаємодіяти із користувачем за допомогою текстових повідомлень, кнопок та команд (Проте важливо сказати, що останні покращення API для розробників додали концепцію міні-додатків (Telegram mini-apps), яка дозволяє втілити вже повноцінний графічний інтерфейс, використовуючи node.js як програмну платформу). Завдяки багатому та, найголовніше, відкритому API Telegram надає широкі можливості для побудови розумних, швидких та безпечних інформаційних та сервісних систем.

У контексті автоматизації моніторингу навчального розкладу чат-бот буде виступати, як проміжна ланка між користувачем та мікросервісами, які у свою чергу, працюють із базами даних університетів, займаються автоматичних збором та завантаженням, а також парсингом даних.

Telegram Bot API [6] - це набір HTTP методів, який дозволяє розробникам створювати ботів та імплементувати будь-який необхідний функціонал. Взаємодія між ботом та серверами Telegram відбувається за принципом клієнт-сервер через REST-запити, інформація у яких передається у форматі JSON.

Основної сутністю взаємодії між клієнтом та сервером є Update[7]. Через цей об’єкт передається інформація про подію: текстове повідомлення, натискання

кнопки, інформація про користувача або групу. Сервер бота отримує ці оновлення, аналізує їх вміст та формує необхідну відповідь.

Існує два механізми отримання оновлень (рис. 1.5):

1) Long Polling - бот раз у деякий час відправляє запити на сервера, щоб отримати повідомлення;

2) WebHook - працює за принципом підписки (Telegram автоматично відсилає оновлення усім підписникам серверної адреси).

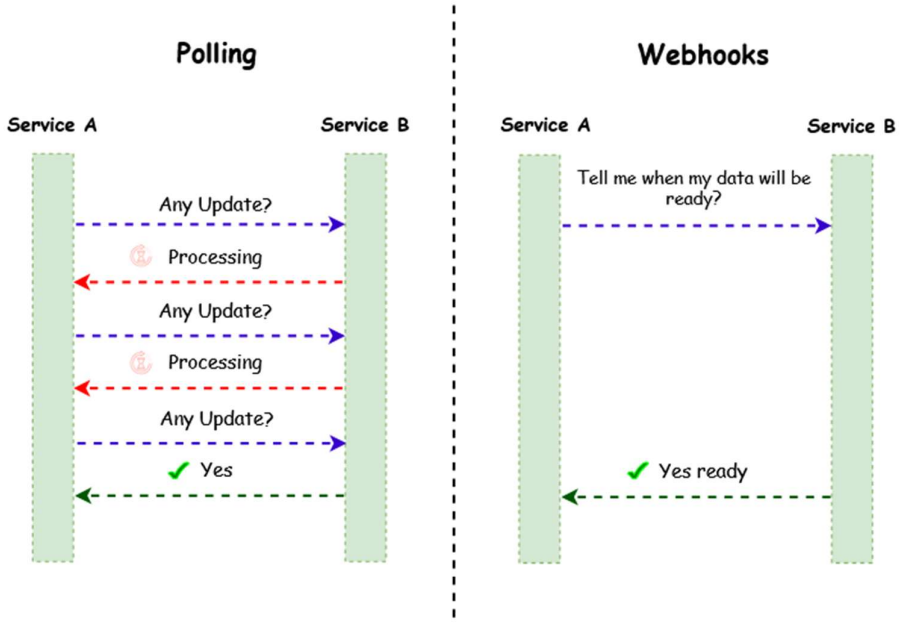


Рис. 1.6 Порівняння механізмів оповіщення [8]

З точки зору продуктивності, швидкості та захищеності цей механізм, безумовно, є більш ефективним. Проте його використання вимагає наявності HTTPS-сервера.

Проте API, який надає Telegram, має певні обмеження, наприклад:

1) Ліміти запитів (Rate limits) - обмеження на кількість запитів за одиницю часу;

2) Максимальну довжину повідомлення (На даний момент вона складає 4096 символи, що може бути недостатнім в деяких випадках, які ми розглянемо у наступних розділах);

3) Обмеження щодо розміру файлів.

Розуміння цих обмежень дозволить нам спроектувати стабільну та ефективну систему, а в деяких випадках й обійти дані обмеження.

Як вже було сказано, Telegram-боти підтримують різні формати взаємодії між користувачем та ботом, що дозволяє адаптувати інтерфейс до конкретної задачі.

Найбільш поширеною є командна модель, у якій користувач вводить команди (наприклад, /start, /help, /menu, тощо). Така модель є найпростішою з можливих, проте накладає на користувача потребу у запам’ятовуванні команд (Що, насправді, є недоліком у UX).

Більш зручною моделлю є використання кнопкових меню (або Reply Keyboard), яке дозволяє відображати список доступних дій без необхідності писати команди самотужки. Це підвищує зручність для користувача, а також зменшує кількість можливих помилок введення.

Проте найзручнішим із наведених моделей взаємодії є inline-кнопки. Такі кнопки прикріплюються безпосередньо до повідомлення. Вони дозволяють реалізовувати динамічну навігацію, зберігати контекст розмови, а також не дає користувачу можливості вписати неправильні дані. Саме такий механізм краще усього використовувати для вибору дня, тижня для отримання розкладу, або факультету та групи під час реєстрації у боті.

Для більш складних сценаріїв існує станова модель (Finite-state machine), яка дозволяє зберігати поточний стан взаємодії із користувачем. Це надає змогу створювати сценарні пайплайни без потреби скидувати стан після кожного завершення роботи.

Підводячи підсумок цього підрозділу можна сказати, що правильний вибір взаємодії суттєво впливає на зручність користування системою, а також її відмовостійкість.

Архітектура бота визначає структуру програмного коду, а також зони відповідальності компонентів та принципи взаємодії із зовнішніми сервісами.

Найпростішим із підходів є монолітна архітектура. У такій архітектурі усі функціональні модулі реалізовані у межах одного додатку. Такий підхід є доцільним

тільки при умові, що застосунок є невеликим та немає великої кількості функціоналу.

Більш складні системи можуть потребувати використання мікросервісної архітектури, який передбачає розподіл системи на окремі модулі та їх незалежність один від одного. Такі системи є більш надійними, оскільки відмова одного сервісу не буде впливати на інші. Проте вибір такої архітектури накладає на розробника необхідність використання додаткового сервісу обміном даних (Наприклад, Apache Kafka або RabbitMQ).

У випадку систем автоматизації моніторингу розкладів є доцільним використання мікросервісної архітектури, де сервіси реєстрації, мережевих запитів, оповіщення та кешування є незалежними від один одного.

Telegram-боти можуть бути реалізовані на базі багатьох мов програмування, найбільш популярними з яких є Java, Python або JavaScript (Node.js). На рис. 1.7 (діаграма) можна побачити розподілення по популярності мов програмування для написання Telegram-ботів.

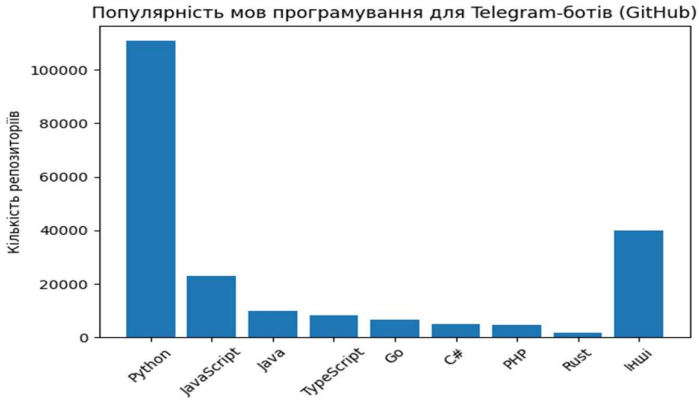


Рис. 1.7 Розподілення кількості ботів по мовах програмування GitHub [9]

Як можна побачити мова програмування Python посідає перше місце по кількості репозиторіїв із ключовими словами “Telegram bot”. Проте вибір мови залежить від вимог продуктивності, потреб у безпеці, а також досвіду розробника.

Для кожної мови існують спеціалізовані бібліотеки, які імплементують функціонал Telegram API. Використання готових бібліотек полегшує задачу розробки рішень, зменшуючи загальний обсяг коду та кількість помилок.

Найбільш популярними бібліотеками для побудови ботів є:

1. Python: python-telegram-bot, aiogram, pyTelegramBotAPI;
2. JavaScript: telegraf, gramjs;
3. Java: TelegramBots;
4. C#: Telegram.Bot.

Наш Telegram-бот буде використовувати мову програмування Java та екосистему Spring Framework[10]. Вибір пов’язаний із швидкістю компільованих мов програмування разом із дуже багатим вибором додатків до основних компонентів Spring, таких як:

1. Spring Web[11] - додаток до Spring, який дозволяє створювати надійні веб-додатки та запускати їх через вбудований веб-сервер Apache Tomcat[12].
2. Spring Data[13] - додаток до Spring, який надає можливість використовувати різні бази даних за допомогою абстракцій. Підтримує моделі ORM, нативні SQL-запити та JPQL-запити.
3. Jackson - високопродуктивна бібліотека для Spring, яка надає можливість працювати із JSON-об’єктами.
4. Spring Test - додаток до Spring, який надає можливість тестування сервісів за допомогою написання юніт- та інших тестів..
5. LiquiBase - спеціалізований інструмент для управління комітами бази даних. Замість ручного створення таблиць, LiquiBase дозволяє створювати їх автоматично, вказуючи структуру, індекси, зовнішні ключі у незалежних файлах конфігурації (чейнджлогах). Використання цього інструменту гарантує консистентність бази даних та розгортання однакових структур на різних серверах.

РОЗДІЛ 2. ПОСТАНОВКА ЗАВДАННЯ ТА ВИМОГИ ДО СИСТЕМИ

2.1 Аналіз проблем у роботі з розкладом у студентів ДУІКТ

Новочасний освітній процес у Державному університеті інформаційно-комунікаційних технологій вирізняється великою активністю. Класичні методи поширення розкладів у форматі статичних файлів та друкованих таблиць або навіть сайтів з розкладами втрачають актуальність через важливість миттєвої реакції на зміни. Наприклад, в умовах воєнного стану, відключення електроживлення та повітряних тривог, студенти мають потребу в інструменті, який надасть гарантований та, найголовніше, швидкий доступ до оновленої, актуальної інформації про навчальні заняття без потреби у завантаженні важких сторінок або контакту із відповідальними за розклад.

Дослідження специфіки взаємодії студентів різних університетів із навчальним розкладом, що підкріплене результатами опитування за допомогою анкети, дозволило виділити ряд критичних проблем, що обумовлюють необхідність впровадження нових інструментів автоматизації. Ключовим дестабілізуючим фактором є висока динаміка змін у графіку занять: згідно з отриманими даними, 69% респондентів зазначили, що часто стикаються з раптовими оновленнями розкладу, про які дізналися із запізненням (рис. 2.1).

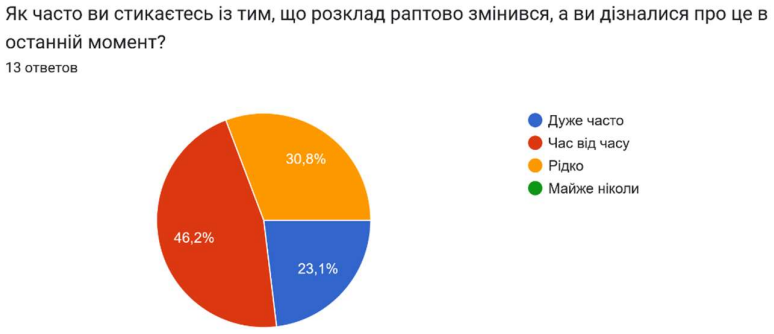


Рис. 2.1 Частота раптових змін у розкладі університетів

Це підтверджує неефективність традиційних методів пасивного моніторингу та необхідність системи миттєвих сповіщень.

Ще однією вагомою проблемою є незручність використання існуючих рішень на мобільних пристроях, аргументуючи це відсутністю адаптивного дизайну та табличного представлення розкладу, який не пристосований до екранів смартфонів (рис. 2.2).

Рис. 2.2 Відповіді студентів щодо незручності потомного розкладу

Отриманий зворотній зв’язок після закритого тестування серед фокус-групи, підтверджують високу ефективність та зручність запропонованого рішення. Більшість респондентів відзначили значне прискорення процесу отримання розкладу порівняно із існуючими методами, а також позитивно оцінили структурованість і читабельність виведення даних (рис 2.3).

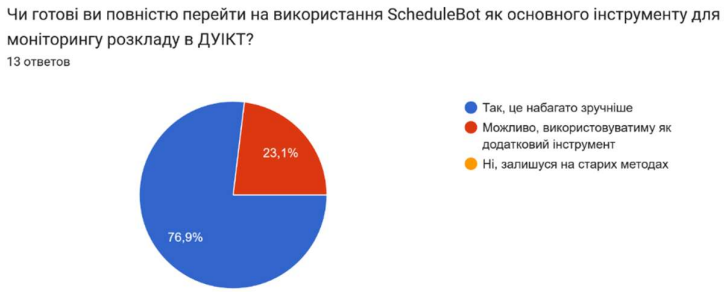


Рис. 2.3 Готовність використовувати ScheduleBot як основний інструмент для отримання інформації про розклад

Підсумовуючи результати проведеного аналізу, можна стверджувати, що існуючи методи отримання розкладу не повною мірою задовольняють потреби студентів. Висока динаміка змін у графіку, відсутність адаптивності веб-ресурсів створюють значні перешкоди для ефективного планування навчального часу. Отримані результати апробації на цільовій аудиторії ДУІКТ беззаперечно підтверджують гіпотезу про те, що впровадження системи ScheduleBot дозволить оптимізувати процес доступу до розкладу, підвищити поінформованість студентів та мінімізувати комунікаційні затримки в освітньому процесі.

2.2 Визначення функціональних вимог до Telegram-бота

Система автоматизації навчальних розкладів у форматі Telegram-бота призначена для надання швидкого, зручного та надійного доступу до своєчасного розкладу навчального закладу.

Функціональність системи забезпечується через взаємодію користувача із інтерфейсом бота у форматі діалогу із використанням текстових команд та інтерактивних елементів (inline-кнопок).

Основними користувачами інформаційними системами є:

1. Студенти;
2. Викладачі (Додатково).

Визначення основних функціональних можливостей системи

Для забезпечення взаємодії користувача із системою, функціональні вимоги мають охопити повний цикл взаємодії користувача із ботом, починаючи від моменту ініціалізації діалогу і закінчуючи автоматизацію внутрішніх процесів обробки даних. Процес роботи із системою розпочинається із ініціалізацією сесії бота командою /start, яка активує привітальне повідомлення та відкриває доступ до меню реєстрації. Реєстрація є критично важливим етапом роботи, оскільки дозволяє зберегти дані користувача для того, щоб запам’ятати його належність до

конкретного університету, інституту, групи, тощо, а також забезпечити кешування даних для оптимізації доступу.

Основний блок функцій зосереджений на отриманні актуальної інформації про розклад. Система має надавати можливість запитувати розклад на поточний та наступні дні, а також поточний та наступний тижні за допомогою відповідних команд або інтерактивних кнопок. Для більш гнучкого використання боту, має бути реалізована можливість отримувати дані не лише за фіксовані проміжки часу, а й за довільні, шоб покрити потреби користувача у випадку якщо йому потрібна функція бектрекінгу занять. У випадках коли на обраний час занять не заплановано, система має видавати зрозуміле для користувача повідомлення.

Система має надавати дві варіанти взаємодії із собою: за допомогою текстових команд (для старих версій телеграму, які не підтримують сучасні елементи інтерфейсу), а також навігаційного меню із використанням inline-кнопок, яке надасть можливість мінімізувати помилки користувача під час введення даних. Додаток має автоматично форматувати отримані з API або інших зовнішніх джерел даних у зручний для сприйняття вигляд, а у випадку великого обсягу інформації застосовувати механізм чанкування оброблених даних, щоб обійти обмеження Telegram на максимальну кількість символів у повідомленні (4096 символів) та гарантувати доставку інформації. Технічна стабільність системи має забезпечуватися фоновими процесами: оновлення даних зі сторонніх джерел за таймером, локальним кешуванням для доступу під час недоступні джерела, логуванням помилок та детальним інформуванням користувача про причини можливих помилок збоїв усередині або зовні системи. Більш розширений список функціональних вимог наведено у додатку 1 «Таблиця фунціональних вимог до системи Telegram-бота для отримання розкладів».

2.3 Визначення основних нефункціональних можливостей системи

Якість роботи системи визначається низкою нефункціональних вимог, які висуваються до її продуктивності, надійності та безпеки. Одним із пріоритетних показників є швидкодія: час відповіді системи при наявності кеші не повинен

перевищувати 500мс, а при зверненні до зовнішніх джерел даних - 2-х секунд. При цьому внутрішня обробка запиту на сервері без урахування мережевих затримок має виконуватися у межах 200мс. З точки зору масшабованості архітектура додатку має надавати послуги понад 1000 одночасних користувачів із пропускною здатністю не менше 100 запитів на секунду.

Викокий рівень надійності досягається завдяки стовідсотковій обробці всіх критичних помилок та використання fallback-механізмів, який дозволяє перемикатися на локальний кеш у разі недоступності зовнішніх джерел даних. Питання безпеки вирішується за допомогою використання протоколу HTTPS для усіх з’єднань та валідації усіх вхідних запитів. Для додаткового захисту усі критичні дані системи мають бути винесені в окремий .env файл і не зберігатися у відкритому коді проекту.

Зручність використання базується на принципі мінімізації зусиль: отримання результату має займати не більше трьох кроків, що відповідає «правилу трьох натискань». Це має реалізовуватися через впровадження inline-кнопок, які дозволять користувачу здійснювати навігацію без необхідності ручного введення текстових команд.

2.4 Вихідні дані та обмеження

Джерело JSON-розкладу**.** Головним джерелом даних для системи, що проєктується, є зовнішній інформаційний ресурс, який надає розклад занять у вигляді документу у форматі JSON, що було розглянуто у розділі 1. Використання цього формату є доцільним завдяки його структурованості, зручності для сприйняття людиною та машинної обробки, а також компактності. Такий підхід дозволяє серверній частині бота максимально швидко та ефективно десеріалізувати вхідні текстові дані у внутрішні типи об’єктів.

Отримання даних відбувається за допомогою HTTP-запиту типу GET. Зовнішнє джерело виступає у вигляді RESTful-сервісу, який повертає дані у вигляді структурованого документу. Для точної вибірки необхідного розкладу такі

запити параметризуються за допомогою номера академічної групи та часовими рамками, при цьому система, що розроблюється має враховувати можливі мережі затримки або тимчасову недоступність сервісу.

Основними характеристиками даного джерела даних є:

1. Протокол передачі: HTTPS;
2. Формат даних (content-type): JSON (application/json);
3. Тип взаємодії: REST API;
4. Спосіб отримання: запити за потребою або кроном (timed requests).

Дуже важливою ознакою є відсутність прямого контролю над джерелом, що накладає додаткові вимоги до обробок помилок, оновлення актуальності мапінгів DTO[15] та заберпечення відмовостійкості.

Особливості структури даних

Дані у форматі JSON мають плоску структуру, яка складається із об’єктів (занять), що вкладені у єдиний об’єкт, де кожний елемент масиву представляє собою окреме заняття.

Загальна структура JSON, що отримана з API, виглядає наступним чином (рис. 2.4):


```
{
  "schedule": [
    {
      "date": "16.03.2026",
      "timeStart": "11:10",
      "timeEnd": "12:30",
      "number": "3",
      "type": "Лк",
      "cabinet": "Дистанц.",
      "shortName": "ШІ-2017",
      "name": "Штучний інтелект",
      "addedOnDate": "06.02.2026",
      "who": "Кисіль Тетяна Миколаївна",
      "whoShort": "Кисіль Т.М."
    }
  ]
}
```


Рис. 2.4 Представлення даних про заняття, які були отримани з API Skedy.yacode.dev

Загальну характеристику структури можна описати наступним чином:

1. Всі заняття зберігаються в одному масиві “schedule”;
2. Кожен запис є незалежним від інших об’єктом;
3. Групування за датою відсутнє на рівні структури;
4. Логічне групування лягає на плечі розробника сервісу.

Це означає, що система має власноруч:

1. Групувати заняття за датами;
2. Сортувати їх за часом;
3. Оформлювати зручне представлення для користувача системи.

Також важливим є описати поля, що присутні у документі (табл. 2.1).

Таблиця 2.1

Опис полів JSON-документа

Назва поля	Тип поля	Опис поля
date	String	Дата проведення заняття у форматі dd.MM.yyyy
timeStart	String	Дата початку заняття
timeEnd	String	Дата закінчення заняття
number	String	Порядковий номер пари
type	String	Тип заняття (Де Лк - лекція, Пз - практичне заняття, Лб - лабораторна робота)
cabinet	String	Номер аудиторії або позначка «Дистанційно»
shortName	String	Скорочена назва дисципліни
name	String	Повна назва дисципліни

Продовження таблиці 2.1

addedOnDate	String	Дата додавання або оновлення запису
who	String	Повне ім’я викладача
whoShort	String	Коротке ім’я користувача

Ключові особливості обробки:

1. JSON у джерелі не містить групування за датами, тому необхідно створити логіку сортування та фільтрації.
2. Оскільки інформація про час представлена у вигляді рядків, необхідно створити логіку парсингу дати та часу, а також обробити можливі помилки.

3. Поле type може містити не тільки прості значення (Лк, Пз, Лб), а також й складені (наприклад, Пз т.6/з.2). Це вимагає додаткового парсингу або виділення типу заняття від додаткової інформації.

4. Відсутність гарантії порядку потребує від розробника створення додаткового алгоритму сортування та гарантування правильності відображення даних.

Зважаючи на вищезазначені проблеми та обмеження, використання вихідної моделі документи без обробки безпосередньо у додатку не є ефективним. Тому було прийнято рішення розробити власну внутрішню об’єктно-орієнтовану структурну модель даних. Така модель вирішує низку завдань:

1. Вона відкидає надлишкову інформацію, залишаючи в пам’яті лише ті поля, які потрібні кінцевому користувачу;
2. Автоматично нормалізує дати і час на етапі перетворення документу в об’єкти;
3. Інкапсулює в собі алгоритми самостійного сортування та фільтрації занять
4. Спрощує подальше опрацювання даних компонентами сервісу, знижуючи зв’язність коду.

Враховуючи ці архітектурні вимоги та результати вхідного документу, розроблена внутрішня модель даних набула ієрархічного вигляду. Схематичне представлення базових класів цієї структури та зв’язків між ними наведено на рис. 2.5.

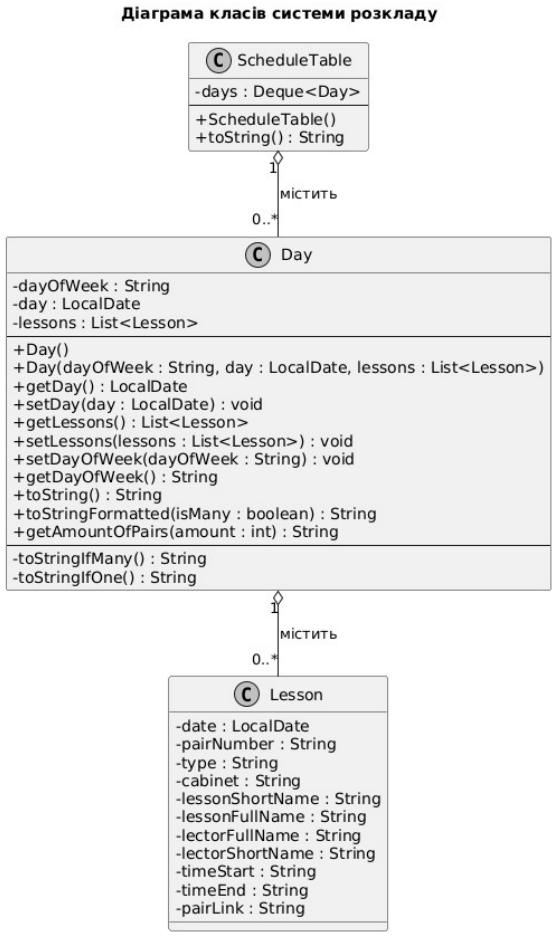


Рис. 2.5 Приклад внутрішньої структури розкладу

UML-діаграма класів відображає структуру сутностей системи, які відносяться до розкладу. Клас “ScheduleTable” агрегує у собі низку об’єктів типу “Day”, кожен із яких відображає окремий день розкладу. У свою чергу клас “Day” є контейнером для метаданих для внутрішнього опрацювання системою: day - дата у форматі LocalDate, назва дня тижня - dayOfWeek, а також колекції типу “Lesson”, який представляє кожне окреме заняття. Клас “Lesson” є шаблоном окремого заняття, яке містить усі необхідні дані, отримані з документу.

У ході аналізу функціональних та нефункціональних вимог було визначено основні сценарії використання бота користувачем, наприклад, реєстрація, пошук групи, отримання розкладу за день, тиждень та власну дату, а також обробку змін у розкладі. Додатково було схарактеризовано основні нефункціональні вимоги, які стосуються швидкодії, надійності та безпеки системи.

Важливою частиною аналізу було приділення уваги формату даних, що представлено у вигляді JSON-документу. Визначено, що хоча загалом формат є дуже зручним як для читання людиною, так і машинної обробки, конкретна реалізація JSON-документа має низку особливостей, які ускладнюють обробку, серед іншого, - плоска структура, відсутність сортування та групування за датою, текстове представлення дат, а також можливу неповноту даних. Через такі особливості є необхідним створення реалізації додаткових внутрішніх механізмів обробки, наприклад, нормалізація дат, групування та фільтрація занять у власні структури даних.

До того ж, було охоплено основні обмеження Telegram, як платформи. Врахування цих обмежень є важливим для розуміння та побудови надійної та стабільної системи.

Таким чином, можна зазначити, що результати даного розділу дозволили сформулювати повний перелік функціональних та нефункціональних вимог, обмежень, що визначають внутрішню логіку майбутньої системи та створюють базу для її подальшого проєктування.

РОЗДІЛ 3. ПРОЄКТУВАННЯ СИСТЕМИ SCHEDULEBOT

3.1 Архітектурна модель Telegram-бота

Інформаційна система для автоматизації моніторингу розкладів навчальних закладів на основі Telegram API побудована за клієнт-серверною архітектурою із використанням патерну MVC[16] (Model-View-Controller, який дуже підходить під контекст Spring), що використовує мультишарову модель обробки даних.

Клієнтом системи є звичайний користувач Telegram, котрий взаємодіє із ботом за допомогою підправки команд, що представлені або текстом (текстові команди) або inline-кнопками. Хоча текстові команди легші у реалізації, inline-кнопки надають розробнику додаткову можливість захисту від помилкових запитів користувача. У свою чергу, серверна частина додатку зроблена у вигляді додатку на мові програмування Java із використанням фреймворку Spring, який використовується у обробці та валідації запитів, взаємодією із зовнішнім API, формуванні відповіді та комунікації із базою даних.

Взаємодія між додатком та Telegram описується механізмом Long Polling. У цьому разі сервіс через деякий проміжок часу самостійно надсилає запити до центральних серверів Telegram для отримання оновлень (Update), кожне з яких містить дані про дію користувача - текст повідомлення, натискання кнопки або надісланий файл. До того ж оновлення мають у собі метадані користувача, що є дуже корисним для його реєстрації та подальшого кешування без зайвих дій з його сторони.

Загальна робота системи представлена на рис. 3.1.

Однією із особливостей реалізації сервісу є використання локального файлового кешу, у якому дані зберігаються у файлах (персистентному кеші), що ізольовані один від одного по групах. Використання такого підходу дає можливість

виключити конкуренцію під час спроби отримати ресурс, тим самим зменшити навантаження на сервіс, проте найбільшим плюсом кешування є можливість отримувати дані навіть якщо API за якихось причин тимчасово недоступний.

Підводячи підсумок, обрана схема забезпечує ефективну обробку запитів при мінімізації затримок, а також гарантію, що розклад буде отриманий навіть за умови тимчасової недоступності зовнішнього API.

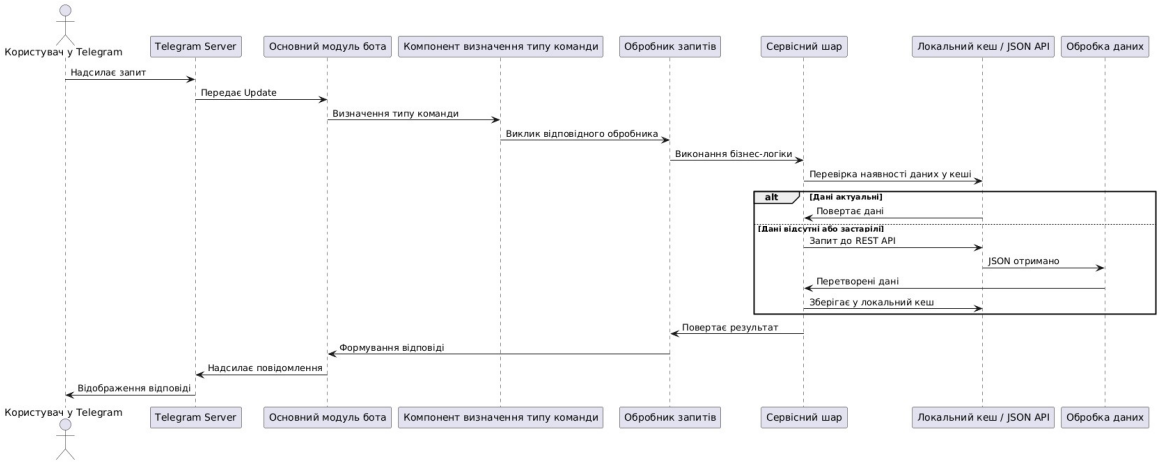


Рис. 3.1 Діаграма послідовності роботи сервісу отримання розкладів

Модульна структура проєкту

Додаток Telegram-бота для моніторингу навчальних розкладів побудований за модульної багаторівневої архітектури, яка реалізує принципи SOLID[17] для більш зручної підтримки та масштабування системи за рахунок розділення функціональності на окремі логічні компоненти, що об’єднані у шари (layers).

У межах даного проєкту можна виділити наступні основні шари:

1. Шар взаємодії з Telegram. Фактично це реалізація View-компонента із MVC архітектури. Даний модуль відповідає за відображення та прийом інформації, яка потім йде далі на обробку у наступні модулі, які знаходяться вище за ієрархією. Шар взаємодії з Telegram включає в себе основний клас бота, який реалізує механізм Long Polling та компонент для створення нових унікальних сесій для користувачів через віртуальні потоки Java.

До складу даного шару входять наступні компоненти:

a. Основний клас бота (точка входу або ентріпоінт);

b. Кеш потоків (використовується для створення та винищення сесій).

2. Шар маршрутизації запитів. Це реалізація DispatcherServlet. Він централізовано приймає команди користувача та проводить подальшу маршрутизацію у відповідні модулі обробки в залежності від стану користувача (Для запам’ятовування стану користувача використовується компонент з шару кешування, який буде описано нижче) для їх подальшого опрацювання. Застосування маршрутизатора дозволяє створити гнучку архітектуру, котра дає можливість додавати нові сценарії без внесення змін у базову логіку системи.

3. Шар обробки запитів (Модуль хендлерів). Можна сказати, що це реалізація Controller-компонента MVC. Обробники реалізують прикладну логіку взаємодії із користувачем. Кожен із таких хендлерів відповідає за власний (окремий) сценарій, як:

a. Реєстрація користувача;

b. Отримання розкладу;

c. Робота із налаштуваннями;

d. Відображення та робота із меню;

e. Робота із адмін-панеллю.

4. Шар сервісів. Модуль сервісів є ядром бізнес-логіки системи. Цей шар відповідає за наступні алгоритми:

a. Взаємодію із зовнішнім API та кешем;

b. Отримання та обробку даних розкладу;

c. Централізоване відправлення повідомлень усім користувачам;

d. Міграцію між різними версіями сервісів.

Сервіси містять логіку, що є незалежною від способу взаємодії із користувачем, що дозволяє повторно використовувати код не порушуючи принцип DRY[18].

5. Шар репозиторіїв. Задача цього шару дуже проста, але від того не стає менш важливою. Даний модуль реалізує взаємодію із базою даних за допомогою засобів Spring Data та Hibernate. Він забезпечує зберегання інформації про користувачів,

інститути, групи та інші сутності системи. Завдяки використанню абстракцій можна абстрагуватися від конкретної реалізації бази даних.

6. Шар кешування. Із метою покращення продуктивності додатку було реалізовано окремий модуль кешування. Він дає змогу зберегати великий перелік службових даних починаючи від даних про користувача і закінчуючи локальним збереженням розкладів. Особливості реалізації дозволяють уникнути боротьби за ресурс під час одночасного доступу та зменшує кільки вузьких місць у системі.

7. Шар обробки даних. Через перелічені вище особливості структури даних JSON-документа було створено окремий модуль, що відповідає за нормалізацію даних із зовнішніх джерел та перетворює їх на зрозумілу для системи внутрішню модель. Результатом роботи цього шару є формування структурованих об’єктів, що використовується сервісним шаром.

8. Шар моделей даних. Цей шар містить у собі класи, які описують внутрішні сутності системи, наприклад, розклад, день, окремі заняття, користувача, тощо. Ці моделі використовуються у різних шарах та дозволяються зберігати та використовувати лише необхідну для роботи інформацію.

Підбиваючи підсумок, можна сказати, що модульна система забезпечує чітке розділення відповідальності відповідно до принципів SOLID. Такий підхід підвищує гнучкість та полегшує подальше розширення додатку.

Логіка обробки запитів

Логіка обробки запитів у додатку реалізується за принципом поетапної обробки повідомлень, які надходять від користувача. Це дозволяє в будь-який момент змінити логіку, що стоїть за конкретним запитом або просто додати нові команди.

Процес обробки запитів починається з отримання оновлення від серверів Telegram, які надходять в основий шар додатку. Як було зазначено вище, кожне таке оновлення містить як корисну інформацію для обробки, наприклад, текст повідомлення або текст кнопки, так і корисну мета-інформацію для збереження,

зокрема ID користувача, його нікнейм або його мову (може бути дуже зручно для інтернаціоналізації!).

На першому етапі виконується аналіз стану користувача та його відправлення у відповідний обробник. Така централізація дозволяє спростити додавання доступних сервісів та команд та легко розширити систему.

Після того як повідомлення потрапляє в обробник, він проводить парсинг даних, вилучаючи з нього команду, визначає тип сервісу (реєстрація, примусове оновлення локальних даних, отримання розкладу, тощо), до якого належить команда та відправляє її далі. Обробник виступає проміжною ланкою між користувачем та потрібними сервісами та відповідає за окремий функціонал.

Після парсингу команди та її відправлення обробник звертається до необхідного класу у сервісному шарі. Ці класи виконують основну роботу, яка необхідна користувачу. Наприклад, якщо поступає запит на отримання розкладу, сервісний модуль здійснює запит на зовнішній API, отримує дані з JSON-документа та відправляє їх на обробку у шар обробки, де отримані дані проходять десеріалізацію та нормалізацію, перетворюючись на внутрішні сутності. Якщо API тимчасово недоступний, до шар сервісів звертається до модуля кешування та отримує необхідні дані із локального кешу.

Останнім етапом є повернення відповіді з сервісу до обробника, який передає його до рівня представлення (основного модуля бота), що у свою чергу приводить відповідь у зрозумілу для користувача форму.

Таким чином, реалізований алгоритм обробки даних забезпечує ефективну взаємодію із користувачем, а також слабку зв’язність між елементами системи, що дозволяє легко її розширити.

3.2 Проєктування компонента отримання даних

Дуже важливим компонентом побудованої системи є компоненти веб-запитів. Хоча цей шар не такий великий, як інший, він грає велику роль у додатку, оскільки надає доступ до актуальної службової інформації із зовнішніх джерел. Компонент

завантаження реалізує механізм отримання даних із REST-ендпоінтів, що повертають інформацію у форматі JSON-документа.

Отримання даних здійснюється за допомогою HTTP-запиту GET, який працює по захищенному протоколу HTTPS. Джерелом інформації виступає зовнішнє API, яке віддає актуальну інформацію про розклад.

Процес завантаження ініціюється користувачем, коли він надсилає запит про отримання розкладу. У разі, якщо API працює, система формує запит до джерела, в іншому ж випадку - читає дані з кешу.

Після цього система отримує JSON-документ, який містить актуальну інформацію про заняття у вигляди масиву об’єктів . Кожен об’єкт описує окремий день заняття та містить важливі атрибути, наприклад, час, номер пари, викладача, назву дисципліни, кабінет, тощо. Проте дані не є нормалізованими, тому наступним кроком є відправлення отриманих даних у модуль обробки. Цей шар займається їх нормалізацією та перетворенням у сутності внутрішньої моделі. Для цього використовуються спеціальні десеріалізатори, які дозволяють легко робити потрібні перетворення, а також змінювати логіку обробки за потребою.

Після успішного отримання та обробки даних відповідь записується у локальний кеш для подальшого зберігання.

Особливістю реалізації є відсутність контролю за зовнішнім джерелом, що накладає на розробника додаткові вимоги до обробки помилок та перевірки коректності даних. Наприклад, шар веб-запитів має механізм повторних запитів, які виконуються, якщо під час отримання була отримана помилка.

Механізм парсингу та перетворення JSON-структур

У системи, що розробляється, парсинг JSON-структур реалізовано за допомогою бібліотеки з відкритим кодом Jackson, яка написана мовою Java та забезпечує надзвичайно ефективну серіалізацію та десеріалізацію даних. Однак окрім стандартних можливостей бібліотеки було створено додаткові компоненти, які дозволяють обробляти специфічну для вхідного документа структуру.

Основою шару парсингу є використання власних десеріалізаторів, які дають можливість прочитати та перетворити вхідний документ на зручні для внутрішньої моделі системи. Кожен такий десеріалізатор відповідає за обробку конкретного поля або типу даних, що у свою чергу забезпечує гнучкість, масштабованість та модульність створеної реалізації.

Одним із прикладів десеріалізаторів, які використовують додаткову логіку, яка не є притаманною для стандартних засобів, є компонент, що використовується під час оновлення сервісу до останньої версії. Після того, як додаток запускається, він викликає шар обробки, який що отримує текстове значення збережених даних користувача і виконує міграцію у нові поля БД, якщо такі існують.

Процес парсингу можна описати схемою, представленою на рис. 3.2.

Рис. 3.2 Діаграма стану, яка показує, яким чином відбувається внутрішня робота шару обробки

Серед особливостей реалізації обробки JSON-документів можна зазначити використання двосторонніх черг (Deque). Вибір цієї структури даних зумовлений внутрішньою роботою цієї колекції, яка, в сутності, є зв’язним списком. У зв’язному списку додавання та видалення елементу завжди відбувається із постійним часом (O(1), також відсутність потреби у реалокації пам'яті, оскільки зв’язні списки всередині не використовують масиви. Серед мінусів зв’язних списків можна зазначити, що у такій колекції немає рандомного доступу (тобто не можна шукати елемент за індексом), проте це не має впливу на даний сервіс, оскільки він завжди виводить розклади послідовно, а не за індексом.

Тобто можна сказати, що реалізований механізм обробки JSON є достатньо гнучким, його можна легко розширити та він добре інтегрується із загальною архітектурою системи. Навіть якщо зв’явиться потреба в обробці більш складних даних, набір десеріалізаторів дуже просто розширити.

Формування внутрішніх сутностей системи

Останнім етапом обробки JSON-даних є створення внутрішньої моделі. Цей етап є ключовим у забезпеченні подальшої роботи системи, оскільки забезпечує зручне представлення інформації.

Після десеріалізації інформації створюється колекція типу Lesson, яка інкапсулює основну інформацію про заняття. На цьому етапі дані ще не оброблені, а тому потребують додаткової обробки. Цей процес передбачає інтеграцію метаданих, які відсутні в зовнішньому джерелі, але важливі для розширення функціональності бота.

Після підготовки колекції для подальшого опрацювання вона відправляється на збагачення, як, наприклад, додавання посилань до онлайн-занять, якщо вони вказані (для своєї групи студенти самостійно можуть додавати такі посилання, але заради безпеки вони перевіряються за маскою DNS-адрес найпопулярніших ресурсів, як Google Meet та Zoom). Така превентивна валідація на рівні доменних імен запобігає потраплянню до інтерфейсу некоректних посилань та гарантує безпеку цифрового середовища користувача. Це дозволяє зберегти усі необхідні

дані в одному місці, щоб студенти не зайвий раз не шукали потрібну інформацію в різних групах.

Після збагачення дані йдуть на групування за датами. Для цього використовується TreeMap, яка забезпечує автоматичне сортування занять у хронологічному порядку, де кожне заняття відноситься до відповідної дати проведення. Вибір саме цієї структури даних зумовлено порядком сортування ключів типу LocalDate, які сортуються від старіших до найновіших, що дозволяє уникнути потребу у написанні власного алгоритму порівняння.

Цікавою особливістю реалізації є своєрідне забезпечення безперервності часового діапазону. Наприклад, якщо є вільний день між двома навчальними днями, то система автоматично додає відсутні дні для того, шоб уникнути пропусків у відображенні розкладу, а також покращити досвід користувача. Такий підхід створює ефект цілісного навчального календаря, де студент може побачити не тільки робочі години, а й вихідні та святкові дні.

Після створення, фільтрації та групування занять на їх основі створюються об’єкти типу Day, які мають у собі наступні дані:

1. Дату;
2. Текстове представлення назви дня тижна (Це свого роду кеш, щоб постійно не перетворювати дату, бо парсинг дат має накладні витрати);
3. Список занять.

Також щоб зберегти логіку представлення занять, вони сортуються у межах кожного дня за порядковим номером пари, що забезпечує коректний порядок відображення розкладу.

Сформовані об’єкти Day інкапсулюються у колекцію-контейнер ScheduleTable - основної структури, яка представляє розклад, який вже потім форматується та виводиться користувачу. У випадку, якщо обсяг даних перевищує ліміт Telegram (4096 символів), використовується механізм розбиття розкладу на частини, що дозволяє обійти обмеження API. Процедура чанкування гарантує розділення тексту виключно на межах об’єктів типу Day, що запобігає розриву інформації про заняття в межах однієї дати. Загальна робота сервісу відображена на рис. 3.3.

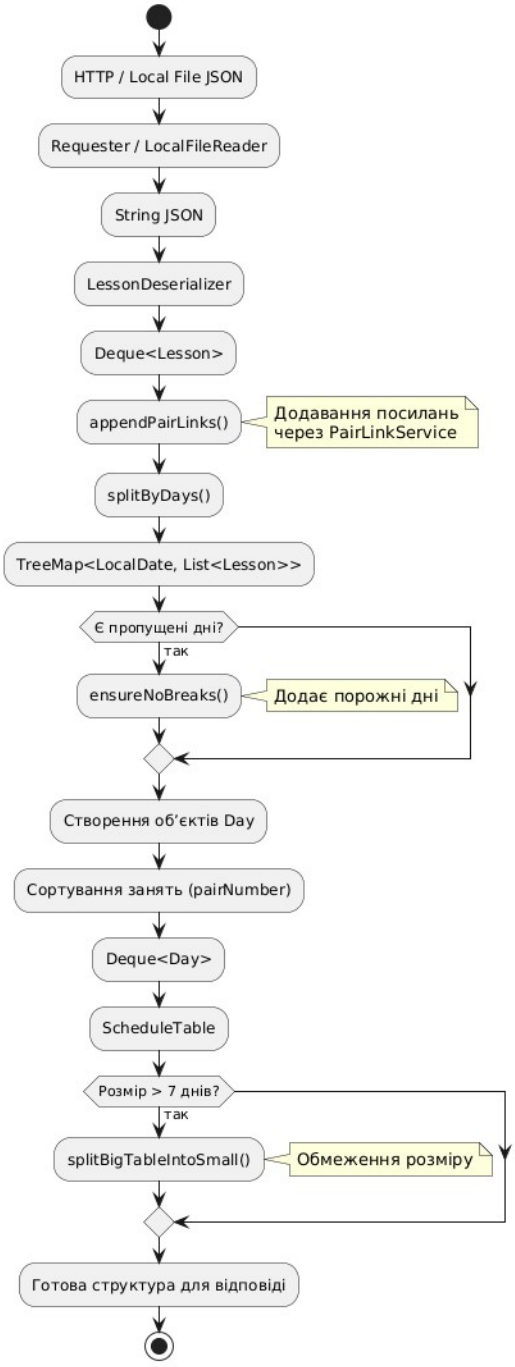


Рис. 3.3 Діаграма діяльності сервісу створення розкладу

Підводячи підсумок можна сказати, що формування внутрішнього представлення отриманих даних дозволяє перетворити неструктурований JSON-документ у нормалізовану, впорядковану, збагачену та, головне, зручну модель для ефективної взаємодії між компонентами системи та користувачем.

3.3 Створення механізму кешування даних

Для підвищення відказостійкості системи було прийнято прийнято рішення додати сервіс кешування розкладів.

Основною ідеєю кешування є збереження отриманих JSON-документів та їх подальше зберігання та використання у випадку, якщо зовнішнє API тимчасово не працює. Такий підхід дозволяє зберегти корисність сервісу навіть за умови аварії зовні.

У системі використовується реалізація файлового персистетного кешу, де дані про розклад зберігаються у вигляді окремних файлів, які відповідають кожній окремій групі студентів, що у свою чергу дозволяє зменшити ризик конфліктів під час одночасного читання файлів, зменшити час на оброку потрібного розкладу та незалежність обробки запитів.

Оновлення кешу здійснюється за допомогою планового механізму, який викликається у нічний час, коли навантаження на систему є мінімальним, що дозволяє зменшити без того велике навантаження на додаток у пікові години та забезпечити актуальність даних у випадку відсутності зовнішнього джерела.

Надважливою особливістю даної реалізації є те, що локальний кеш використовується як fallback-механізм на який система автоматично перемикається, якщо виникають будь-які мережеві помилки. Це збільшує ступінь надійності та адаптації до нестального мережевого оточення.

Застосування файлового кешу має наступні переваги перед іншими вже готовими рішеннями, як, наприклад, Redis[19]:

1. Простота реалізації - просто не значить «погано». Простота дозволяє перекинути лімітовані ресурси на інші частини системи, які потребують більш пильної уваги.
2. Відсутність необхідності у сторонніх залежностях - налаштування Redis для Spring потребує додаткових зусиль через відсутність готової

реалізації рівня інтерфейсів. Знову ж таки це дозволяє не витрачати час на надмірну інженерію.

3. Зберігання даних між запусками системи та віртуальної машини - Redis - це, в першу чергу, резидентна СУБД. Вона працює лише допоки додаток активний і не зберігає дані між сеансами.

4. Простота міграції - навіть зашифровані файли легше мігрувати, ніж дані СУБД.

Таким чином, впроваджений механізм кешування підвищує надійність системи, дозволяючи незважати на короткочасну недоступність сторонніх джерел.

3.4 Проєктування логіки взаємодії користувача із чат-ботом

Взаємодія користувача із додатком реалізована двома способами: за допомогою текстових повідомлень, а також кнопок. Використання кнопок дозволяє зменшити кількість помилок, які може зробити користувач, що спрощує обробку помилок для розробника та полегшує взаємодію споживача із сервісом, оскільки складні команди замінюються на просте натискання потрібної кнопки.

Основним елементом графічного інтерфейсу є головне меню, яке надає користувачу основний перелік текстових команд: отримання розкладу, налаштування, а також додаткових команд, якщо користувач є адміністратором. Проте, для покращення UX, меню також доповнюється кнопками, які відповідають за наступні функції:

1. Отримання розкладу на поточний день;
2. Отримання розкладу на наступний день;
3. Отримання розкладу на поточний тиждень;
4. Отримання розкладу на наступний тиждень;
5. Доступ до налаштувань (яке у свою чергу має своє меню, яке дозволяє керувати основними властивостями акаунта).

Самі кнопки реалізовані у вигляді Inline-Keyboard, які входять у стандартну бібліотеку Telegram API та представлені у вигляді приєднаний до повідомлення

сутностей із закодованими у них командами. Такий підхід дозволяє користувачу взаємодіяти із системою без потреби запам’ятовувати довгі команди або навіть їх послідовності.

Саме формування меню лежить на плечах серверної частини. В залежності від стану користувача та контексту взаємодії, набір кнопок динамічно змінюються. Для прикладу можна привести появу можливості дивитися розклад після реєстрації або отримання додаткових функцій, якщо користувач отримав права адміністратора.

Обробка натискання кнопок є такою ж самою, як і обробка текстових повідомлень, бо, фактично, кнопки є такими самими текстовими командами, але представленими через графічні елементи. Такі отримані команди передаються до шару обробки запитів, яка визначає за що відповідає надана команда та маршрутизує запит далі, до рівня сервісів.

Підводячи підсумки, використання кнопок є дуже корисним, оскільки зменшує навантаження на користувача, котрому не потрібно запам’ятовувати велику кількість команд, щоб отримати доступ до необхідного функціоналу.

Порівняння Inline-Keyboard та Reply-Keyboard

Як було зазначено вище, дана система використовує Inline-кнопки, але чим зумовлений даний вибір? Telegram API надає розробнику два основних типи кнопок: Inline-Keyboard та Reply-Keyboard, кожен з яких має свої переваги та сфери застосування.

Reply-Keyboard (рис. 3.4) являє собою звичайну клавіатуру, яка відображається замість стандартної клавіатури користувача. Натискання кнопок призводить до відправлення текстового повідомлення у чат, яке можна побачити (а це доволі сильно засмічує чат, що є не дуже гарним для навігації користувача) та обробляється як звичайна текстова команда. Такий підхід є легким у реалізації, оскільки не потребує обробки Callback-запитів. Єдиною перевагою цього виду кнопок зазначена простота реалізації, а ось недоліків більше:

1. Засмічення чату;

2. Відсутня можливість передавати додаткові параметри разом із запитом;
3. Не може зберегати контекст взаємодії.

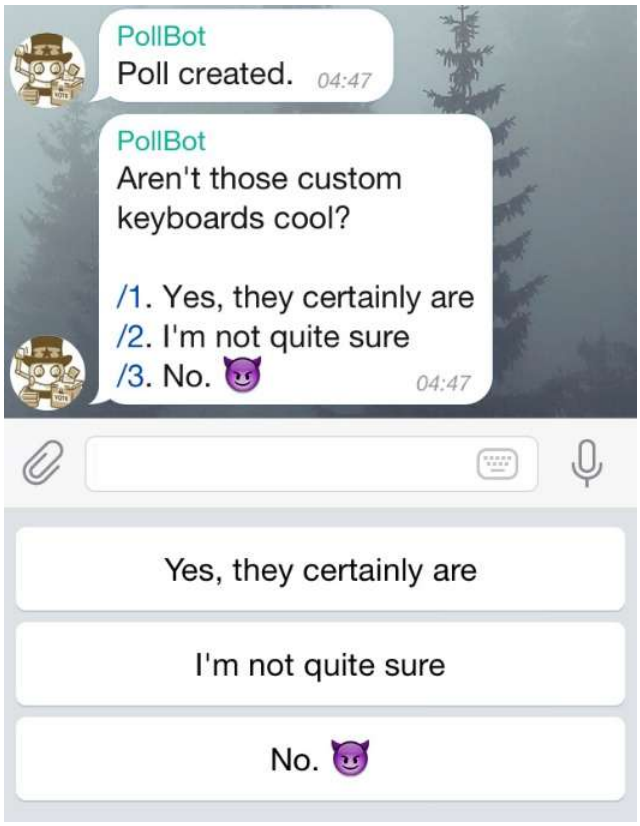


Рис. 3.4 Приклад Reply-Keyboard

У свою чергу, Inline-Keyboard (рис. 3.5) прикріплена до повідомлення та дозволяє обробляти дії користувача через Callback-запити. Кожна кнопка окрім команди може містити й додаткові метадані, котрі передаються разом із запитом та дозволяють реалізовувати більш складні сценарії взаємодії.

Серед переваг Inline-Keyboard можна зазначити наступні:

1. Не засмічує чат текстовими повідомленнями;
2. Можливість передачі метаданих разом із запитом;
3. Підтримка контекстної взаємодії;
4. Зручність для навігації та вибору значень.

Проте, звісно, існують і недоліки, наприклад:

1. Більш складна реалізація;

2. Необхідність опрацювання Callback-запитів;
3. Більш складна підтримка.

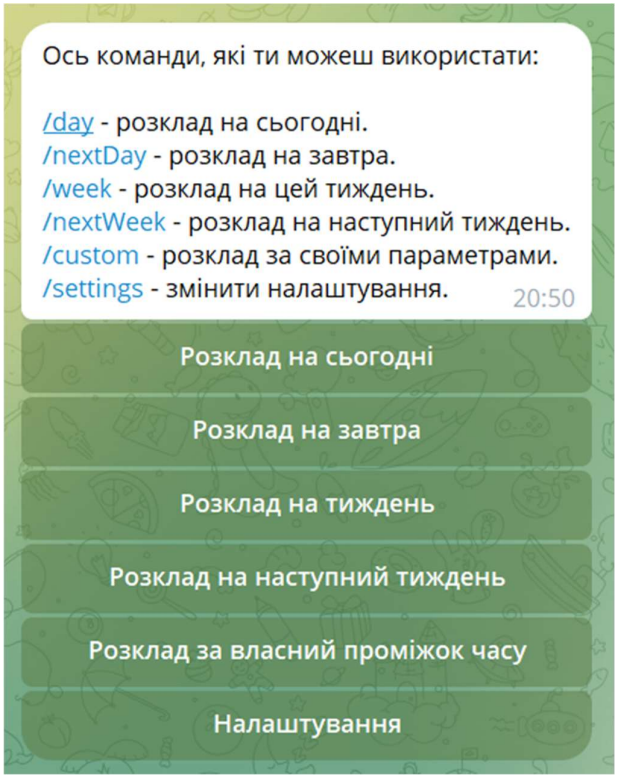


Рис. 3.5 Приклад Inline-Keyboard

Тому у системі, що розробляється, було прийнято рішення використовувати саме Inline-Keyboard, яка дозволяє створювати складі сценарії взаємодії, а також значно полегшує графічне навантаження на користувача через відсутність засміченості робочого простору.

Повідомлення про помилки

Під час своєї роботи будь-яка система схильна до появи виняткових ситуацій, які необхідно обробити. Такі ситуації можуть бути не лише результатом внутрішньої помилки, а й зовнішнього впливу на систему (користувач ввів неправильні дані або зовнішнє джерело не відповідає на запити, тощо). Через це у додатку реалізований комплексний механізм обробки помилок, який дозволяє зберегти стабільність стану системи та коректну роботи із користувачем.

Основними проблемними місцями, де можуть виникнути помилки, є:

1. Недоступність зовнішнього джерела;
2. Помилки під час виконання HTTP запитів;
3. Некоректна структура JSON запитів;
4. Неправильно введені дані від користувача;
5. Відсутність необхідних даних у системі.

Для обробки виняткових ситуацій у системі використовуються власні класи помилок, наприклад, RequestCreationFailedException або ScheduleFetchException. Використовування власних помилок дозволяє давати більш точні причини виникнення помилки та відокремити бізнес-логіку від технічних помилок.

У випадку виникнення помилки система не завершує свою работу, а використовує fallback-механізми. Наприклад, якщо зовнішнє джерело недоступне, то замість нього використовуються дані локального кешу, який реалізовано у шарі кешування. Такий підхід дозволяє отримувати розклад навіть якщо немає мережевого зв’язку або збої зовнішнього сервісу.

Окрім обробки помилок у додатку реалізований механізм логування із використанням відповідних бібліотек. Усі помилки логуються у журнал, який потім може передивитися розробник, щоб проаналізувати проблемні місця системи.

З точки розу взаємодії із користувачем, при появі будь-яких виняткових ситуацій, система формує зрозумілі повідомлення із причиною помилки, які не містять технічних деталей, які можуть перенавантажити користувача.

Таким чином, реалізований механізм обробки помилок поєднує у собі використання власних помилок, логування аномалій для їх подальшого аналізу та виправлення, а також формування повідомлень для користувача.

РОЗДІЛ 4. РЕАЛІЗАЦІЯ СИСТЕМИ ОТРИМАННЯ НАВЧАЛЬНИХ РОЗКЛАДІВ SCHEDULEBOT

4.1 Опис реалізації серверної частини

Програмна реалізація системи ScheduleBot базується на використанні такого сучасного стеку технологій, як мова програмування Java версії 21, фреймворку Spring Boot 3.4.1 та його підмодулів, а за зберігання даних відвовідає реляційна база даних PostgreSQL. Вибір такої програмної основи зумовлений необхідністю побудови масштабованої, надійної, стійкої до навантажень та продуктивної системи. Проєкт організовано за модульним принципом, де кожен пакет відповідає за конкретний рівень абстракції (рис. 4.1).

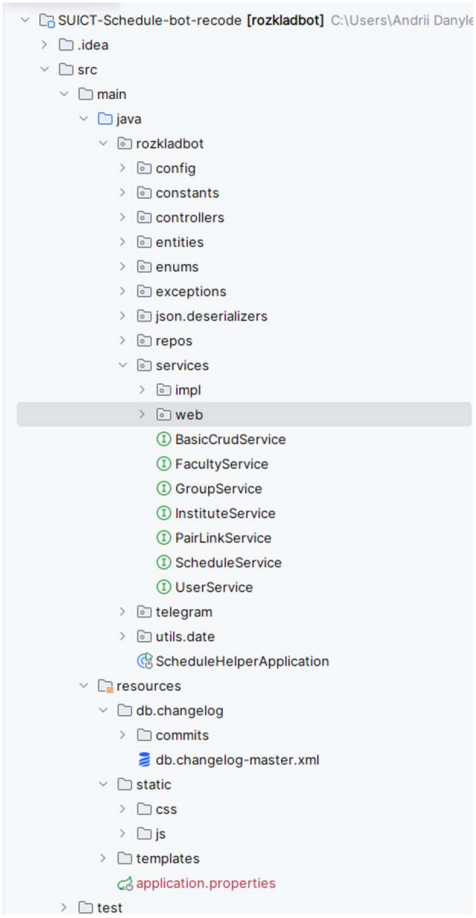


Рис. 4.1 Структура проєкту

У структурі проєкту виділено кілька ключових рівнів, які логічно розділяють відповідальність між компонентами системи. Серед основних пакетів можна зазначити наступні:

1. rozkladbot.telegram.bot - містить основний клас боту, який забезпечує з'єднання із серверами Telegram за моделлю Long Polling та створює нові сесії користувачів та перехоплює потік вхідних даних від них.

2. rozkladbot.telegram.router - імплементація маршрутизатору, який відповідає за направлення повідомлень до відповідних обробників за допомогою станів користувача.

3. rozkladbot.telegram.handlers - містить бізнес-логіку взаємодії з користувачем. Компоненти цього рівня відповідають за взаємодію із користувачем: проведення реєстрації, формування та вивід розкладу, управління налаштуванням профілю, а також виконання команд адміністратора.

4. rozkladbot.services - сервісний шар, що інкапсулює логіку роботи із зовнішніми джерелами, алгоритми агрегації даних, механізм багатошарового кешування та управління фоновими задачами.

5. rozkladbot.entities - фундаментальний рівень моделі додатку, який містить POJO[20] для роботи внутрішніх сервісів.

6. rozkladbot.repos – містить компоненти доступу до фізичної бази даних. Завдяки інтеграції Spring Data JPA, він дозволяє ефективно виконувати операції читання та запису на основі механізмів об'єктивно-реляційного відображення, позбавляючи розробника потреби у написанні використання JDBC та написання низькорівневих SQL-запитів.

Основні класи та модулі

Центральним компонентом, що забезпечує інтеграцію із платформою Telegram є клас ScheduleBot. Технічно цей компонент реалізовано шляхом успадкування від класу AbilityBot (Спеціальна абстракція Telegram API, яка містить потрібні методи для взаємодії із месенджером). Під час запуску системи ScheduleBot ініціалізує захищене мережеве з'єднання із серверами Telegram по протоколу HTTPS. Для

автентифікації використовується спеціальний криптографічний токен, який має бути згенерованим через службу BotFather. Основною задачею цього класу є безперервне прослуховування потоку вхідних оновлень від користувача, їх первинна ідентифікація та створення або відновлення індивідуальних користувацьких сесій у системі (рис. 4.2).

```
@Component("scheduleBot")
public class ScheduleBot extends AbilityBot {

  private final Router router;
  private final ThreadMemoryCache threadCache;
  private final UserMemoryCache userCache;
  private final ExecutorService telegramBotExecutor;

  @Autowired
  public ScheduleBot(
      ThreadMemoryCache threadCache,
      Environment environment,
      UserMemoryCache userCache,
      @Lazy Router router,
      ExecutorService telegramBotExecutor) {
    super(environment.getProperty("telegram.bot.api.key"),
        environment.getProperty("telegram.bot.name"));
    this.router = router;
    this.threadCache = threadCache;
    this.userCache = userCache;
    this.telegramBotExecutor = telegramBotExecutor;
  }

  public Ability startBot() {
    return Ability
        .builder()
        .name("start")
        .info("ScheduleBot")
        .locality(ALL)
        .privacy(PUBLIC)
        .action(ctx -> {
          long chatId = ctx.chatId();
          CompletableFuture<?> existing = threadCache.get(chatId);
          if (existing != null && !existing.isDone()) {
            return;
          }
          threadCache.put(chatId, CompletableFuture.completedFuture(null));
          telegramBotExecutor.submit(() -> {
            try {
              router.route(ctx.update(), chatId);
            } finally {
              threadCache.remove(chatId);
            }
          });
        })
        .build();
  }

  public Reply replyToButtons() {

    BiConsumer<BaseAbilityBot, Update> action = (bot, upd) -> {
      long chatId = getChatId(upd);
      CompletableFuture<?> existing = threadCache.get(chatId);
      if (existing != null && !existing.isDone()) {
        return;
      }
      threadCache.put(chatId, CompletableFuture.completedFuture(null));
      telegramBotExecutor.submit(() -> {
        try {
          router.route(upd, chatId);
        } finally {
          threadCache.remove(chatId);
        }
      });
    };
    return Reply.of(
        action,
        upd -> {
          long chatId = getChatId(upd);
          return userCache.existsByKey(chatId)
              && (upd.hasMessage() || upd.hasCallbackQuery());
        }
    );
  }

  @Override
  public long creatorId() {
    return 1;
  }
}
```

Рис. 4.2 Реалізація компоненту ScheduleBot

Проте для дотримання принципів SOLID, логіка обробки не була зосерджена в одному класі. Компонент Router виступає в ролі маршрутизатора: він аналізує стан користувача і делегує подальше виконання запиту конкретному обробнику команд (рис. 4.3).

```java
package rozkladbot.telegram.router;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import org.telegram.telegrambots.meta.api.objects.Update;
import rozkladbot.entities.HandlerContext;
import rozkladbot.entities.User;
import rozkladbot.telegram.caching.UserCache;
import rozkladbot.telegram.factories.HandlerFactory;
import rozkladbot.telegram.handlers.*;
import rozkladbot.telegram.handlers.commandresolver.CommandsResolver;
import rozkladbot.telegram.utils.message.MessageUtils;

import static rozkladbot.enums.UserState.*;

@Component("routerImpl")
public class RouterImpl implements Router {

  private final UserCache userCache;
  private final CommandsResolver commandsResolver;
  private final HandlerFactory handlerFactory;

  @Autowired
  public RouterImpl(
      CommandsResolver commandsResolver,
      UserCache userCache,
      HandlerFactory handlerFactory
  ) {
    this.userCache = userCache;
    this.commandsResolver = commandsResolver;
    this.handlerFactory = handlerFactory;
  }

  @Override
  public void route(Update update, long chatId) {
    User user = userCache.get(chatId);
    if (user == null) {
      user = User.getDefaultUser(update, chatId);
      userCache.put(chatId, user);
    }
    boolean override = !update.hasMessage();
    user.setLastSentMessageId(MessageUtils.getCorrectMessageIdWithOffset(update));
    if (user.isRegistered()) {
      commandsResolver.resolveCommand(update, user);
    }
    HandlerContext handlerContext = new HandlerContext(user, update, override);
    if (user.getUserState().equals(AWAITING_SETTINGS)) {
      handlerFactory.getStrategy("settingshandler").handleRequest(handlerContext);
    }
    if (user.getUserState().equals(AWAITING_GREETINGS)) {
      handlerFactory.getStrategy("newusershandler").handleRequest(handlerContext);
      return;
    }
    if (registerUserStates.contains(user.getUserState())) {
      handlerFactory.getStrategy("registrationhandler").handleRequest(handlerContext);
    }
    if (scheduleUserStates.contains(user.getUserState())) {
      handlerFactory.getStrategy("schedulefetchhandler").handleRequest(handlerContext);
    }
    if (user.getUserState().equals(MAIN_MENU)) {
      handlerFactory.getStrategy("mainmenuhandler").handleRequest(handlerContext);
    }
    if (adminUserStates.contains(user.getUserState())) {
      handlerFactory.getStrategy("admincommandshandler").handleRequest(handlerContext);
    }
  }
}
```

Рис. 4.3 Реалізація компоненту Router

Такий підхід дозволяє легко розширювати функціонал бота, додаючи нові команди без ризику порушити роботу вже існуючих компонентів.

Також важливими компонентами системи є обробники команд. Вони відповідають за подальшу обробку отриманих команд після визначення їх

маршруту. Для цієї мети було впроваджено єдиний інтерфейс HandlerStrategy (рис. 4.4), який реалізує патерн «Стратегія», що визначає єдиний метод HandleRequest.

```
public interface HandlerStrategy {
  void handleRequest(HandlerContext ctx);
}
```

Рис. 4.4 Метод інтерфейсу обробки запитів

Використання об’єкта HandlerContext (рис. 4.5) дозволяє інкапсулювати усі необхідні для обробки дані в одну логічну структуру, що у свою чергу спрощує сигнатури методів та покращує читабельність коду.

```
public class HandlerContext {
  private final User user;
  private final Update update;
  private final boolean overrideMessage;
}
```

Рис. 4.5 Клас, що містить у собі контекст обробки

Для того, щоб повноцінно реалізувати патерн «Стратегія» та позбавитися зайвих залежностей, було прийняте рішення зробити централізовану фабрику (рис. 4.6), яка дозволяє динамічно керувати усіма стратегіями. Завдяки контейнеру Spring, фабрика автоматично збирає всі біни, які реалізують інтерфейс HandlerStrategy у словник, де ключ є назвою класу.

```
@Component
public class HandlerFactory {
  private final Map<String, HandlerStrategy> strategies;

  public HandlerFactory(List<HandlerStrategy> strategies) {
    this.strategies = strategies.stream().collect(Collectors.toMap(strategy ->
        strategy.getClass().getSimpleName().toLowerCase(), Function.identity()));
  }

  public HandlerStrategy getStrategy(String strategyName) {
    return Optional.ofNullable(strategies.get(strategyName)).orElseThrow(() -> new
NoSuchHandlerStrategyFoundException(
        ErrorConstants.NO_SUCH_STRATEGY_FOUND.formatted(strategyName)));
  }
}
```

Рис. 4.6 Реалізація фабрики обробників

Завдяки використанню фабрики, можна легко додавати нові функції до бота, які пов’язані із обробниками (наприклад, нові меню або адмін-дашборди) просто створюючи новий клас-обробник без необхідності змінювати код фабрики.

4.2 Реалізація модулю парсингу розкладу

Модуль парсингу є ключовим компонентом інтеграції, оскільки він забезпечує роботу внутрішніх систем із зовнішніми джерелами даних. Його головна мета – абстрагувати складність мережевої комунікації, отримати необроблені документи та гарантувати їхню безпечну трансформацію у внутрішні об’єкти системи для подальшої обробки, кешування та збереження. Це гарантує, що всі наступні етапи роботи бота спиратимуться на консистентні та валідовані структури даних.

Отримання даних через веб-запити

Архітектурна реалізація процесу отримання розкладів базується на створенні спеціалізованого мережевого сервісу WebRequestService (рис. 4.7). Як основний інструмент для HTTP-взаємодії із зовнішніми API використовується синхронний клієнт RestTemplate з екосистеми Spring Web. Вибір цього компонента дозволяє інкапсулювати низькорівнені деталі роботи із мережевими протоколами, такі як відкриття та закриття з’єднань, управління таймаутами з’єднання або читання потоків даних, делегуючи управління ресурсами безпосередньо фреймворку Spring.

Критично важливим рішенням у сервісі є алгоритм динамічної генерації цільових URL-адрес, який реалізовано за допомогою методу buildLink. Замість використання жорстко заданих значень, алгоритм автоматично будує повне посилання на джерело з декількох складових частин. Базове посилання на джерела повністю винесено за межі вихідного коду програми і динамічно інжектиться під час створення Spring-контексту за допомогою анотації @Value з конфігураційного файлу application.properties. Цей архітектурний крок дозволяє додавати або змінювати цільові сервери або параметри підключення без необхідності перекомпіляції всього проєкту.

```
@Component
public class WebRequestServiceImpl implements WebRequestService {

  private static final Logger logger = LoggerFactory.getLogger(WebRequestServiceImpl.class);
  private final String BASE_LINK;
  private final RestTemplate restTemplate;
  private final RetryTemplate retryTemplate;

  public WebRequestServiceImpl(
      @Value("${base.request.link}") String baseLink,
      RestTemplate restTemplate,
      RetryTemplate retryTemplate) {
    this.BASE_LINK = baseLink;
    this.restTemplate = restTemplate;
    this.retryTemplate = retryTemplate;
  }

  @Override
  public String makeRequest(Map<String, String> params, String additionalPaths)
      throws URISyntaxException {
    URI uri = buildLink(params, additionalPaths);
    logger.info(OPENED_CONNECTION_BY_URL, uri);
    return retryTemplate.execute(ctx -> restTemplate.getForObject(uri, String.class));
  }

  private URI buildLink(Map<String, String> params, String additionalPaths)
      throws URISyntaxException {
    StringBuilder link = new StringBuilder(BASE_LINK);
    if (additionalPaths != null) {
      link.append(additionalPaths);
    }
    for (String key : params.keySet()) {
      if (Objects.equals(GROUP_NAME, key)) {
        continue;
      }
      link.append(QUERY_PARAMETER_DELIMITER).append(key).append(QUERY_PARAMETER_KEY_SEPARATOR)
          .append(params.get(key));
    }
    return new URI(link.toString());
  }
}
```

Рис. 4.7 Реалізація сервісу мережевих запитів WebRequestService

Для надання сервісу додаткової надійності було використано механізм повторів у випадку невдалого запиту. Якщо сервіс видає помилку, то метод автоматично повторює своє виконання через одну секунду протягом 3 спроб. Налаштування цього механізму описано біном retryTemplate (рис. 4.8) у пакеті конфігурації.

```
@Bean
public RetryTemplate retryTemplate() {
  RetryTemplate retryTemplate = new RetryTemplate();
  FixedBackOffPolicy fixedBackOffPolicy = new FixedBackOffPolicy();
  fixedBackOffPolicy.setBackOffPeriod(1000);
  SimpleRetryPolicy simpleRetryPolicy = new SimpleRetryPolicy();
  simpleRetryPolicy.setMaxAttempts(3);
  retryTemplate.setBackOffPolicy(fixedBackOffPolicy);
  retryTemplate.setRetryPolicy(simpleRetryPolicy);
  return retryTemplate;
}
```

Рис. 4.8 Налаштування механізму повтору запитів

Використання динамічної побудови посилань для запитів дає можливість отримувати будь-яку потрібну інформацію лише змінюючи параметри запиту без необхідності змінювати код, а механізм повтору захищає сервіс від короткочасних проблем із зовнішніми джерелами.

Архітектура десереалізації JSON

Оскільки дані, отримані із зовнішніх джерел (зокрема API) представлені у вигляді JSON документу було прийняте рішення розробити шар десеріалізації JSON. Він дозволить забирати із документу лише потрібні дані та, при необхідності, проводити додаткові операції над ними. Для того, щоб уникнути жорсткої прив’язки до бізнес-логіки формату десеріалізованих даних було створено загальний інтерфейс JsonDeserializer<T> (рис. 4.9), який використовує функціонал дженеріків[21].

```
public interface JsonDeserializer<T> {
    Deque<T> deserialize(String json);
}
```

Рис. 4.9 Загальний інтерфейс десеріалізаторів

Використання Deque у цьому випадку є доречним, оскільки прибирає потребу у реалокації пам’яті, як у випадку із List, а також надає можливість обробляти елементи, як FIFO та LIFO без необхідності створення нового масиву даних.

Безпосередня трансформація документу у POJO реалізована за допомогою бібліотеки Jackson. Клас LessonDeserializer (рис. 4.10), на прикладі якого буде розглянуто роботу десеріалізаторів, імплементує інтерфейс JsonDeserialier<T> із типом Lesson і відповідає за виділення масиву занять із JSON-дерева.

```java
@Component("lessonDeserializer")
public class LessonDeserializer implements JsonDeserializer<Lesson> {
    private static final Logger logger = LoggerFactory.getLogger(LessonDeserializer.class);
    private final ObjectMapper mapper;

    @Autowired
    public LessonDeserializer(ObjectMapper mapper) {
        this.mapper = mapper;
    }

    @Override
    public Deque<Lesson> deserialize(String json) {
        try {
            JsonNode jsonNode = mapper.readTree(json).get(JSON_TREE_SCHEDULE_OBJECT_NAME);
            return mapper.readValue(jsonNode.toString(), new TypeReference<>() {
            });
        } catch (JsonProcessingException e) {
            logger.error(e.getMessage());
            throw new RuntimeException(ErrorConstants.JSON_PROCESSING_FAILED);
        }
    }
}
```

Рис. 4.10 Логіка парсингу занять у LessonDeserializer

У цьому рисунку ключовим є використання JsonNode, яке є представленням JSON-дерева. Десеріалізатор отримує дані із тексту та перетворює його на Deque<Lesson>, а для збільшення надійності системи було реалізовано механізм перехоплення помилки, який логує виняток із власним аварійним повідомленням, запобігаючи непередбачуваному падінню сервісу.

4.3 Реалізація механізму кешування

Для забезпечення високої продуктивності додатку необхідно зберігати важливі дані для швидкого доступу та мінімізації звернень до бази даних та зовнішніх джерел. Із цією метою було створено систему локального in-memory кешування даних.

Абстрактна модель in-memory сховища

Основою архітектури кешування є абстрактний дженерік клас SimpleMemoryCache<K,V> (рис. 4.11). Оскільки додаток працює у багатопотоковому середовищі, де велика кількість користувачів можуть одночасно

надсилати запити до боту, використання стандартних колекцій є небезпечним. Через потребу у синхронізації застосовано ConcurrentHashMap, використання якої гарантує потокобезпечність без ручного блокування потоків.

```
@Component("simpleCache")
public abstract class SimpleMemoryCache<K, V> {
    protected final ConcurrentHashMap<K, V> localCache = new ConcurrentHashMap<>();

    public V get(K key) {
        return localCache.get(key);
    }

    public void put(K key, V value) {
        localCache.put(key, value);
    }

    public void remove(K key) {
        localCache.remove(key);
    }

    public void clear() {
        localCache.clear();
    }

    public boolean existsByKey(K key) {
        return localCache.containsKey(key);
    }

    public Set<V> getAllValues() {
        return new HashSet<>(localCache.values());
    }

    public int size() {
        return localCache.size();
    }
}
```

Рис. 4.11 Реалізація базового абстрактного класу SimpleMemoryCache

За замовченням усі біни в контексті Spring мають лише одну копію, тому IoC-контейнер гарантує, що записи з будь-якої частини додатку потраплятимуть в один кеш.

Кешування сесій користувачів

При наявності великої кількості користувачів можливе постійне звернення до бази даних задля отримання метаданих користувача (зокрема, його стану, оскільки саме його стан визначає подальший маршрут запиту), що створить вузьке місце у системи та зменшить її швидкодію. Для вирішення цієї задачі створено компонент UserMemoryCache (рис. 4.12), який зберігає дані у локальному кеші.

```java
package rozkladbot.telegram.caching;

import java.time.LocalDateTime;
import org.springframework.stereotype.Component;
import rozkladbot.entities.User;

@Component("simpleUserCache")
public class UserMemoryCache extends SimpleMemoryCache<Long, User> {
  @Override
  public User get(Long key) {
    User user = super.get(key);
    user.setLastInteractionDate(LocalDateTime.now());
    return user;
  }

  @Override
  public void put(Long key, User value) {
    value.setLastInteractionDate(LocalDateTime.now());
    super.put(key, value);
  }
}
```

Рис. 4.12 Реалізація кешу користувачів UserMemoryCache

Як можна побачити на рисунку, UserMemoryCache наслідує абстрактний клас SimpleMemoryCache із параметрами Long (id чату) та User. Оскільки архітектура Telegram-ботів базується на обміні повідомленнями без збереження постійного мережевого з'єднання, система повинна самостійно відстежувати життєвий цикл кожного клієнта. Додатково цей клас перевизначає базові методи батьківського класу get та put, котрі оновлюють дату останньої взаємодії користувача із ботом, що дозволяє ідентифікувати сесії, які зависли у стані бездіяльності (наприклад, якщо користувач давно не користувався ботом). Це дозволяє ефективно запобігати витокам оперативної пам'яті, які неминуче виникли б через накопичення незавершених користувацьких сценаріїв або покинутих діалогів. Фактично даний кеш реалізує стратегію кешування LRU[22] (Least recently used), який видаляє з кешу найбільш старі записи, якщо пам’яті не вистачає (проте краще не чекати поки пам’ять вичерпається і видаляти записи як тільки вони стануть невалідними). Для автоматизації цього процесу в системі передбачено фоновий механізм очищення,

який періодично сканує колекцію та превентивно інвалідує об'єкти, що перевищили дозволений час бездіяльності (Time-To-Live). Це проактивне вивільнення ресурсів суттєво знижує навантаження на збирач сміття (Garbage Collector) віртуальної машини Java, мінімізуючи мікрозатримки у роботі додатка. Отже, розроблена кеш-модель вдало поєднує принципи класичного витіснення LRU із жорстким контролем часу життя сесії, що гарантує стабільно низьке споживання пам'яті сервером навіть при пікових навантаженнях.

Автоматичне очищення неактивних сесій

Щоб запобігти нераціональному використанню пам’яті, було розроблено компонент CacheCleaner (рис. 4.13), який виконує роль збирача сміття для локального кешу сервісу.

Для автоматизації процесу використовується анотація @Scheduled з фреймворку Spring, яка ініціює перевірку сесій за заданим CRON-виразом. Логіка валідації інкапсульована в інтерфейс Predicate<User>, який перевіряє чи перевищує час бездіяльності користувача встановлений ліміт (базовий ліміт становить 4 доби). Важливо зауважити, що цей метод виконуються в окремому потоці, який створюється за допомогою анотації @Async, і не заважає основному потоку користувача.

```
  @Async
  @Scheduled(cron = AppConstants.USER_CACHE_CLEANING_CRON, zone =
AppConstants.APPLICATION_TIME_ZONE)
  public void cleanUserCache() {
    for (User user : userMemoryCache.getAllValues()) {
      if (userPredicate.test(user)) {
        messageSender.sendMessage(user, LAST_INTERACTION_TIME_WAS_TOO_LONG_AGO, null, false, null);
        userMemoryCache.remove(user.getId());
        threadMemoryCache.remove(user.getId());
      }
    }
  }
}
```

Рис. 4.13 Фоновий процес очищення кешу

Проте цей клас не тільки звільняє ресурси видаляючи дані із userMemoryCache та зупиняє відповідні потоки, що були виділені під користувача сервісу, а й дбає про досвід користувача, повідомляючи його перед завершенням простроченої сесії.

Кешування розкладів

Однак найважливішими даними, якими керує система є навчальні розклади, тому їх кешування має бути найбільш надійним. Оскільки зовнішні джерела можуть мати обмеження по кількості запитів або просто тимчасово недоступними, було прийнято рішення реалізувати багаторівневу систему кешування, що поєднує in-memory підхід із персистентним (база даних).

Для забезпечення довгострокової інформації про розклади було спроєктовано сутність ScheduleCache (рис. 4.14), який на відміну від in-memory кешу зберігається у БД та зберігається навіть після перезапуску сервера.

```
@Entity
@Table(name = "schedules_cache")
@Data
@NoArgsConstructor
public class ScheduleCache {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private long id;

  @Column(name = "group_id")
  private long groupId;

  @Enumerated(EnumType.STRING)
  @Column(name = "schedule_type")
  private ScheduleType scheduleType;

  @Column(name = "content", columnDefinition = "TEXT")
  private String content;

  @UpdateTimestamp
  @Column(name = "update_time")
  private LocalDateTime updateTime;

  public ScheduleCache(long groupId, ScheduleType scheduleType, String content) {
    this.groupId = groupId;
    this.scheduleType = scheduleType;
    this.content = content;
  }
}
```

Рис. 4.14 Структура сутності ScheduleCache

Ключовою особливістю цієї сутності є поле updateTime. Вона зберігає інформацію про останнє оновлення розкладу та використовується для інвалідації та оновлення кешу відносно поточного запиту користувача.

Централізація операцій із персистентним кешем реалізована у сервісі ScheduleCacheService. Він виконує роль сервісу між репозиторієм та основною бізнес-логікою, дозволяючи фільтрувати розклади за їх типом (THIS_WEEK або NEXT_WEEK), як можна побачити на рис. 4.15.

```
  @Override
  @Cacheable(value = "schedules", key = "#groupId + '_' + #scheduleType.name()", unless = "#result
== null")
  public ScheduleCache findByGroupIdAndScheduleType(long groupId,
      ScheduleType scheduleType) {
    return scheduleCacheRepo.findByGroupIdAndScheduleType(groupId, scheduleType);
  }
```

Рис. 4.15 Фрагмент коду ScheduleCacheService, який відповідає за пошук розкладу за типом та номером групи

Основний сервіс розкладу ScheduleService реалізує патерн «Cache aside», тобто додаток самостійно керує даними: спочатку шукає їх у in-memory кеші, а якщо такого немає, то завантажує дані з бази даних. Якщо у базі даних теж немає потрібної інформації або вони більше не актуальні, то виконується мережевий запит через WebRequestService до зовнішнього джерела, супутньо оновлюючи дані у БД. Реалізацію цього патерну показана на рис. 4.16.

```
return CompletableFuture.supplyAsync(() -> {
  ScheduleType type = THIS_WEEK.equals(mode) ? ScheduleType.THIS_WEEK : ScheduleType.NEXT_WEEK;
  Optional<ScheduleCache> cache = scheduleCacheService.findByGroupIdAndScheduleType(
      Long.parseLong(params.get(AppConstants.GROUP_ID)),
      type);
  if (cache.isPresent() && StringUtils.isNotBlank(cache.get().getContent())) {
    return cache.get().getContent();
  }
  try {
    String content = webRequestService.makeRequest(params, ApiEndpoints.API_SCHEDULE);
    scheduleCacheService.updateContent(
        Long.parseLong(params.get(GROUP_ID)),
        type,
        content
    );
    return content;
  } catch (IOException | URISyntaxException | InterruptedException e) {
    logger.error(REQUEST_CREATION_FAILED);
    throw new RequestCreationFailedException(REQUEST_CREATION_FAILED);
  }
}
```

Рис. 4.16 Реалізація вибору кешу у ScheduleService

Проте іноді можливі випадки, коли база даних перенавантажена і не приймає нових запитів. Для забезпечення кращої відмовостійкості додатково було реалізовано підсистему файлового кешування. Її головна мета - забезпечення виконання запитів навіть якщо сторонні джерела або БД недоступні.

Реалізація цього механізму покладена на ScheduleCacheFileWriter (рис. 4.17), який автоматично записує дані за наданим кроном один раз на день (оскільки навантаження на сервіс вночі мінімальне, то запис починається у 3 години ранку).

Задля додаткової оптимізації сервіс не зберігає файли для усіх груп, а тільки тих, що закріплені за дійсними користувачами.

```
    @Async
    @Scheduled(cron = AppConstants.SCHEDULE_DUMP_CRON, zone = AppConstants.APPLICATION_TIME_ZONE)
    public void dumpSchedule() {
        dumpSchedule(false);
    }

    @Async
    public void dumpSchedule(boolean isForced) {
        Set<Long> alreadyDumpedGroups = new HashSet<>();
        logger.info(SCHEDULE_DUMPING_BEGAN);
        groupService.getAllGroupsThatAssignedToUsers().forEach(group -> {
            long groupNumber = group.getGroupId();
            if (groupNumber == 0 || alreadyDumpedGroups.contains(groupNumber)) return;
            try {
                logger.info(SCHEDULE_DUMPING_BEGAN_FOR_GROUP, group.getGroupId(), group.getName());
                Path directoryPath = Paths.get(AppConstants.GROUP_SCHEDULES_FOLDER_NAME);
                String fileName =
AppConstants.THIS_WEEK_SCHEDULE_FILE_NAME.formatted(group.getName(), group.getGroupId());
                prepareForWriting(directoryPath, fileName, group,
                        DateUtils.getStartOfWeek(DateUtils.getTodayDateString()),
                        DateUtils.getStartOfWeek(DateUtils.getTodayDateString()).plusDays(7),
                        isForced);
                fileName = AppConstants.NEXT_WEEK_SCHEDULE_FILE_NAME.formatted(group.getName(),
group.getGroupId());
                prepareForWriting(directoryPath, fileName, group,
                        DateUtils.getStartOfWeek(DateUtils.getTodayDateString()).plusDays(7),
                        DateUtils.getStartOfWeek(DateUtils.getTodayDateString()).plusDays(14),
                        isForced);
                alreadyDumpedGroups.add(groupNumber);
            } catch (IOException | URISyntaxException | InterruptedException e) {
                logger.error(SCHEDULE_DUMPING_FAILED, group, e);
            }
        });
    }
```

Рис. 4.17 Логіка файлового завантаження даних

Безпосередня взаємодія зі сховищем абстрагована в окремі компоненти LocalFileWriter та LocalFileReader. Для уникання непотрібних операцій вводу-виводу, перед записом данних відбувається валідація файлу.

Метод checkIfAlreadyWritten (рис. 4.18) робить декілька перевірок: чи є оновлення примусовим (адмін-панель дозволяє примусово оновлювати будь-які дані додатку), вік файлу (файл не перезаписується, якщо він був перезаписаний менше, ніж день тому), а також поточний день тижня. Примусове оновлення файлів завжди відбувається у понеділок, гарантує наявність свіжих даних на два навчальних тижні, що забезпечує надійний локальний захист від відключення зовнішньої та внутрішньої інфраструктури.

```
    public boolean checkIfAlreadyWritten(Path directoryPath, String fileName, boolean isForced) {
        Path filePath = directoryPath.resolve(fileName);
        try {
            return isForced || DateUtils.instantToLocalDate(
                            Files.getLastModifiedTime(filePath).toInstant())
                    .isBefore(DateUtils.getToday().minusDays(1)) ||

AppConstants.MONDAY.equalsIgnoreCase(DateUtils.getFullDayName(DateUtils.getToday()));
        } catch (IOException exception) {
            logger.info(SCHEDULE_CREATE_IF_NOT_EXISTS, fileName);
            return true;
        }
    }
```

Рис. 4.18 Алгоритм перевірки дати останнього запису

4.4 Реалізація взаємодії користувача із ботом

У даному підрозділі розглядається практичне втілення користувацького інтерфейсу для взаємодії користувача із сервісом, а також бізнес-логіки обробки ключових запитів ScheduleBot, які забезпечують повний цикл взаємодії студента з інформацією про розклади.

Ініціалізація користувача та старт сесії

Точкою входу для користувача є команда /start, яке ініціює створення сесії. Відповідно до офіційної специфікації Telegram API, саме ця базова директива виступає універсальним тригером для активації нового діалогу та первинної ідентифікації клієнта. Під час надходження об’єкта Update маршрутизатор Router перевіряє наявність ідентифікатора чату у системі (після перезапуску пошук проводиться у БД, а для наступних запитів - у in-memory кеші UserMemoryCache). Така дворівнева стратегія перевірки наявності користувача дозволяє суттєво мінімізувати кількість дорогих I/O транзакцій до реляційної бази даних, забезпечуючи миттєву диспетчеризацію повідомлень. Якщо користувач звертається до бота вперше, то система автоматично створює для нього стандартний профіль у базі даних, викликаючи фабричний метод User.getDefaultUser (рис. 4.19) та присвоює йому початковий стан AWAITING_GREETINGS. Цей згенерований базовий профіль одразу інкапсулює у собі дефолтні налаштування сповіщень та мінімальний рівень прав доступу (UserRole), гарантуючи безпеку додатка на етапі

ініціалізації. Переведення сесії у стан очікування привітання виступає стартовою точкою в життєвому циклі кінцевого автомата (State Machine), який керує процесом реєстрації. Надалі маршрутизатор делегує обробку цього стану спеціалізованому компоненту NewUsersHandler, що розпочне інтерактивний покроковий збір академічних параметрів студента.

```
public static User getDefaultUser(Update update, long chatId) {
  User user = new User();
  user.setUsername(getCorrectName(update));
  user.setId(chatId);
  user.setLastSentMessageId(update.getMessage().getMessageId());
  user.setUserState(UserState.AWAITING_GREETINGS);
  Group group = new Group();
  Institute institute = new Institute();
  Faculty faculty = new Faculty();
  faculty.setInstitute(institute);
  group.setFaculty(faculty);
  user.setGroup(group);
  return user;
}
```

Рис. 4.19 Фрагмент коду фабричного методу

Подальша обробка делегується компоненту NewUsersHandler, який надсилає привітальне повідомлення з інструкціями та автоматично переводить стан користувача у стан AWAITING_INSTITUTE, розпочинаючи процес реєстрації.

Оскільки розклад має складну ієрархічну структуру (університет → інститут → курс → група (рис. 4.20, 4.21, 4.22, 4.23 відповідно), то процес закріплення студента за групою реалізовано у вигляді багатокрокового майстра налаштувань всередині RegistrationHandler.

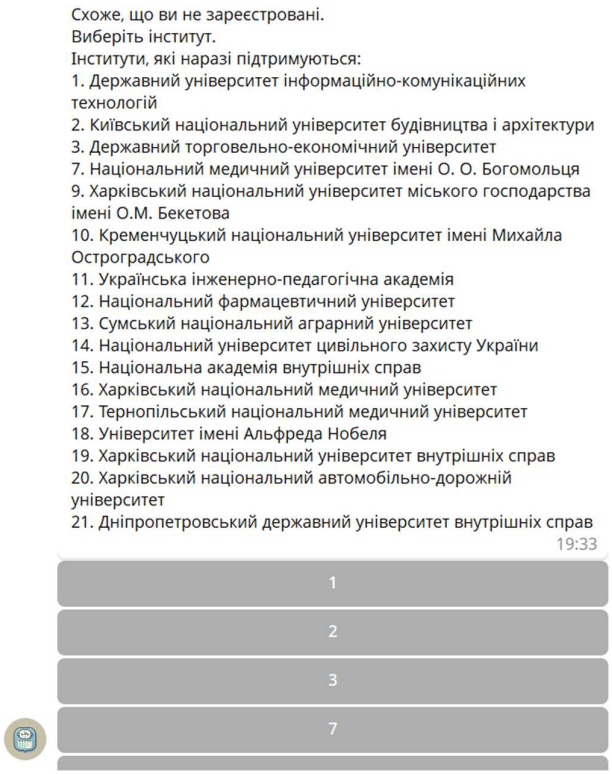


Рис. 4.20 Сторінка вибору університету

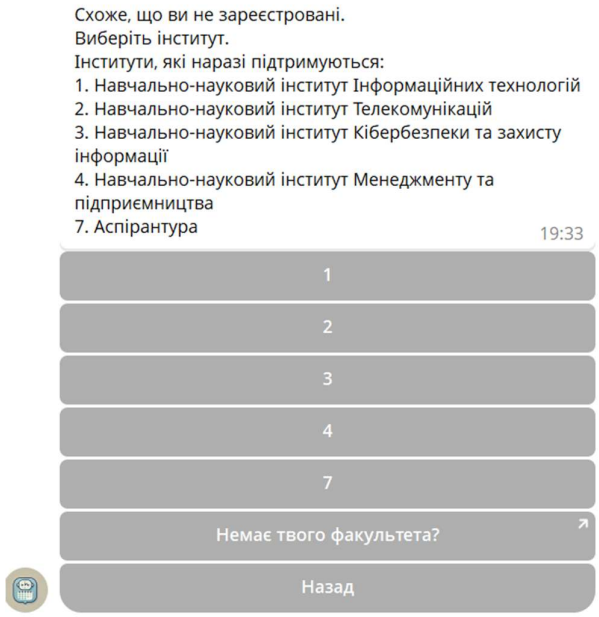


Рис. 4.21 Сторінка вибору інституту

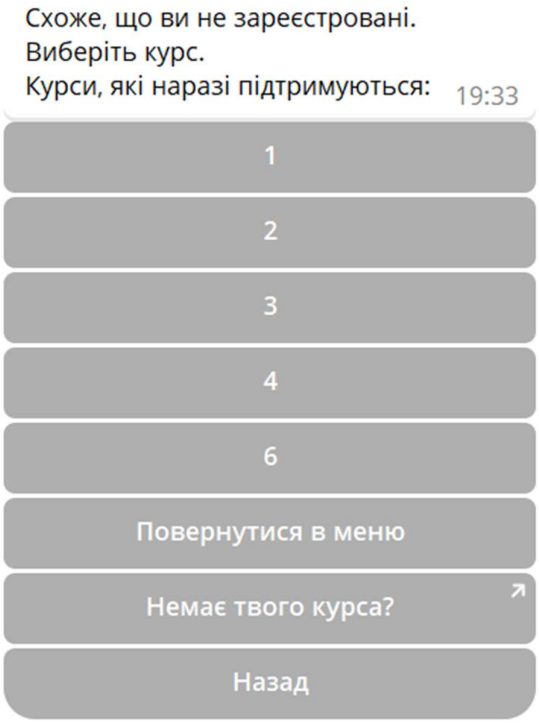


Рис. 4.22 Сторінка вибору курсу

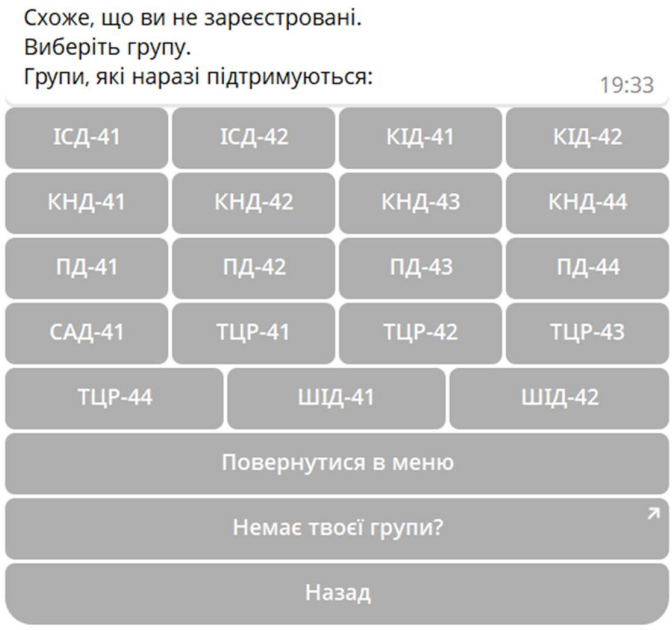


Рис. 4.23 Сторінка вибору групи

Після того як користувач обрав потрібну групу, йому пропонується підтвердити коректність даних (рис. 4.24).

Рис. 4.24 Сторінка підтвердження даних

Якщо студент неправильно ввів дані, то він має змогу повторити процес реєстрації, натиснувши кнопку «Повторити спробу» (рис. 4.25).

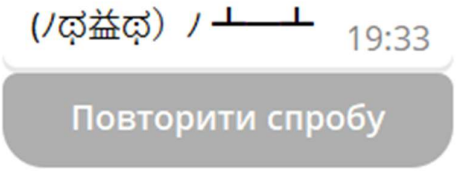


Рис. 4.25 Сторінка невдалої реєстрації

Якщо студент підтверджує правильних даних, то інформація про нього заноситься у БД та у in-memory кеш, а його стан змінюється на IDLE.

Головне меню та навігація

Після успішного завершення процесу реєстрації або закінчення минулого запиту система переводить користувача у стан IDLE, відкриваючи доступ до основного функціоналу (рис. 4.26). Переведення кінцевого автомата у цей базовий стан є критично важливим механізмом стабілізації сесії, який гарантує, що користувач не залишиться заблокованим у проміжних гілках діалогу після виконання, скасування або виникнення помилки під час попередніх дій. Точкою взаємодії у стані IDLE є головне навігаційне меню, формування якого лежить на компоненті MainMenuHandler. Архітектурною особливістю цього модуля є імплементація політики динамічного розмежування доступу на основі ролей (Role-Based Access Control). Під час відмальовування інтерфейсу система перевіряє рівень прав поточного клієнта (UserRole) та автоматично розширює стандартний перелік кнопок блоком службових команд, якщо ідентифіковано сесію

адміністратора. Для забезпечення максимальної ергономічності, графічне представлення меню будується виключно на базі Inline-клавіатур за допомогою застосування спеціалізованого патерну фабрики у класі KeyBoardFactory. Такий підхід до побудови клієнтського інтерфейсу повністю усуває необхідність ручного введення текстових директив, мінімізує ймовірність синтаксичних помилок та дозволяє студенту отримати актуальний розклад у суворій відповідності до ергономічного «правила трьох натискань».

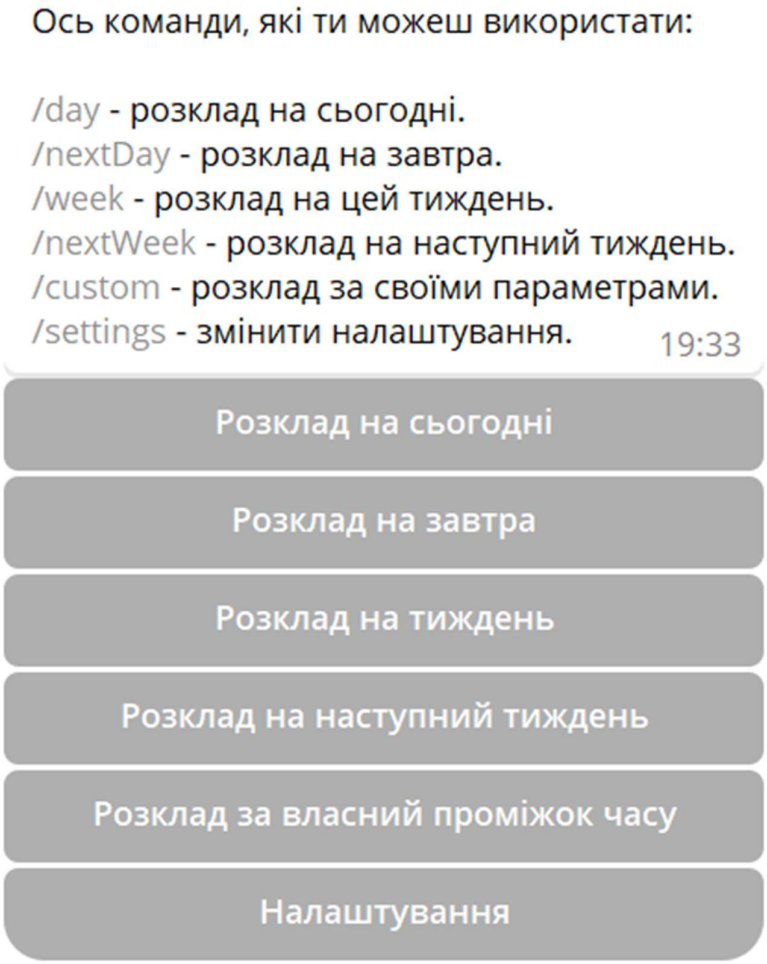


Рис. 4.26 Головне меню бота

Архітектурною особливістю цього модуля є те, що він розмежовує доступ відповідно до ролі користувача. Під час виклику методу sendMenu система

перевіряє його права і якщо було ідентифіковано права адміністратора (рис. 4.27), до стандартного набору команд додаються службові.

```
    public void handleRequest(HandlerContext ctx) {
        messageSender.sendMessage(
                ctx.getUser(),
                (ctx.getUser().getUserRole().equals(UserRole.ADMIN) ?
                        BotMessageConstants.AVAILABLE_USER_COMMANDS +
BotMessageConstants.AVAILABLE_ADMIN_COMMANDS :
                        BotMessageConstants.AVAILABLE_USER_COMMANDS),
                KeyBoardFactory.getCommandsList(),
                ctx.isOverrideMessage(), ctx.getUpdate());
        ctx.getUser().setUserState(UserState.IDLE);

 ctx.getUser().setLastSentMessageId(MessageUtils.getCorrectMessageIdWithOffset(ctx.getUpdate()));
    }
```

Рис. 4.27 Фрагмент коду, який відповідає за динамічне створення головного меню

Для забезпечення більшої зручності навігацію, використовуються inline-клавіатури із кнопками, що побудована на основі класа-фабрики KeyBoardFactory, що дозволяє не прописувати команди текстом, прискорюючи навігацію.

Вивід розкладу

Основа бізнес-логіки генерації та формування розкладів зосереджена у класі ScheduleFetchHandler, який імплементує вищезгаданий інтерфейс HandlerStrategy. Звертаючись до сервісу ScheduleService, хендлер формує повідомлення залежно від запиту користувача (розклад на день, тиждень, тощо).

Критично важливим технічним рішенням є алгоритм розбиття повідомлень на частини. Оскільки обмеження Telegram API не дозволяють надіслати більше 4096 символів, потреба у чанкуванні довгих повідомлень (наприклад, розкладів за власний проміжок часу, який не є обмеженим) є життєвонеобхідною.

Алгоритм реалізовано через ітерацію циклом do-while (рис. 4.28), котрий розбиває дані на блоки по сім днів і надсилає їх користувачу у вигляді окремих повідомлень (рис. 4.29). Додатково реалізовано механізм перехоплення помилок, який гарантує формування зрозумілого повідомлення про збій у системі та безпечно скидає стан користувача до базового стану IDLE, запобігаючи блокуванню процесу взаємодії.

```
case AWAITING_CUSTOM_SCHEDULE -> {
  scheduleTable = scheduleService.getScheduleWithCustomParameters(user, update);
  int scheduleTableSize = scheduleTable.getDays().size();
  int i = 0;
  do {
    messageSender.sendMessage(
        user,
        scheduleTableNormalizer.splitBigTableIntoSmall(scheduleTable).toString(),
        null,
        i == 0,
        update);
    i++;
  } while (!scheduleTable.getDays().isEmpty());
  messageToSend = BotMessageConstants.BACK_TO_MENU;
  isCustom = true;
  user.setUserState(UserState.IDLE);
}
```

Рис. 4.28 Алгоритм розбиття довгих даних

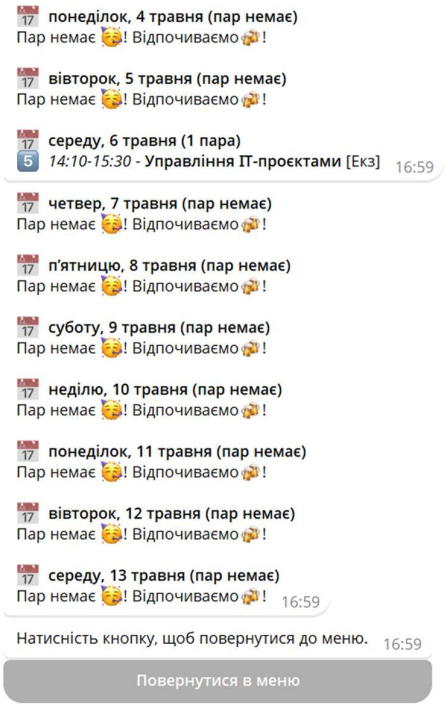


Рис. 4.29 Приклад чанкування розкладу за власний час

Налаштування користувача

Система має особистий кабінет користувача (рис. 4.30), де він може побачити дані про себе, а також змінити налаштування через компонент SettingsHandler. У цьому розділі студент отримує можливість змінити свої дані про університет,

отримати техпідтримку від розробника, а також керувати налаштуваннями автоматичної вечірньої розсилки розкладів на наступний день.

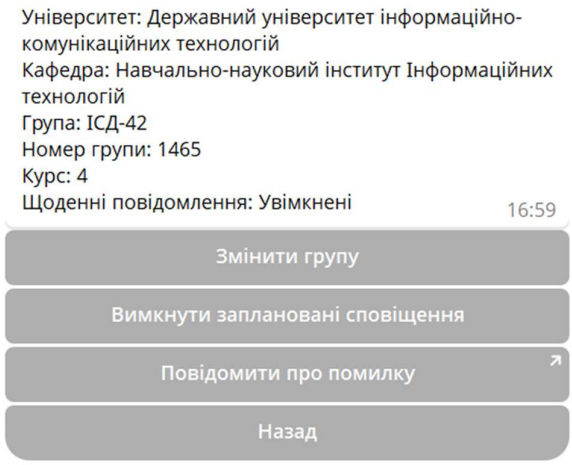


Рис. 4.30 Меню налаштувань користувача

Підсистема автоматичної розсилки розкладів

Для збільшення зручності користування ботом, його функціонал було розширено підсистемою вечірнього автоматичного розсилання розкладів на наступний день. Головним компонентом, який керує даним функціоналом є метод broadCastAndPinTomorrowSchedule (рис. 4.31) класу ScheduleBroadcastService, який використовує планувальник Spring та асинхронну роботу за допомогою анотації @Async. Планувальник допомагає автоматично запускати процес розсилання розкладів у чітко визначений час, який конфігурується у константах додатку.

```
@Scheduled(cron = AppConstants.BROADCASTING_CRON, zone = AppConstants.APPLICATION_TIME_ZONE)
public void broadcastAndPinTomorrowSchedule() {
    logger.info(LoggingConstants.BEGIN_BROADCAST_MESSAGE);
    try {
        Set<User> users = userCache.getAllValues();
        CompletableFuture<?>[] futures = users.stream()
                .filter(User::isBroadcasted)
                .map(user -> CompletableFuture.runAsync(() -> processUser(user.getId(), user)))
                .toArray(CompletableFuture[]::new);
        CompletableFuture.allOf(futures).join();
    } finally {
        logger.info(LoggingConstants.END_BROADCAST_MESSAGE);
    }
}
```

Рис. 4.31 Фрагмент коду, який відповідає за ініціацію розсилання розкладів

Проте важливо зазначити, що даний компонент реалізовує додатковий функціонал, який покращує користувацький досвід, а саме автоматичне закріплення нового повідомлення про розклад (минуле повідомлення, якщо воно існує, відкріплюється).

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи було розроблено та впроваджено автоматизовану систему моніторингу навчального розкладу ScheduleBot для студентів університетів. На основі проведеного аналізу існуючих рішень та результатів опитування цільової аудиторії було виявлено низьку ефективність традиційних веб-ресурсів для мобільних пристроїв, що зумовило вибір Telegram як основної платформи. Це дозволило успішно реалізувати принципи адаптивності інтерфейсу та забезпечити швидкий доступ до даних згідно з ергономічним «правилом трьох натискань».

Технічна реалізація серверної частини на базі екосистеми Java та фреймворку Spring Boot гарантує високу масштабованість та простоту підтримки програмного продукту. Завдяки впровадженню патернів проектування «Фабрика» та «Стратегія», була побудована гнучка архітектура для динамічної маршрутизації запитів і управління сесіями залежно від поточного стану та ролі користувача. Особливу увагу було приділено надійності системи, що досягається через комплексну багатошарову підсистему кешування. Поєднання обробки даних в оперативній пам’яті з автоматичним очищенням неактивних сесій, збереження розкладів у базі даних та фоновий дампінг на жорсткий диск дозволяють боту стабільно функціонувати навіть у разі тимчасової недоступності зовнішніх джерел даних.

Оптимізація користувацького досвіду була досягнута завдяки відмові від ручного введення команд на користь динамічної генерації Inline-клавіатур через спеціалізовану фабрику, а впровадження підсистеми автоматичної розсилки перетворило бота на проактивний інструмент персоналізованого інформування. Результати тестування фокус-групою та зібраний зворотний зв’язок підтвердили повну відповідність системи функціональним і нефункціональним вимогам, засвідчивши її стабільність та готовність до експлуатації в освітньому процесі.

Закладена архітектурна база забезпечує значний потенціал для подальшого

розвитку проєкту, зокрема для інтеграції нових стратегій парсингу даних та розширення функціональних можливостей системи.

ПЕРЕЛІК ПОСИЛАНЬ

1. Веб-додаток для отримання розкладу Skedy. [електронний ресурс] - Режим доступу: https://skedy.yacode.dev/
2. Веб-сайт для отримання розкладу у вигляді HTML-Таблиць. НАУ. [електронний ресурс] - Режим доступу: https://portal.nau.edu.ua/schedule/group?id=295
3. Working with JSON. Mozilla. [електронний ресурс] - Режим доступу: https://developer.mozilla.org/enUS/docs/Learn_web_development/Core/Scripting/JSON
4. Anatomy of the DOM. Mozilla. [електронний ресурс] - Режим доступу: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Anatomy_of_the_DOM
5. What is OCR. Amazon. [електронний ресурс] - Режим доступу: https://aws.amazon.com/what-is/ocr/
6. Telegram Bot API documentation. Telegram. [електронний ресурс] - Режим доступу: https://core.telegram.org/
7. Working with Updates. Telegram. [електронний ресурс] - Режим доступу: https://core.telegram.org/api/updates
8. Polling vs Webhooks. [електронний ресурс] - Режим доступу: https://www.prakashbhandari.com.np/posts/polling-vs-webhooks/
9. Розподілення кількості ботів по мовах програмування за ключовими словами “Telegram bot”. GitHub. [електронний ресурс] - Режим доступу: https://github.com/search?q=telegram+bot&type=repositories
10. Spring Framework official page. Spring. [електронний ресурс] - Режим доступу: https://spring.io/projects/spring-framework
11. Spring Web reference. Spring. [електронний ресурс] - Режим доступу: https://docs.spring.io/spring-boot/reference/web/index.html
12. Apache Tomcat overview. Apache Foundation. [електронний ресурс] - Режим доступу: https://tomcat.apache.org/

13. Spring Data Reference. [електронний ресурс] - Режим доступу: https://docs.spring.io/spring-data/jpa/reference/index.html
14. Three clicks rule. [електронний ресурс] - Режим доступу: https://www.uxtweak.com/ux-glossary/three-click-rule/
15. Data Transfer Object (DTO). [електронний ресурс] - Режим доступу: https://en.wikipedia.org/wiki/Data_transfer_object
16. Model-View-Controller Pattern. [електронний ресурс] - Режим доступу: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93viewmodel
17. SOLID principles. [електронний ресурс] - Режим доступу: https://en.wikipedia.org/wiki/SOLID
18. DRY principle. [електронний ресурс] - Режим доступу: https://en.wikipedia.org/wiki/Don%27t_repeat_yourself
19. Redis. [електронний ресурс] - Режим доступу: https://ru.wikipedia.org/wiki/Redis
20. What is POJO class? [електронний ресурс] - Режим доступу: https://www.baeldung.com/java-pojo-class
21. Generic Types. Java documention. [електронний ресурс] - Режим доступу: https://docs.oracle.com/javase/tutorial/java/generics/types.html
22. What is LRU. [електронний ресурс] - Режим доступу: https://en.wikipedia.org/wiki/Cache_replacement_policies#LRU

ДОДАТКИ

Таблиця 1

Таблиця фунціональних вимог до системи Telegram-бота для отримання розкладів

ID	Назва функції	Опис функції	Вхідні дані	Вихідні дані	Передумови	Післяумови
FR1	Ініціалізація (команда /start)	Запуск чат-бота, відображення привітання та меню	Команда /start	Привітальне повідомлення, меню реєстрації	Чат-бот доступний	Користувач може зареєструватися у чат-боті
FR2	Реєстрація	Можливість реєстрації у чат-боті для запам’ятовування даних користувача для подальшого кешування	Меню реєстрації	Меню команд бота	Користувач не зареєстрований у сервісі	Користувач може взаємодіяти із функціями чат-бота
FR3	Розклад на сьогодні	Виведення розкладу на поточний день	Команда (/day) або кнопка	Розклад за поточний день	Користувач зареєстрований	Користувач отримує потрібні дані
FR4	Розклад на поточний тиждень	Виведення розкладу на поточний тиждень	Команда (/week) або кнопка	Розклад за поточний тиждень	Користувач зареєстрований	Користувач отримує потрібні дані

Продовження таблиці 1

FR5	Розклад на наступний день	Виведення розкладу на день після поточного	Команда (/nextday) або кнопка	Розклад за день після поточного	Користувач зареєстрований	Користувач отримує потрібні дані
FR6	Розклад на наступний тиждень	Виведення розкладу на тиждень після поточного	Команда (/nextweek) або кнопка	Розклад за тиждень після поточного	Користувач зареєстрований	Користувач отримує потрібні дані
FR7	Розклад за обрану дату	Виведення даних на необхідний день для користувача, що може не бути прив’язаним до поточного дня або тижня	Команда (/custom) або кнопка	Розклад за потрібну дату (може бути день, тиждень, місяць)	Користувач зареєстрований	Користувач отримує потрібні дані
FR8	Кешування	Локальне збереження даних, пов’язаних із розкладом або користувачем для швидкого доступу	Дані з API/взаємодії із користувачем	Дані у кеші	Отримані дані з API/від користувача	Кеш оновлено

Продовження таблиці 1

FR9	Оновлення даних	Оновлення даних з API	Запит по таймеру	Актуальні дані	API доступне	Кеш оновлено
FR10	Обробка помилок	Обробка помилок, пов'язаних із API, мережею, даними або діями користувача	Помилка	Повідомлення про помилку або коригування вхідних даних	Виникла помилка	Користувач проінформований за допомогою детальної причини помилки
FR11	Відсутність занять	Повідомлення про відсутність занять	Запит	Повідомлення «Немає занять»	Дані відсутні	Користувач отримує відповідь
FR11	Відсутність занять	Повідомлення про відсутність занять	Запит	Повідомлення «Немає занять»	Дані відсутні	Користувач отримує відповідь
FR12	Меню навігації	Відображення списку доступних команд та кнопок для зручної взаємодії	Дія користувача	Команди (Reply та Inline-buttons)	Чат-бот активний	Користувач обирає дію
FR13	Обробка команд	Обробка дій користувача	Команда	Відповідь системи	Бот запущений	Команда оброблена

Продовження таблиці 1

FR14	Форматування відповіді	Форматування даних з API та внутрішніх відповідей у форматі, зручному для користувача	Дані розкладу/внутрішні дані	Відформатоване повідомлення	Дані отримані	Інформація легка для сприйняття
FR15	Розбиття повідомлень	Чанкування довгих (> 4096 символів) повідомлень (обмеження телеграм)	Великий обсяг даних (> 4096 символів)	Декілька повідомлень	Ліміт символів перевищено	Дані доставлені повністю
FR16	Логування	Запис подій та помилок	Події системи	Логи	Система працює	Доступні дані для аналізу

Таблиця 2

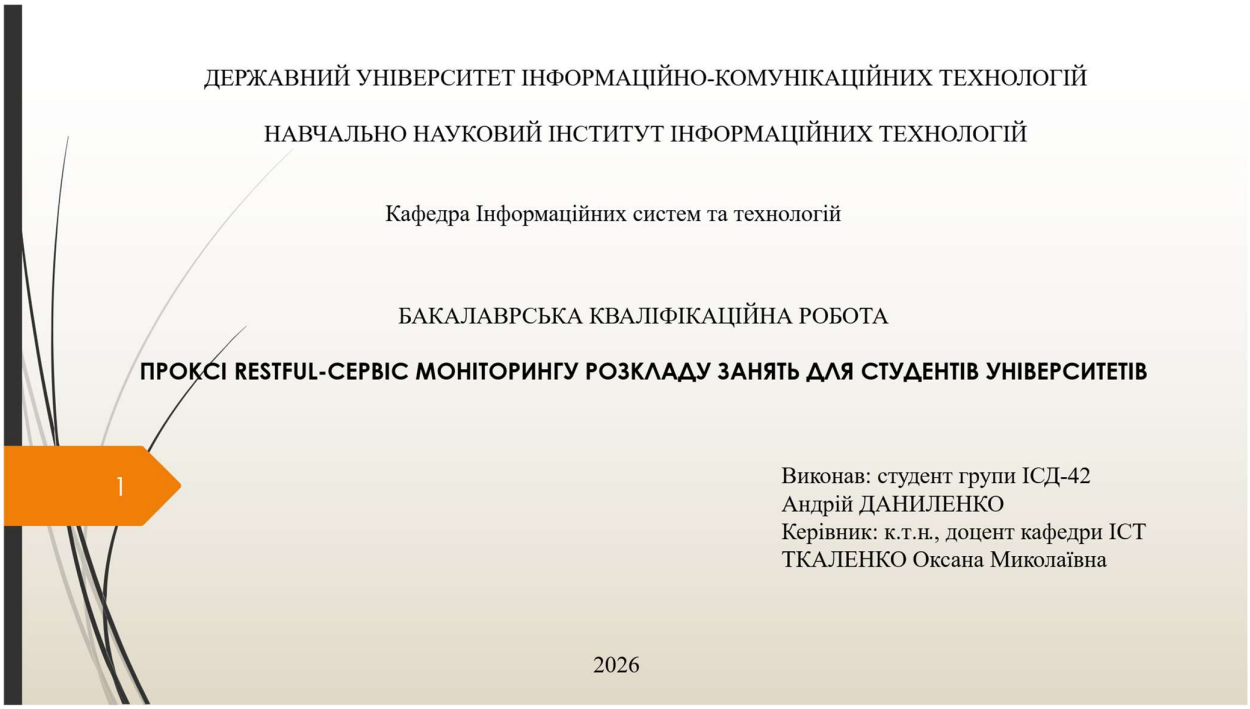
Таблиця нефунціональних вимог до системи Telegram-бота для отримання розкладів

ID	Тип	Вимога	Крітерій	Опис
NFR1	Швидкодія	Час відповіді (кеш)	<=500мс	Відповідь при наявності даних у кеші
NFR2	Швидкодія	Час відповіді (API)	<=2с	Відповідь при звертанні до API

Продовження таблиці 2

NFR3	Швидкодія	Час обробки на сервері	<=200мс	Без урахування мережевих затримок
NFR4	Масштабованість	Одночасні користувачі	>=1000 користувачів	Система має витримувати навантаження
NFR5	Масштабованість	Пропускна здатність	>=100 запитів/сек	Система має витримувати навантаження
NFR5	Надійність	Обробка помилок	100% критичних помилок	Усі критичні помилки мають бути оброблені
NFR6	Надійність	Fallback на кеш	Реалізовано	Використання кешу при недоступності API
NFR7	Безпека	Використання HTTPS	Обов’язково	Зашифровані з’єднання
NFR8	Безпека	Захист службових змінних	Не зберігаються у коді	Усі службові змінні винесені у env-файл
NFR9	Безпека	Валідація даних	100% вхідних даних	Перевірка усіх запитів
NFR10	UX	Кількість дій до результату	<=3 кроків	Мінімізація взаємодії для отримання результату

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ (ПРЕЗЕНТАЦІЯ)



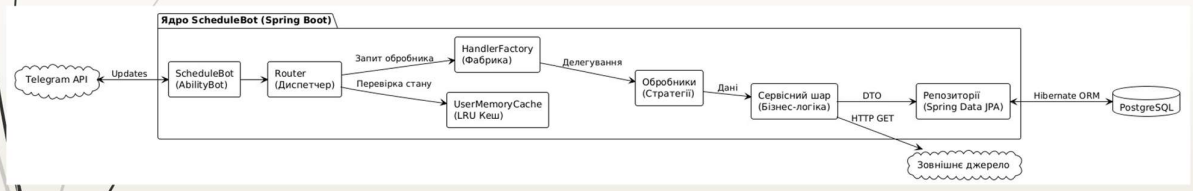
3

Завдання

- Дослідити предметну область, існуючі програмні рішення та особливості автоматизації моніторингу розкладу занять;
- Обґрунтувати вибір технологічного стека (Java, Spring, PostgreSQL) та архітектурних патернів проєктування для забезпечення масштабованості системи;
- Розробити архітектуру системи, UML-моделі внутрішньої структури даних та алгоритми парсингу JSON-документів;
- Реалізувати та дослідити відмовостійкий механізм багаторівневого кешування даних (рантайм кеш, БД, файлове кешування).
- Виконати програмну реалізацію серверної частини бота, інтегрувати її з Telegram API.

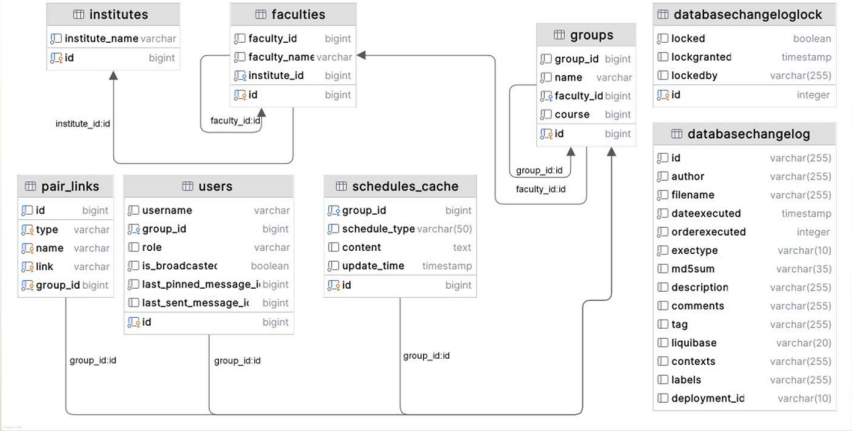
4

Архітектура системи



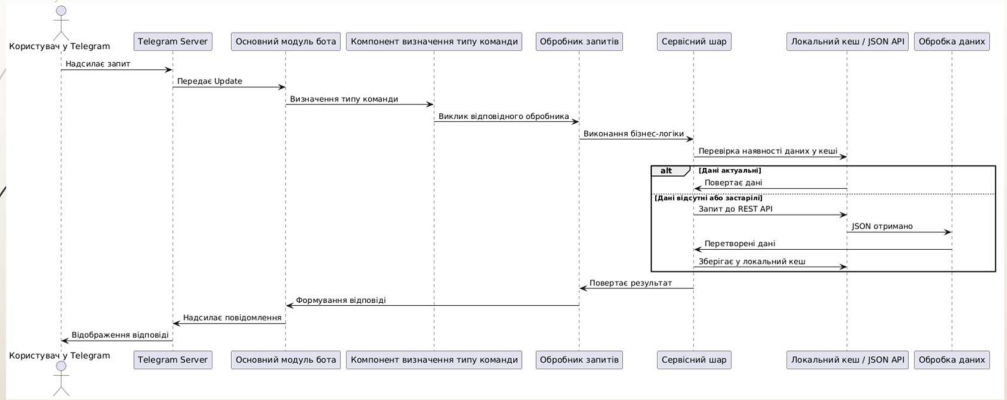
5

Архітектура бази даних



6

Діаграма послідовності взаємодії користувача із системою



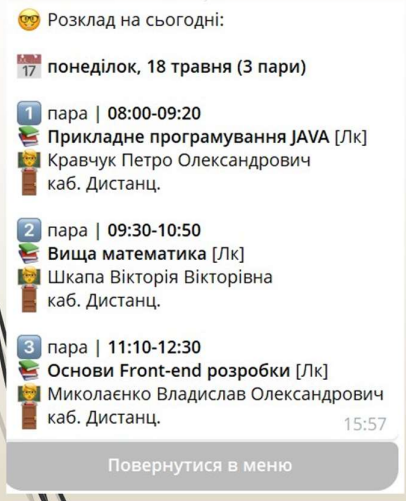
7

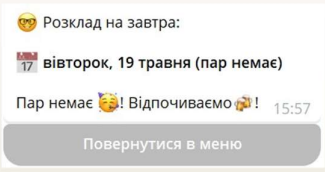
Реалізація

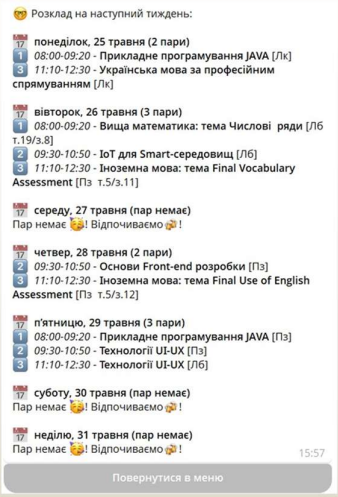
Для реалізації бек-енду було використано мову програмування Java та фреймворки Spring, надає абстракції для взаємодії із мережевими сервісами та JSON-документами, а також Hibernate, який дає можливість використовувати ORM у проєкті. Формат вхідних даних представлений у форматі JSON, який кешується у БД або в окремому файлі. У відповідь на запит користувача система генерує нормалізований розклад, представлений внутрішньою моделлю системи та представляє його у вигляді повідомлення у месенджері Telegram.

8

Результати тестування

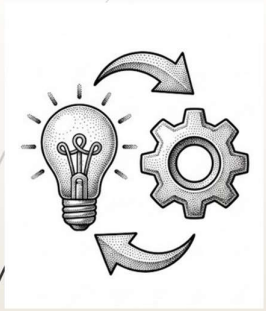






9

Перспективи впровадження



Перспективи впровадження створеного додатку полягають у його інтеграції зі сторонніми джерелами закладів освіти для автоматизованого формування академічних розкладів, що дозволить зменшити навантаження на викладачів, адміністрацію та самих користувачів, підвищити точність отриманої інформації та забезпечити її безперебійне отримання.

10

Сценарії впровадження

Система ScheduleBot інтегрується з існуючими інформаційними серверами ЗВО (зокрема відкритим API розкладу), забезпечуючи автоматизований збір, парсинг та миттєву трансляцію академічного графіка безпосередньо у Telegram-середовище студентів. Це дозволить здобувачам освіти швидко отримувати персоналізовані розклади на будь-який період, а також підключити проактивні push-сповіщення про щоденні заняття або раптові зміни. Гнучка модульна архітектура бота підтримуватиме легку адаптацію та масштабування під структуру інших університетів або факультетів. Крім того, впровадження системи багаторівневого кешування суттєво знизить пікове ранкове навантаження на головні веб-ресурси закладу, а наявність рольової моделі дозволить адміністрації (деканатам) використовувати бота як надійний канал для екстреного інформування студентів.

11

Апробація результатів та публікації

1. Даниленко А. Є. «Використання штучного інтелекту для моніторингу мережі» Тези доповіді на II Всеукраїнській науково-технічній конференції «Технологічні горизонти: дослідження та застосування інформаційних технологій для технологічного прогресу України і світу». – Київ, 18 листопада 2024 року.
2. Даниленко А. Є. «Тенденції та технології сучасної розробки програмного забезпечення» Тези доповіді на III Всеукраїнській науково-технічній конференції «Технологічні горизонти: дослідження та застосування інформаційних технологій для технологічного прогресу України і світу». – Київ, 18 листопада 2025 року.

12

ДЯКУЮ ЗА УВАГУ!