Міністерство освіти і науки України
Національний технічний
університет
«Дніпровська політехніка»

Інститут електроенергетики
(інститут)
факультет інформаційних технологій
(факультет)
Кафедра інформаційних технологій та комп’ютерної інженерії
(повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА
кваліфікаційної роботи ступеня магістр

Здобувача вищої освіти Кваші Олега Олександровича
(ПІБ)
академічної групи 126м-23-1
(шифр)

спеціальності 126 «Інформаційні системи та технології»
(код і назва спеціальності)
спеціалізації за освітньо-професійною (освітньо-науковою) програмою
126 «Інформаційні системи та технології»
(офіційна назва)

на тему «Дослідження та впровадження патернів Cache-Aside і Claim-Check у веб-застосунку "Телеграм-бот ІТКІ"»
(назва за наказом ректора)

Керівники	Прізвище, ініціали	Оцінка за шкалою		Підпис
		рейтинговою	інституційною	
кваліфікаційної роботи	доц. Каштан В.Ю.			
розділів:				

Рецензент				

Нормоконтролер	Коротенко Г.М.			

Дніпро
2024

ЗАТВЕРДЖЕНО:
завідувач кафедри
інформаційних технологій
та комп'ютерної інженерії
(повна назва)

___________ Гнатушенко В.В.
(підпис) (прізвище та ініціали)

"_" __________ 2024 року

ЗАВДАННЯ
на кваліфікаційну роботу
ступеня магістр

здобувача вищої освіти ___Кваші О.О.____ **академічної групи** ___126м-23-1
(прізвище та ініціали) (шифр)

спеціальності 126 «Інформаційні системи та технології»__________________

спеціалізації за освітньою-професійною програмою______________
__126 «Інформаційні системи та технології»_______________________________

на тему__«Дослідження та впровадження патернів Cache-Aside і Claim-Check у веб-застосунку "Телеграм-бот ІТКІ"»_________________

затверджену наказом ректора НТУ «Дніпровська політехніка» від 17.10.2024 № 1388-с

Розділ	**Зміст**	**Термін виконання**
Розділ 1	Аналіз теми та постановка задачі	02.10.2024 – 23.10.2024
Розділ 2	Побудова архітектури інформаційної системи. Впровадження патернів проектування.	24.10.2024 – 01.11.2024
Розділ 3	Експерементальна частина. Підготовка матеріалів для захисту роботи	02.11.2024 – 30.11.2024

Завдання видано __________________ доц. Каштан В.Ю.
(підпис керівника) (прізвище, ініціали)

Дата видачі ________________

Дата подання до екзаменаційної комісії ________________

Прийнято до виконання _______________ Кваша О.О.
(підпис студента) (прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка: 142 с., 16 рис., 1 дод., 19 джерел.

Об'єктом дослідження є веб-застосунок "Телеграм-бот ІТКІ".

Предметом дослідження є патерни проектування Cache-Aside і Claim-Check, їх впровадження з використанням технологій Redis та Minio, а також їх вплив на підвищення ефективності та безпеки веб-застосунку.

Метою дослідження є розробка алгоритмів реалізацій патернів Cache-Aside і Claim-Check у веб-застосунку "Телеграм-бот ІТКІ" для оптимізації його роботи. Це передбачає інтеграцію системи кешування на основі Redis для зменшення навантаження на базу даних та прискорення аутентифікації через JWT-токени, а також використання Minio для ефективної обробки файлів.

Для досягнення мети дослідження визначено наступні задачі:

– проаналізувати поточну архітектуру веб-застосунку "Телеграм-бот ІТКІ" та виявити основні проблеми, пов'язані з продуктивністю та безпекою;

– дослідити патерни Cache-Aside і Claim-Check, визначити їх переваги та доцільність застосування у контексті даного веб-застосунку;

– розробити та впровадити алгоритм кешування даних за допомогою Redis відповідно до патерну Cache-Aside для зниження навантаження на базу даних;

– інтегрувати Minio як сховище файлів, розробити алгоритм-релізаці патерну Claim-Check для ефективної передачі та зберігання даних між адміністративною платформою та клієнтами;

– розробити методи захисту аутентифікаційних токенів, щоб запобігти їх несанкціонованому використанню після виходу користувача із системи;

– провести порівняльний аналіз продуктивності та безпеки системи до і після впровадження запропонованих патернів для оцінки їх ефективності.

Ключові слова: Телеграм-бот ІТКІ, Cache-Aside, Claim-Check, Redis, Minio, кешування даних, зберігання файлів, JWT-токени.

ABSTRACT

Explanatory note: 142 p., 16 fig., 1 appendix, 19 sources.

The object of the study is the web application "ITKI Telegram-bot".

The subject of the study is the Cache-Aside and Claim-Check design patterns, their implementation using Redis and Minio technologies, as well as their impact on increasing the efficiency and security of the web application.

The purpose of the study is to develop algorithms for implementing the Cache-Aside and Claim-Check patterns in the web application "ITKI Telegram-bot" to optimize its operation. This involves integrating a Redis-based caching system to reduce the load on the database and accelerate authentication via JWT tokens, as well as using Minio for efficient file processing.

To achieve the goal of the study, the following tasks have been defined:

– analyze the current architecture of the web application "ITKI Telegram-bot" and identify the main problems related to performance and security;

– investigate the Cache-Aside and Claim-Check patterns, determine their advantages and feasibility of application in the context of this web application;

– develop and implement a data caching algorithm using Redis according to the Cache-Aside pattern to reduce the load on the database;

– integrate Minio as a file storage, develop an algorithm-release of the Claim-Check pattern for efficient data transfer and storage between the administrative platform and clients;

– develop methods for protecting authentication tokens to prevent their unauthorized use after the user logs out of the system;

– conduct a comparative analysis of the performance and security of the system before and after the implementation of the proposed patterns to assess their effectiveness.

Keywords: Telegram bot ITKI, Cache-Aside, Claim-Check, Redis, Minio, data caching, file storage, JWT tokens.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ І ТЕРМІНІВ **7**
ВСТУП **8**
1 АНАЛІЗ ПРОБЛЕМАТИКИ ТА СТАНУ ОБЛАСТІ РІШЕННЯ ЗАДАЧІ **10**
1.1 Огляд веб-застосунку "Телеграм-бот ІТКІ" 10
1.2 Аналіз проблематики архітектури веб-застосунку 14
1.3 Аналіз проблематики безпеки веб-застосунку 16
1.4 Постановка задачі 19
1.5 Висновки по розділу 1 20
2 МОДЕЛІ ТА МЕТОДИ РОЗВ’ЯЗАННЯ ЗАДАЧІ **21**
2.1 Проектування архітектури системи 21
2.2 Огляд патернів проектування в веб-застосунках 22
2.2.1 Патерн Cache-Aside 22
2.2.2 Патерн Claim-Check 26
2.3 Аналіз альтернативних патернів для оптимізації 30
2.3.1 Альтернативні патерни Cache-Aside 30
2.3.2 Альтернативні патерни Claim-Check 39
2.4 Огляд технологій Redis та Minio 48
2.4.1 Особливості Redis для кешування 48
2.4.2 Особливості Minio для зберігання файлів 63
2.5 Висновки по розділу 2 65
3 АНАЛІЗ ПРОГРАМНОЇ РЕАЛІЗАЦІЇ **67**
3.1 Опис програмної реалізації 67
3.2 Аналіз механізму кешування даних 68
3.2.1 Інтеграція Redis у застосунок 68
3.2.2 Аналіз алгоритму роботи з кешем 69
3.3 Аналіз механізму зберігання та обробки файлів 71
3.3.1 Інтеграція Minio у застосунок 71
3.3.2 Аналіз обробки файлів за патерном Claim-Check 74
3.4 Аналіз додаткового захисту аутентифікаційних токенів 76
3.5 Перевірка результатів оптимізації додатку 78

3.5.1 Перевірка результатів отримання даних користувача з кешу 78
3.5.2 Перевірка результатів відповідей /login ендпоінту з кешем 79
3.5.3 Перевірка результатів асинхронного запиту масової розсилки 81
3.6 Висновки по розділу 3 82
ВИСНОВКИ **84**
ДОДАТОК А. ОНОВЛЕННІ ФАЙЛИ ЗАСТОСУНКУ **88**

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ І ТЕРМІНІВ

API – програмний інтерфейс додатку (англ. Application programing interface)

IT – інформаційні технології

REST – передача стану представлення (англ. Representational State Transfer)

HTTP – протокол передачі гіпертексту (англ. Hypertext Transfer Protocol), протокол, що використовується для передачі даних у веб-переглядачах та веб-серверах

SQS – керована служба черг повідомлень (англ. Simple Queue Service)

S3 – масштабована та надійна хмарна служба зберігання даних (англ. Simple Storage Service)

ВСТУП

У сучасних умовах стрімкого розвитку інформаційних технологій веб-застосунки стають невід'ємною частиною повсякденного життя. Користувачі очікують від них високої швидкодії, надійності та безпеки. Зі зростанням обсягу даних і кількості користувачів виникають нові виклики, пов'язані з оптимізацією продуктивності та масштабованості систем.

"Телеграм-бот ІТКІ" — це комплексний веб-застосунок, який забезпечує зручний доступ до різноманітних сервісів через платформу Telegram для клієнтів та зручні інструменти адміністрування телеграм-боту через власні графічні інтерфейси (Node-RED flow, адміністративна панель).

Під час аналізу його поточної архітектури було виявлено низку недоліків, які обмежують можливості застосунку та можуть впливати на якість користувацького досвіду. Серед них: неефективне управління даними, недосконалість механізмів аутентифікації та недостатньо оптимізовані процеси зберігання та обробки файлів.

Ця кваліфікаційна робота магістра спрямована на дослідження та усунення виявлених недоліків шляхом впровадження сучасних архітектурних рішень. Зокрема, розглядається застосування патернів проектування Cache-Aside та Claim-Check для оптимізації роботи з даними та файлами. Використання Redis для кешування даних за патерном Cache-Aside дозволить знизити кількість звернень до основної бази даних та прискорити процеси аутентифікації. Інтеграція Minio як сховища файлів відповідно до патерну Claim-Check забезпечить ефективне зберігання та передачу файлів між компонентами системи.

У ході дослідження детально аналізуються виявлені проблеми поточної архітектури та обґрунтовується вибір відповідних патернів та технологій для їх вирішення. Особлива увага приділяється питанням безпеки, зокрема захисту аутентифікаційних токенів та запобіганню їх несанкціонованому використанню після виходу користувача із системи.

Практична реалізація запропонованих рішень спрямована на підвищення продуктивності веб-застосунку, оптимізацію використання ресурсів та покращення безпеки даних. Очікується, що впровадження цих змін дозволить поліпшити загальну ефективність системи та забезпечити більш комфортний та надійний досвід для користувачів.

Результати роботи демонструють можливість значного вдосконалення веб-застосунку шляхом усунення виявлених недоліків та впровадження сучасних архітектурних підходів. Це відкриває перспективи для подальшого розвитку системи та підвищення її конкурентоспроможності в умовах сучасних вимог до якості програмного забезпечення.

1 АНАЛІЗ ПРОБЛЕМАТИКИ ТА СТАНУ ОБЛАСТІ РІШЕННЯ ЗАДАЧІ

1.1 Огляд веб-застосунку "Телеграм-бот ІТКІ"

"Телеграм-бот ІТКІ" – це комплексний веб-застосунок, що складається з 4 серверів та взаємодіє з Telegram bot API для надання послуг та виконання масових розсилок повідомлень.

Перший компонент цієї інформаційної системи – це PostgreSQL-сервер, що забезпечує зберігання основних даних про користувачів бота, дані аккаунтів адміністраторів, права доступу до платформи та API (ролі), академічні групи, кураторів академічних груп, заготовлені питання та відповіді до них. Всі таблиці для представлення цих можна побачити на ER-діаграмі додатку (Рис. 1.1).

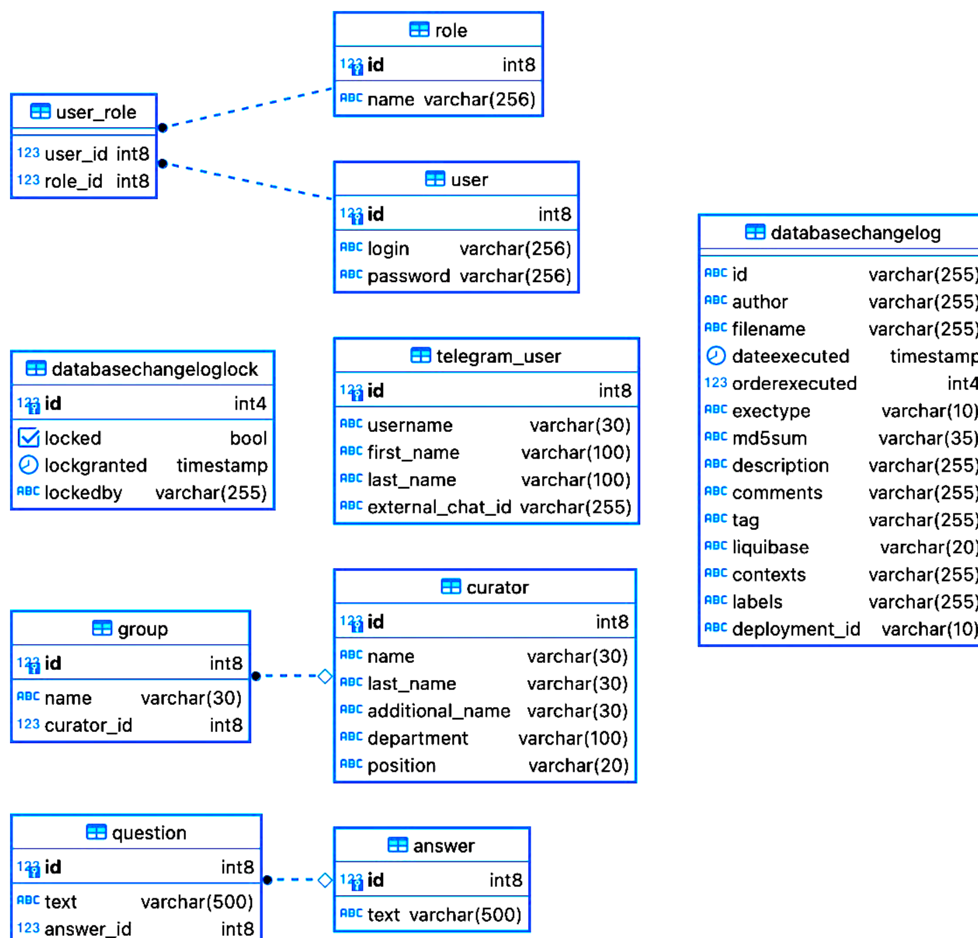

Рисунок 1.1 – ER-діаграма схеми серверу PostgreSQL

Другий компонент – це REST API яке взаємодіє напряму з PostgreSQL-сервером. Цей компонент виконує кілька значущих функцій у додатку.

Перше – REST API виконує функції CRUD інтерфейсу для таблиць в PostgreSQL.

Друге – виконує функції фіксації змін до PostgreSQL за рахунок інтеграції з кросплатформовою системою міграції баз даних Liquibase.

Третє – виконує роль серверу аутентифікації. Видає права на взаємодію з endpoints та JWT-токени для авторизації в системі і оновлення сесії.

Четверте – надає інтерфейс для відправки масових розсилок повідомлень до телеграму. Самі endpoints для відправки розсилок відправляють повідомлення в синхронному режимі. При відправці форми (Рис. 1.2), спершу завантажуються прикріплені файли, потім API виконує запит до БД, щоб отримати всіх користувачів, що користуються ботом додатка. Після цього API формує запити на відправку повідомлення по кожному користувачу і надсилає їх до Telegram Bot API.

Рисунок 1.2 – Форма надсилання фото з приписом

Саме REST API побудовано на основі фреймворку Spring Boot.

Третій компонент системи – це Node-RED. Він виконує запити на отримання інформації з сайту університету, створює меню команд в боті (Рис. 1.3) та оброблює запити по командам в боті.

Рисунок 1.3 – Меню бота створене за допомогою Node-RED

Четвертий компонент системи – це Angular frontend який забезпечує основний UI-інтерфейс для взаємодії з основними інструментами адміністрування веб-застосунку.

Після успішної аутентифікації користувач потрапляє на головну сторінку адміністративної панелі, де представлені чотири основні групи опцій:

– конфігурація бота. Цей розділ дозволяє адміністратору керувати питаннями, на які бот може відповідати. Інтерфейс відображає список питань з можливістю додавання нових через модальне вікно та видалення існуючих.

Додані питання автоматично відображаються у клавіатурі бота для користувачів;

– адміністрування. Тут можна додавати та редагувати кураторів і студентські групи. Додавання кураторів реалізовано через модальні вікна з валідацією введених даних. Для кожного куратора можна додати відповідні групи студентів. Якщо куратор вже має призначені групи, його видалення стає неможливим, що запобігає втраті важливої інформації;

– статистика. У цьому розділі відображається список всіх користувачів, які розпочали взаємодію з ботом у Telegram. Доступна інформація може включати ідентифікатор користувача, ім'я та інші дані, надані Telegram API. Ця інформація допомагає відстежувати активність та залученість користувачів;

– обліковий запис. Розділ для керування особистими налаштуваннями адміністратора та виходу з системи.

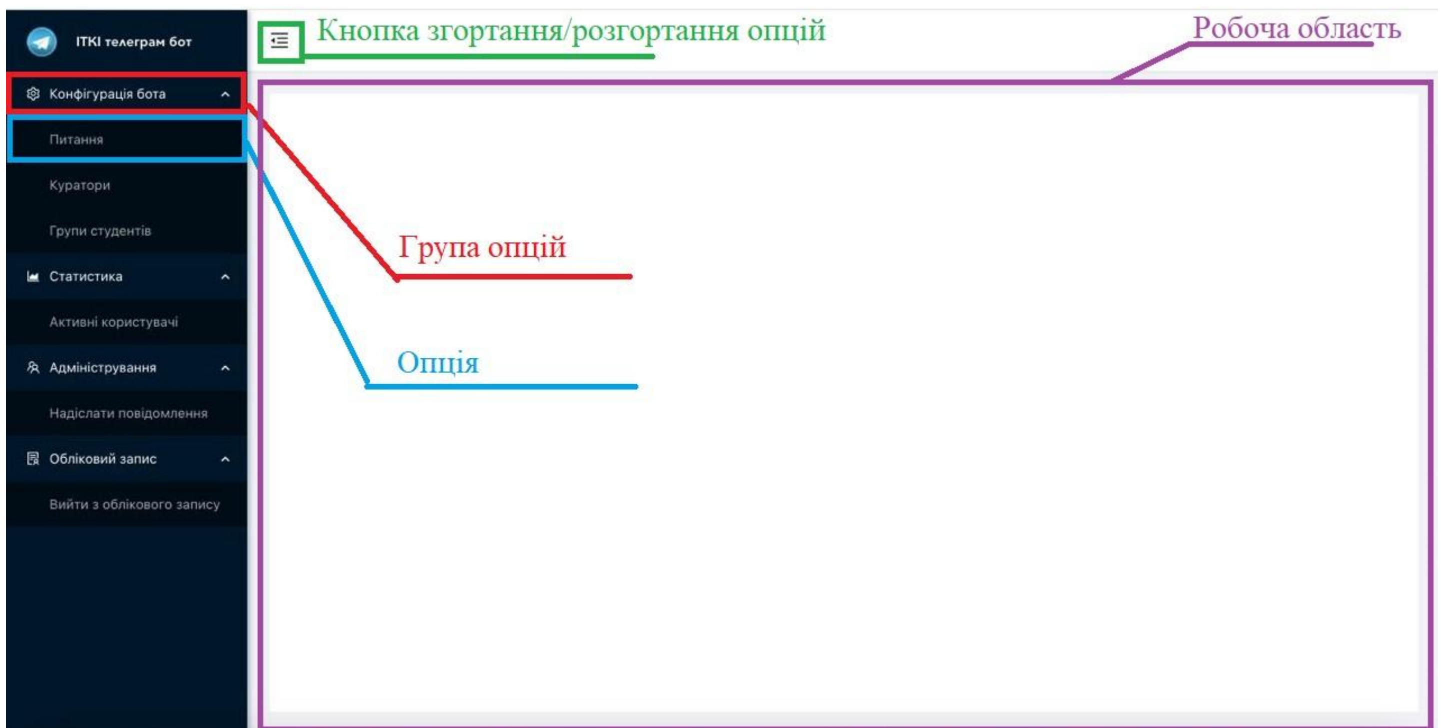

Рисунок 1.4 – Схема структури Angular UI

1.2 Аналіз проблематики архітектури веб-застосунку

Архітектура веб-застосунку "Телеграм-бот ІТКІ" побудована переважно на монолітному підході і використовує синхронну обробкою запитів для масових розсилок (Рис. 1.5). Така архітектура має свої особливості, які впливають на продуктивність, масштабованість та гнучкість системи, а також породжує низку недоліків.

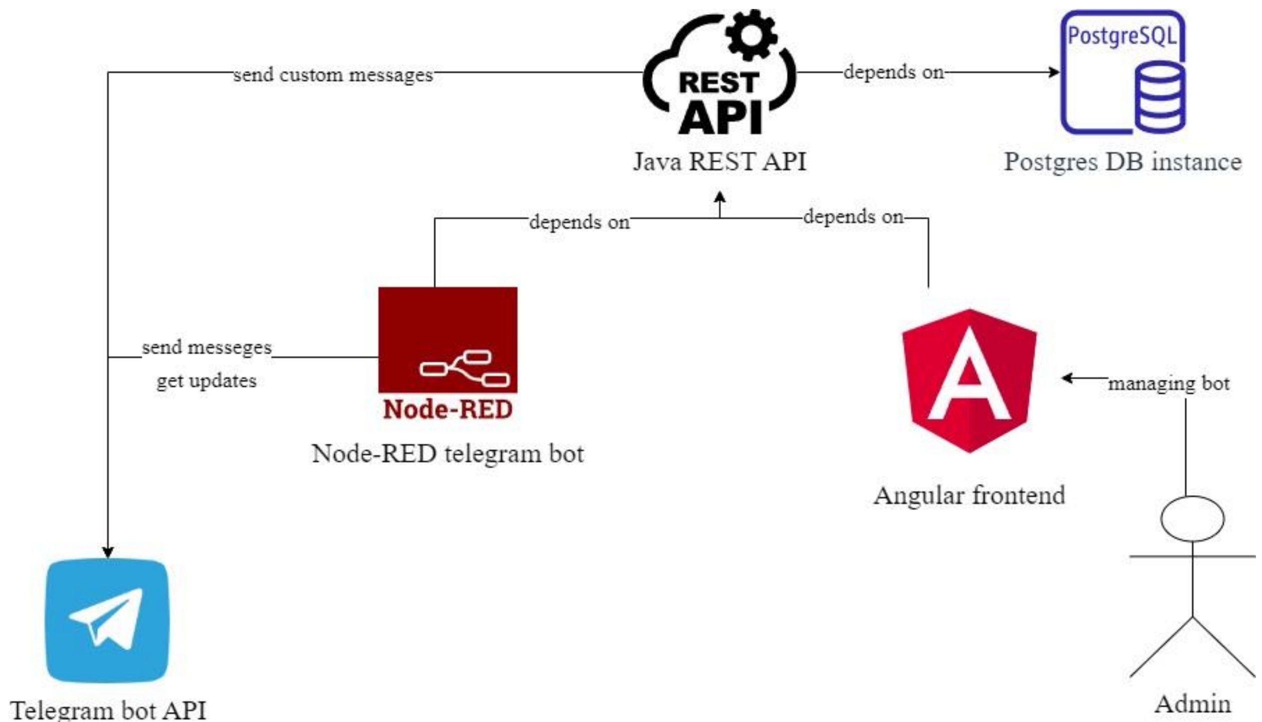

Рисунок 1.5 – Діаграма архітектури "Телеграм-бот ІТКІ"

Монолітна архітектура передбачає, що всі функціональні компоненти застосунку – від інтерфейсу користувача до управління базою даних — інтегровані в єдину кодову базу. Цей підхід спрощує початкову розробку та розгортання, оскільки не потребує складної координації між окремими сервісами. Однак з ростом обсягу функціональності та кількості користувачів монолітний застосунок стає важко підтримувати та масштабувати.

Одним з головних недоліків монолітної архітектури є складність масштабування. При збільшенні навантаження немає можливості масштабувати окремі компоненти системи незалежно; необхідно розгортати додаткові копії всього застосунку. Це неефективно з точки зору ресурсів і

може призвести до зайвих витрат. Крім того, будь-які зміни або оновлення в одній частині застосунку вимагають повторного розгортання всього сервісу, що підвищує ризик виникнення помилок і знижує стабільність системи.

Синхронна обробка запитів на масові розсилки також створює суттєві проблеми. У такому режимі всі запити обробляються послідовно, і кожен наступний запит чекає завершення попереднього. Це може призвести до значних затримок, особливо при великій кількості одержувачів. Користувачі можуть відчувати повільну реакцію системи або навіть збоїв в обслуговуванні. Також синхронна обробка обмежує здатність системи ефективно використовувати ресурси, оскільки блокуються потоки, які могли б виконувати інші завдання.

Іншим недоліком є складність впровадження нових технологій та патернів проектування в монолітній архітектурі. Оскільки всі компоненти тісно пов'язані між собою, оновлення однієї частини може вимагати змін в інших, що ускладнює процес розробки та тестування. Це обмежує гнучкість системи і може стримувати впровадження інновацій.

Неефективне управління даними в монолітному застосунку може призвести до перевантаження бази даних. Відсутність механізмів кешування змушує систему щоразу звертатися до бази даних навіть для часто використовуваних або незмінних даних. Це збільшує час відповіді та знижує загальну продуктивність. Недосконалі механізми аутентифікації можуть стати причиною вразливостей в безпеці, що особливо критично для застосунків, які працюють з особистими даними користувачів.

Процеси зберігання та обробки файлів також можуть бути неоптимальними. Зберігання файлів безпосередньо в базі даних або файловій системі сервера може призвести до проблем з масштабованістю та резервуванням даних. Відсутність спеціалізованих сервісів для зберігання файлів ускладнює управління ними та може впливати на швидкодію інших компонентів системи.

Таким чином, використання монолітної архітектури з синхронною

обробкою запитів у веб-застосунку "Телеграм-бот ІТКІ" призводить до таких проблем як:

– обмежена масштабованість, неможливість незалежного масштабування окремих компонентів веде до неефективного використання ресурсів;

– складність підтримки та розвитку, тісна зв'язаність компонентів ускладнює внесення змін та впровадження нових функцій;

– низька продуктивність, синхронна обробка запитів та відсутність кешування підвищують час відповіді та знижують швидкодію системи;

– проблеми з безпекою, недосконалі механізми аутентифікації можуть призвести до несанкціонованого доступу та компрометації даних;

– неоптимальне управління файлами, зберігання файлів без спеціалізованих рішень ускладнює їх обробку та доступ.

1.3 Аналіз проблематики безпеки веб-застосунку

Безпека веб-застосунку "Телеграм-бот ІТКІ" є критичним аспектом, який потребує особливої уваги, зокрема в частині аутентифікації користувачів та управління їхніми сесіями. Поточна реалізація механізму аутентифікації базується на використанні JWT-токенів (JSON Web Tokens), що забезпечує швидку та ефективну ідентифікацію користувачів без значного навантаження на сервер. Однак цей підхід має і суттєві вразливості, які можуть призвести до серйозних наслідків у разі експлуатації зловмисниками.

Алгоритм аутентифікації у застосунку працює наступним чином: користувач надсилає POST-запит на ендпоінт /login, передаючи свій логін та пароль. У відповідь REST API повертає два підписані JWT-токени — authToken та refreshToken, які містять службову інформацію та мають обмежений строк дії. Для доступу до захищених ресурсів користувач додає authToken до заголовка аутентифікації кожного запиту. Коли строк дії authToken закінчується, клієнт надсилає refreshToken на ендпоінт /refresh і

отримує нові токени, що дозволяє безперервно продовжувати сесію. У фронтенді, реалізованому на Angular, ці токени зберігаються в cookies і видаляються при виході користувача з системи.

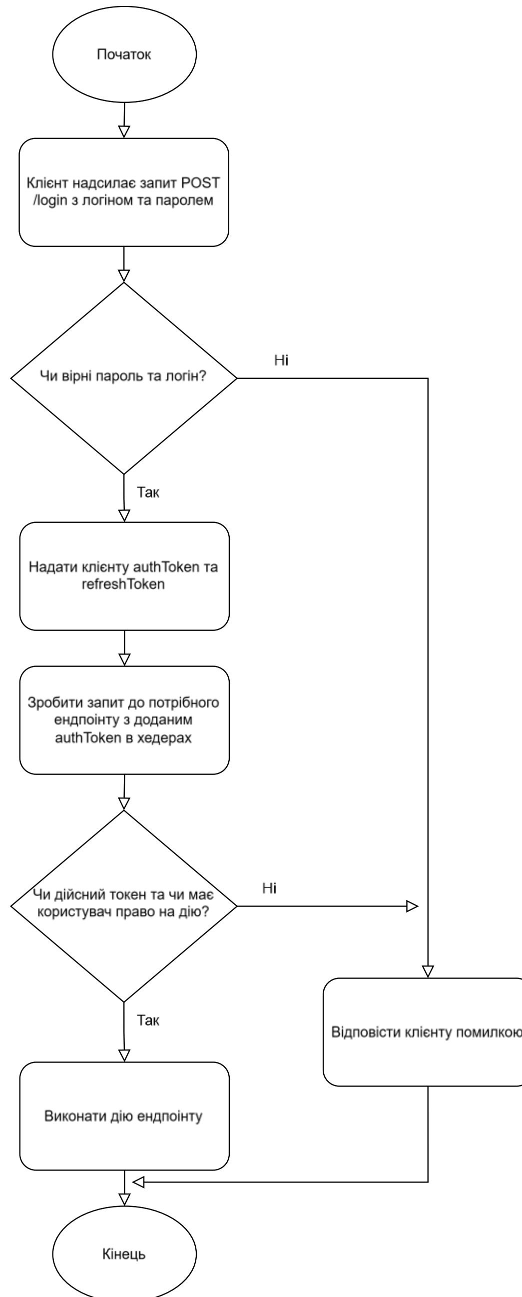

Рисунок 1.6 – Алгоритм аутентифікації на API

Основна проблема полягає в тому, що після виходу користувача з системи його JWT-токени залишаються дійсними до моменту закінчення їхнього строку дії. Це створює серйозний ризик: якщо зловмисник зуміє отримати ці токени до виходу користувача, він зможе продовжувати

користуватися його сесією навіть після того, як користувач вийшов із системи. Така ситуація може виникнути через різні вразливості, наприклад, через XSS-атаки, компрометацію пристрою користувача або ненадійне зберігання токенів на клієнтській стороні.

Причина цієї вразливості криється в природі JWT-токенів. Вони є самодостатніми і не потребують збереження стану сесії на сервері. Сервер не перевіряє кожного разу, чи дійсна сесія користувача, а довіряє інформації, закодованій у токені, до закінчення його строку дії. Видалення токенів з cookies на клієнтській стороні не вирішує проблему, оскільки скомпрометовані токени можуть бути використані зловмисником незалежно від дій користувача.

Наслідки такої вразливості можуть бути значними. Зловмисник отримує можливість несанкціонованого доступу до системи, може виконувати дії від імені користувача, отримувати та змінювати конфіденційну інформацію. Це не лише порушує конфіденційність та безпеку даних, але й може призвести до юридичних наслідків та серйозно підірвати довіру користувачів до застосунку. Репутація організації також може постраждати внаслідок безпекових інцидентів.

Для вирішення цієї проблеми доцільно впровадити механізм відкликання токенів шляхом їх додавання до чорного списку після виходу користувача з системи. Це означає, що після завершення сесії користувача його authToken та refreshToken зберігатимуться на сервері як недійсні до закінчення їхнього строку дії. При кожному запиті сервер перевірятиме, чи не знаходиться наданий токен у цьому чорному списку. Якщо токен виявиться недійсним, запит буде відхилено. Таким чином, навіть якщо зловмисник отримає токени, він не зможе ними скористатися після виходу користувача з системи.

Впровадження цього механізму підвищить рівень безпеки застосунку, оскільки дозволить контролювати дійсність токенів в режимі реального часу. Зберігаючи переваги використання JWT-токенів, такий підхід додає

додатковий рівень захисту.

1.4 Постановка задачі

Метою даної роботи є вдосконалення веб-застосунку "Телеграм-бот ІТКІ" шляхом розробки та впровадження сучасних архітектурних рішень, які підвищать продуктивність, масштабованість та безпеку системи. Для досягнення цієї мети пропонується інтеграція систем кешування на основі Redis за патерном Cache-Aside, що знизить навантаження на базу даних і прискорить процеси аутентифікації. Також планується використання Minio для зберігання та обробки файлів за патерном Claim-Check, що забезпечить ефективну передачу даних між адміністративною платформою та клієнтами. Для підвищення безпеки буде розроблено механізм захисту аутентифікаційних токенів, який запобігатиме їх використанню після виходу користувача з системи.

Для реалізації поставленої мети необхідно вирішити такі завдання:

– провести детальний аналіз існуючої архітектури веб-застосунку "Телеграм-бот ІТКІ" та ідентифікувати основні проблеми, що впливають на його продуктивність та безпеку;

– розробити та впровадити механізм кешування даних за допомогою Redis, застосовуючи патерн Cache-Aside, що дозволить знизити кількість звернень до бази даних та прискорить аутентифікацію через JWT-токени;

– інтегрувати Minio як систему зберігання файлів, реалізуючи патерн Claim-Check для оптимізації процесів зберігання та передачі файлів між компонентами системи;

– розробити механізм захисту аутентифікаційних токенів, який передбачає зберігання відкликаних токенів у чорному списку, що унеможливить їх використання після виходу користувача з системи;

– провести тестування та оцінку продуктивності та безпеки системи до і після впровадження запропонованих рішень, щоб визначити їх ефективність та вплив на користувацький досвід.

1.5 Висновки по розділу 1

У розділі 1 було проведено детальний аналіз архітектури веб-застосунку "Телеграм-бот ІТКІ" з метою виявлення проблем, що впливають на його продуктивність, масштабованість та безпеку. Аналіз показав, що використання монолітної архітектури з синхронною обробкою запитів та неефективними механізмами управління даними призводить до затримок у роботі системи, підвищеного навантаження на базу даних та потенційних вразливостей в аутентифікації користувачів.

З метою усунення виявлених недоліків було запропоновано впровадження сучасних архітектурних патернів Cache-Aside та Claim-Check. Реалізація патерну Cache-Aside з використанням Redis дозволила знизити кількість звернень до бази даних та прискорити процеси аутентифікації за допомогою JWT-токенів. Інтеграція Minio за патерном Claim-Check забезпечила ефективне зберігання та обробку файлів, що покращило передачу даних між адміністративною платформою та клієнтами.

Особлива увага була приділена підвищенню безпеки веб-застосунку. Розроблено механізм захисту аутентифікаційних токенів, який передбачає зберігання відкликаних токенів у чорному списку після виходу користувача із системи. Це унеможливлює використання скомпрометованих токенів зловмисниками та запобігає несанкціонованому доступу до системи.

У результаті проведеної роботи було досягнуто підвищення продуктивності та надійності веб-застосунку "Телеграм-бот ІТКІ", а також забезпечено високий рівень безпеки даних користувачів. Впроваджені рішення сприяли покращенню користувацького досвіду та відкрили можливості для подальшого розвитку системи.

2 МОДЕЛІ ТА МЕТОДИ РОЗВ'ЯЗАННЯ ЗАДАЧІ

2.1 Проектування архітектури системи

Зважаючи на архітектурні проблеми, розглянуті у розділі 1, було прийнято рішення розширити поточну архітектуру веб-застосунку "Телеграм-бот ІТКІ" (Рис. 1.5) шляхом впровадження двох нових сервісів, що реалізують допоможуть реалізувати патерни Cache-Aside та Claim-Check. Метою цього розширення є підвищення ефективності системи, зменшення навантаження на базу даних та забезпечення надійного зберігання великих файлів-вкладень.

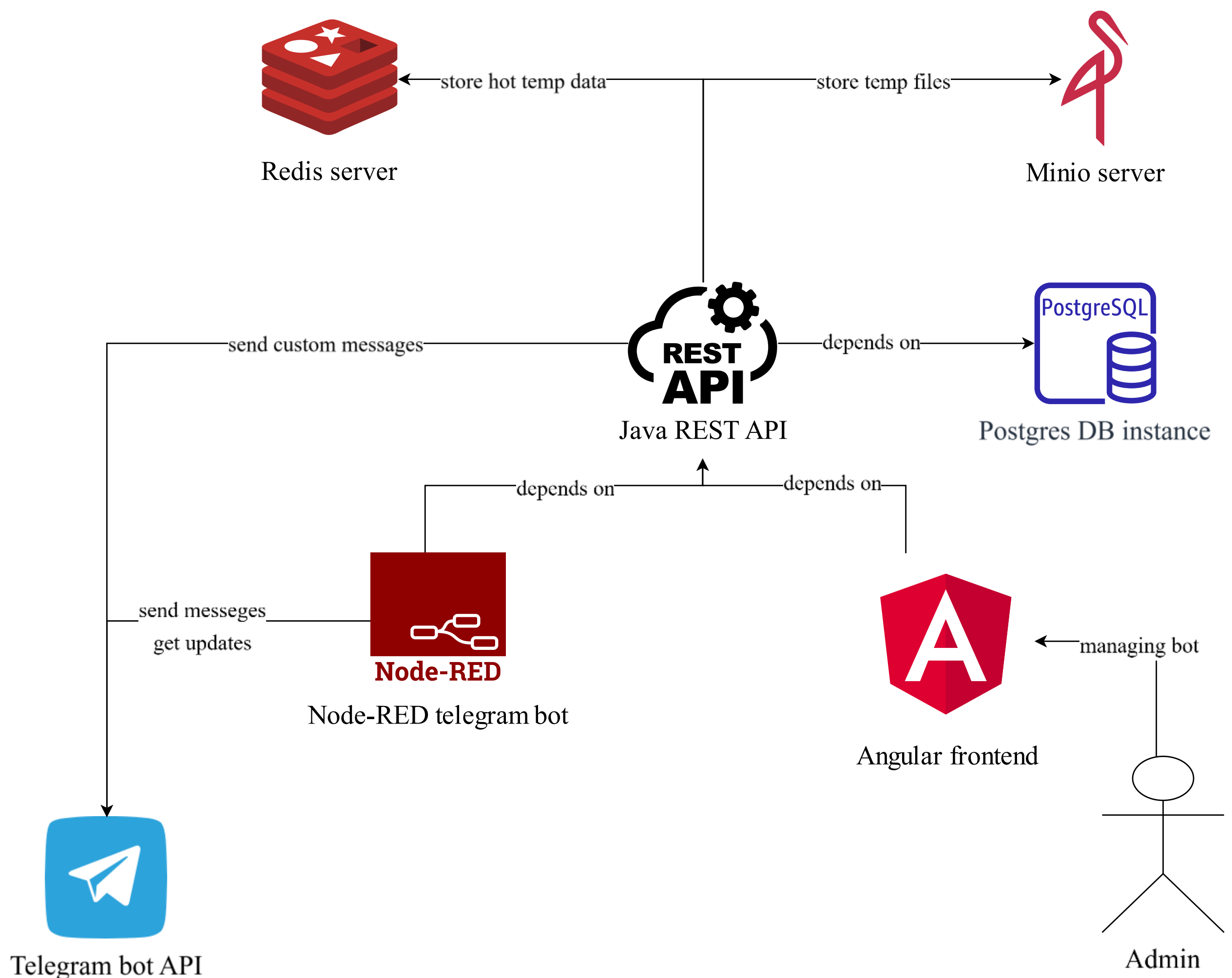

Рисунок 2.1 – Оновлена архітектура веб-застосунку

Як можна побачити на Рис. 2.1 до архітектури додатку було додано два окремих сервіси з якими взаємодіє REST API.

Першим із впроваджених сервісів є Redis server. Він використовується

як швидкодіюче сховище даних, що часто запитуються через REST API. Це дозволяє зменшити навантаження на основну базу даних, зберігаючи в кеші результати частих запитів. Крім того, Redis служить для зберігання тимчасових даних, які необхідно автоматично знищувати після заданого проміжку часу. Реалізація патерну Cache-Aside за допомогою Redis забезпечить ефективне управління кешем, де додаток самостійно вирішує, коли звертатися до кешу, а коли — безпосередньо до бази даних.

Другим сервісом є Minio server, який використовується для зберігання файлів, прикріплених до повідомлень масових розсилок від адміністраторів. Використання Minio як об'єктного сховища дозволяє надійно зберігати великі файли та легко масштабувати сховище при зростанні обсягу даних. Інтеграція Minio реалізує патерн Claim-Check, який полягає в тому, що замість передачі великих файлів через основний сервіс, передаються лише посилання або ідентифікатори цих файлів. Це зменшує розмір переданих даних і оптимізує використання мережевих ресурсів.

Впровадження цих сервісів сприяє підвищенню масштабованості та продуктивності веб-застосунку. Користувачі отримують швидкий доступ до часто використовуваних даних завдяки кешуванню, а обробка великих файлів відбувається більш ефективно через використання зовнішнього сховища. Оновлена архітектура забезпечує модульність системи, спрощує її підтримку та подальший розвиток, а також дозволяє легше інтегрувати нові функціональні можливості в майбутньому.

2.2 Огляд патернів проектування в веб-застосунках

2.2.1 Патерн Cache-Aside

Патерн Cache-Aside є одним із ключових архітектурних рішень у веб-розробці, спрямованих на підвищення продуктивності системи та оптимізацію доступу до даних. У сучасних веб-застосунках, де швидкодія та масштабованість є критичними факторами, ефективне кешування стає необхідністю для забезпечення позитивного користувацького досвіду та

зниження навантаження на серверні ресурси. Суть патерну Cache-Aside полягає в тому, що кешування даних здійснюється на рівні застосунку, а не на рівні бази даних або інших нижчих шарів системи. При цьому застосунок самостійно вирішує, коли і які дані потрібно зберігати в кеш [1].

Основний механізм читання за патерном Cache-Aside можна описати наступним чином:

– застосунок спочатку перевіряє наявність необхідних даних у кеші;

– якщо дані знайдено (cache-hit), вони негайно повертаються користувачу;

– якщо даних немає в кеші (cache-miss), застосунок звертається до основної бази даних або іншого сховища;

– після отримання даних із бази вони зберігаються в кеші для майбутніх запитів.

А основний механізм запису за патерном Cache-Aside можна представити як:

– під час оновлення або видалення даних застосунок вносить зміни безпосередньо в базу даних;

– якщо при читанні з кешу даних не було (cache-miss), тоді вони записуються в кеш.

Такий підхід дозволяє контролювати процес кешування та забезпечує гнучкість у управлінні даними. Патерн Cache-Aside є особливо ефективним для систем з високою частотою читання даних і відносно низькою частотою їх оновлення.

До переваг патерну Cache-Aside можна віднести:

– підвищення швидкодії – завдяки зберіганню часто запитуваних даних у кеші, зменшується час відгуку системи, оскільки доступ до кешу зазвичай швидший, ніж до бази даних;

– зниження навантаження на базу даних – менша кількість запитів до бази даних дозволяє знизити її навантаження, що може бути критичним у

високонавантажених системах;

– гнучкість – застосунок самостійно керує кешем, що дозволяє реалізовувати складні стратегії кешування відповідно до специфіки бізнес-логіки;

– простота реалізації – патерн є відносно простим у впровадженні та не вимагає суттєвих змін у існуючій архітектурі застосунку.

Однак при використанні патерну Cache-Aside існують певні недоліки та виклики:

– консистентність даних – можливі ситуації, коли дані в кеші не відповідають актуальному стану бази даних, особливо при високій частоті оновлень;

– складність управління кешем – необхідно розробляти механізми інвалідизації та оновлення кешу, що може ускладнити код застосунку;

– використання пам'яті – кешування потребує додаткової пам'яті для зберігання даних, що може бути обмеженням у деяких середовищах.

Для успішного впровадження патерну Cache-Aside слід дотримуватися кращих практик:

– визначення часу життя даних (TTL) – встановлення відповідних значень TTL для кешованих даних допомагає підтримувати їх актуальність та запобігає надмірному споживанню пам'яті;

– інвалідизація кешу – реалізація механізмів, які забезпечують оновлення або видалення даних із кешу після зміни в базі даних;

– моніторинг та аналітика – відстеження ефективності кешування та налаштування параметрів на основі реальних даних про роботу системи;

– розподіл кешу – у розподілених системах слід враховувати необхідність синхронізації кешу між різними вузлами або використання централізованого кешу.

У веб-застосунках патерн Cache-Aside часто використовується для кешування результатів складних або ресурсомістких запитів до бази даних, які

не змінюються часто. Наприклад, це можуть бути дані про продукти в інтернет-магазині, конфігураційні дані або статистичні показники. У контексті веб-застосунку "Телеграм-бот ІТКІ" впровадження патерну Cache-Aside може мати суттєвий вплив на продуктивність та масштабованість системи. Зокрема, кешування сесій користувачів дозволить швидко перевіряти аутентифікацію та авторизацію без постійних звернень до бази даних.

Патерн Cache-Aside може ефективно поєднуватися з іншими архітектурними рішеннями для досягнення максимального ефекту. Наприклад, у випадку недоступності бази даних кеш може слугувати тимчасовим джерелом даних, дозволяючи системі продовжувати роботу. Також він добре поєднується з патерном Lazy Loading, коли дані завантажуються в кеш лише тоді, коли вони вперше запитуються, що зменшує початкове навантаження при старті застосунку.

Патерн Lazy Loading (ліниве завантаження) – це підхід у розробці програмного забезпечення, при якому ініціалізація об'єктів або завантаження ресурсів відбувається лише тоді, коли вони стають необхідними. Замість того, щоб завантажувати всі можливі дані при старті застосунку, система відкладає цю операцію до моменту першого запиту конкретного ресурсу. Такий підхід дозволяє зменшити початкове навантаження на систему, скоротити час запуску та оптимізувати використання ресурсів.

У контексті патерну Cache-Aside, Lazy Loading означає, що дані додаються в кеш лише після першого звернення до них. Якщо певні дані ніколи не запитуються, вони і не завантажуються, що допомагає економити пам'ять та зменшити непотрібний трафік між застосунком і базою даних. Поєднання цих двох патернів сприяє підвищенню ефективності системи, оскільки забезпечує швидкий доступ до даних при мінімальному споживанні ресурсів [2].

Lazy Loading особливо корисний у системах з великою кількістю потенційних даних, але з непередбачуваними або рідкими запитами до них. Наприклад, у веб-застосунках, де користувачі можуть ніколи не дійти до

певних розділів або функцій, немає сенсу завантажувати відповідні дані заздалегідь. Використання цього патерну дозволяє системі масштабуватися та підтримувати високу продуктивність навіть при обмежених ресурсах.

Хоча конкретні технології для реалізації кешу можуть варіюватися, важливо враховувати вибір сховища кешу, підтримку необхідних структур даних та аспекти безпеки. Захист кешованих даних від несанкціонованого доступу є критичним, особливо якщо вони містять конфіденційну інформацію. Враховуючи всі ці аспекти, патерн Cache-Aside є потужним інструментом для підвищення ефективності веб-застосунків. Його правильне впровадження в "Телеграм-боті ІТКІ" може знизити навантаження на базу даних, прискорити обробку запитів та покращити користувацький досвід, забезпечуючи необхідний рівень продуктивності системи.

2.2.2 Патерн Claim-Check

Патерн Claim-Check є архітектурним рішенням, яке широко використовується у розподілених системах та веб-застосунках для ефективного управління великими обсягами даних, зокрема файлами або великими повідомленнями. Основна мета цього патерну — оптимізувати передачу важких даних між компонентами системи, зменшуючи навантаження на мережу та ресурси системи. Суть патерну полягає в тому, що замість передачі великих даних безпосередньо між компонентами, передаються лише посилання або маркери (так звані "claim-checks"), які вказують, де ці дані можна отримати [3].

Механізм роботи патерну Claim-Check можна розділити на наступні основні складові:

– зберігання даних;
– передача маркера;
– отримання даних.

Зберігання даних відбувається наступним чином:

– відправник даних зберігає великий об'єкт (наприклад, файл або велике повідомлення) у спільному сховищі, доступному для всіх компонентів системи;

– після успішного зберігання відправник отримує унікальний ідентифікатор або посилання на збережений об'єкт.

Передачу маркеру відбувається можна описати як:

– замість передачі самого об'єкта, відправник передає одержувачу лише маркер або посилання на об'єкт у сховищі;

– маркер містить всю необхідну інформацію для отримання об'єкта зі сховища.

А отримання даних можна описати наступними кроками:

– одержувач, отримавши маркер, використовує його для доступу до спільного сховища;

– за допомогою маркера одержувач завантажує необхідний об'єкт для подальшої обробки.

Такий підхід дозволяє значно зменшити обсяг даних, що передаються між компонентами системи, що особливо важливо у випадках, коли мережеві ресурси обмежені або необхідно забезпечити високу пропускну здатність системи.

До переваг патерну Claim-Check можна віднести:

– ефективність передачі даних – зменшення обсягу переданих даних між компонентами системи підвищує швидкодію та знижує затримки;

– зниження навантаження на мережу – передача лише маркерів замість великих об'єктів дозволяє оптимізувати використання мережевих ресурсів;

– масштабованість – патерн сприяє кращому розподілу навантаження між компонентами системи та спрощує масштабування;

– безпека – зберігання великих об'єктів у централізованому сховищі дозволяє більш ефективно контролювати доступ та забезпечувати захист даних.

Однак при використанні патерну Claim-Check існують певні виклики та недоліки:

– складність реалізації – необхідність налаштування додаткового сховища та забезпечення доступу до нього може ускладнити архітектуру системи;

– управління доступом – потрібно забезпечити належні механізми аутентифікації та авторизації для доступу до спільного сховища;

– відмовостійкість – спільне сховище стає критичним компонентом системи, і його недоступність може призвести до збоїв у роботі застосунку;

– консистентність даних – необхідно стежити за тим, щоб маркери та відповідні об'єкти у сховищі були синхронізовані та актуальні.

Для успішного впровадження патерну Claim-Check рекомендується дотримуватися наступних кращих практик:

– вибір надійного сховища – використання стійких та масштабованих систем зберігання даних, які забезпечують високий рівень доступності та безпеки;

– захист маркерів – забезпечення безпечної передачі маркерів, щоб запобігти несанкціонованому доступу до даних;

– моніторинг та логування – відстеження операцій із зберігання та отримання об'єктів для виявлення можливих проблем та оптимізації роботи системи;

– очистка застарілих даних – реалізація механізмів видалення або архівування невикористовуваних об'єктів у сховищі для оптимізації використання ресурсів.

У контексті веб-застосунку "Телеграм-бот ІТКІ" впровадження патерну Claim-Check може значно покращити роботу з файлами та великими даними. Наприклад, якщо адміністратор створює масову розсилку, зберігання вкладених до повідомлення файлів у спільному сховищі та передача лише посилань дозволить знизити навантаження на мережу та сервер. Це також

сприятиме швидшій обробці запитів адміністраторів на масову розсилку та покращенню загальної продуктивності системи.

Поєднання патерну Claim-Check з іншими архітектурними рішеннями може підвищити ефективність системи. Наприклад, використання його разом із патерном Message Queue дозволяє організувати асинхронну обробку даних та розподіл навантаження між компонентами. Також він добре інтегрується з сервісно-орієнтованою архітектурою (SOA) або мікросервісами, де компоненти системи можуть бути географічно розподілені.

Патерн Message Queue є архітектурним рішенням, яке використовується для організації асинхронної комунікації між компонентами системи шляхом передачі повідомлень через чергу. Він дозволяє різним частинам системи обмінюватися даними без прямого підключення, що підвищує гнучкість та масштабованість застосунку. Коли відправник хоче передати повідомлення, він поміщає його в чергу, де воно зберігається до тих пір, поки одержувач не буде готовий його обробити. Це дозволяє розвантажити систему, оскільки компоненти можуть працювати незалежно один від одного, не чекаючи на негайну відповідь. Патерн Message Queue часто використовується для обробки задач у фоновому режимі, балансування навантаження та покращення загальної продуктивності системи. Він сприяє підвищенню надійності, оскільки повідомлення зберігаються в черзі до успішної доставки, і може підтримувати різні моделі доставки, такі як точка-точка або публікація-підписка.

Важливим аспектом є вибір технології для реалізації спільного сховища. Це можуть бути об'єктні сховища, такі як Amazon S3, Minio або інші подібні рішення, які забезпечують високу доступність та масштабованість. Використання таких сховищ дозволяє легко інтегруватися з існуючою інфраструктурою та забезпечити необхідний рівень безпеки та контролю доступу.

Підсумовуючи, патерн Claim-Check є ефективним інструментом для оптимізації роботи з великими даними у веб-застосунках. Його впровадження

в "Телеграм-боті ІТКІ" може знизити навантаження на мережу та сервери, покращити швидкодію та забезпечити більш гнучку та масштабовану архітектуру системи. При цьому важливо ретельно планувати реалізацію патерну, враховуючи всі можливі виклики та дотримуючись кращих практик для забезпечення надійності та безпеки системи.

2.3 Аналіз альтернативних патернів для оптимізації

2.3.1 Альтернативні патерни Cache-Aside

У процесі оптимізації веб-застосунків кешування відіграє ключову роль у забезпеченні високої продуктивності та швидкого доступу до даних. Поряд із патерном Cache-Aside, який широко використовується для керування кешем на стороні застосунку, існують альтернативні патерни кешування, що пропонують різні підходи до взаємодії між кешем та базою даних. До таких патернів належать:

- Read-Through Cache;
- Write-Through Cache;
- Write-Around Cache;
- Write-Behind (Write-Back) Cache [4].

Розглядаючи альтернативні патерни до Cache-Aside, важливо звернути увагу на патерн Read-Through Cache, який також широко використовується для оптимізації доступу до даних у веб-застосунках. У патерні Read-Through Cache кеш виступає як прозорий проміжний шар між застосунком і базою даних. Коли застосунок запитує дані, він звертається безпосередньо до кешу. Якщо необхідні дані присутні в кеші (cache-hit), вони одразу повертаються застосунку. У випадку відсутності даних у кеші (cache-miss), кеш автоматично звертається до бази даних, отримує потрібні дані, зберігає їх у себе і повертає результат застосунку [5].

Переваги патерну Read-Through Cache відносно Cache-Aside:

- спрощення логіки застосунку, застосунок не потребує додаткової

логіки для роботи з кешем. Вся відповідальність за отримання та оновлення даних лежить на кеші, що зменшує складність коду та потенційні помилки, пов'язані з кешуванням;

– централізоване управління даними, кеш самостійно керує процесом отримання та оновлення даних, що дозволяє впроваджувати більш ефективні стратегії кешування, такі як автоматична експірація та політики оновлення;

– зменшення навантаження на базу даних, оскільки кеш обробляє всі запити до даних і звертається до бази даних лише при cache-miss, це знижує кількість прямих звернень до бази даних і покращує загальну продуктивність системи.

Недоліки патерну Read-Through Cache відносно Cache-Aside:

– менша гнучкість контролю, застосунок має менший контроль над процесом кешування. У патерні Cache-Aside застосунок сам вирішує, коли і які дані кешувати або оновлювати, що може бути критично в специфічних випадках використання;

– складність у реалізації специфічних логік, якщо необхідно впровадити нестандартну логіку отримання даних або обробки помилок, патерн Read-Through Cache може обмежувати можливості, оскільки кеш самостійно керує взаємодією з базою даних;

– потенційні проблеми з консистентністю даних, у випадку, коли дані можуть змінюватися поза контролем кешу (наприклад, при прямому оновленні бази даних іншими сервісами), існує ризик використання застарілих даних з кешу.

У контексті веб-застосунку "Телеграм-бот ІТКІ", де важливо забезпечити швидкий і надійний доступ до даних, а також підтримувати високу безпеку та актуальність інформації, вибір між патернами Read-Through Cache та Cache-Aside є особливо важливим. Патерн Cache-Aside надає більшого контролю над процесом кешування, що дозволяє застосунку самостійно визначати, які дані кешувати та коли їх оновлювати. Це особливо корисно при роботі з аутентифікаційними токенами та іншими критичними

даними, де необхідно забезпечити їх своєчасне оновлення або видалення з кешу після виходу користувача із системи.

З іншого боку, патерн Read-Through Cache може спростити архітектуру застосунку, знявши з нього відповідальність за управління кешем. Однак це може призвести до меншої гнучкості та потенційних проблем з консистентністю даних, що є небажаним у системах з високими вимогами до безпеки та актуальності інформації [6].

Враховуючи зазначені переваги та недоліки, у рамках даної роботи та специфіки "Телеграм-боту ІТКІ" доцільніше використовувати патерн Cache-Aside. Він забезпечує необхідну гнучкість для тонкого налаштування процесів кешування та оновлення даних, що є критичним для підвищення продуктивності застосунку та покращення користувацького досвіду. Крім того, це дозволить більш ефективно впровадити механізми безпеки, зокрема щодо управління аутентифікаційними токенами та запобігання їх несанкціонованому використанню.

Таким чином, хоча патерн Read-Through Cache і має свої переваги, його використання в даному контексті може бути менш ефективним порівняно з Cache-Aside. Обраний підхід сприятиме досягненню основних цілей роботи: оптимізації продуктивності, забезпеченню надійності та безпеки веб-застосунку "Телеграм-бот ІТКІ".

Продовжуючи аналіз альтернативних патернів до Cache-Aside, варто розглянути патерн Write-Through Cache. Якщо Read-Through Cache оптимізує операції читання шляхом прозорого отримання даних з бази при кеш-місі, то Write-Through Cache зосереджується на операціях запису, забезпечуючи синхронне оновлення кешу та бази даних при кожному записі.

У патерні Write-Through Cache застосунок записує дані безпосередньо в кеш. Кеш, у свою чергу, одразу ж оновлює базу даних, гарантує актуальність та консистентність даних між кешем і сховищем. Це відрізняється від Cache-Aside, де застосунок самостійно керує записом як у кеш, так і в базу даних.

Переваги патерну Write-Through Cache відносно Cache-Aside:

– гарантована консистентність даних, завдяки синхронному оновленню кешу та бази даних, дані завжди залишаються узгодженими. Це знижує ризик розходжень між кешем і базою, що може бути критичним у системах з високими вимогами до достовірності даних;

– спрощення взаємодії із сховищем, застосунок взаємодіє лише з кешем для всіх операцій, що спрощує логіку коду та зменшує ймовірність помилок при роботі з даними;

– автоматичне управління кешем, оскільки кеш самостійно оновлюється при записах, відпадає потреба в додаткових механізмах для інвалідації або оновлення кешу.

Недоліки патерну Write-Through Cache відносно Cache-Aside:

– зниження продуктивності запису, синхронне оновлення бази даних при кожному записі в кеш може збільшити час виконання операції, що негативно впливає на продуктивність при високій інтенсивності записів;

– підвищене навантаження на базу даних, постійні записні операції до бази даних можуть призвести до її перевантаження, особливо в системах з великим обсягом трафіку;

– складність масштабування, у розподілених системах забезпечення консистентності між кешем і базою даних може стати складним завданням, вимагатиме додаткових ресурсів і механізмів синхронізації.

У контексті веб-застосунку "Телеграм-бот ІТКІ", де швидкодія та масштабованість є ключовими вимогами, вибір патерну кешування має суттєвий вплив на ефективність системи. Write-Through Cache, забезпечуючи високу консистентність даних, може бути привабливим варіантом для систем, де точність даних превалює над продуктивністю.

Проте для "Телеграм-боту ІТКІ" важливою є швидкість обробки запитів та мінімізація затримок. Застосунок повинен оперативно реагувати на дії користувачів, надавати актуальну інформацію та підтримувати стабільну роботу при збільшенні навантаження. Використання Write-Through Cache може призвести до збільшення часу обробки записів і підвищити

навантаження на базу даних, що суперечить вимогам до продуктивності.

На відміну від цього, патерн Cache-Aside надає можливість оптимізувати взаємодію з кешем і базою даних, дозволяючи застосунку самостійно керувати процесами читання та запису. Це дає змогу:

– покращити продуктивність, зменшити кількість звернень до бази даних, особливо при операціях читання, за рахунок ефективного використання кешу;

– контролювати навантаження, регулювати частоту та обсяг записів у базу даних, що допомагає уникнути її перевантаження;

– гнучко управляти даними, реалізовувати специфічні логіки кешування та інвалідації даних, що важливо для безпеки та актуальності інформації.

Особливо це актуально для управління аутентифікаційними токенами та іншими конфіденційними даними. Патерн Cache-Aside дозволяє оперативно видаляти або оновлювати такі дані в кеші при зміні статусу користувача, наприклад, після виходу із системи. Це підвищує рівень безпеки та запобігає можливому несанкціонованому доступу.

Таким чином, хоча Write-Through Cache і забезпечує певні переваги в контексті консистентності даних, його недоліки щодо продуктивності та навантаження на базу даних роблять його менш привабливим для "Телеграм-боту ІТКІ". Патерн Cache-Aside більш відповідає потребам застосунку, надаючи необхідну гнучкість та контроль для оптимізації роботи з даними.

Вибір патерну Cache-Aside у поєднанні з використанням Redis для кешування дозволяє досягти основних цілей кваліфікаційної роботи магістра:

– підвищення продуктивності, швидкий доступ до часто запитуваних даних зменшує затримки та покращує користувацький досвід;

– оптимізація ресурсів, зниження навантаження на базу даних сприяє більш ефективному використанню серверних ресурсів;

– покращення безпеки, контроль над кешем дає можливість

реалізовувати механізми швидкого видалення конфіденційних даних.

У підсумку, врахування особливостей Write-Through Cache у порівнянні з Cache-Aside підтверджує доцільність вибору останнього для впровадження в "Телеграм-бот ІТКІ". Це рішення сприятиме створенню більш ефективного, надійного та безпечного веб-застосунку, що відповідає сучасним вимогам до якості програмного забезпечення.

Одним із альтернативних патернів до Cache-Aside є Write-Around Cache.

Патерн Write-Around Cache характеризується тим, що під час операцій запису дані безпосередньо зберігаються в основному сховищі, оминаючи кеш. Кеш оновлюється лише при читанні даних: якщо необхідні дані відсутні в кеші, вони завантажуються з основного сховища та додаються до кешу для подальших запитів. Таким чином, кешування відбувається "навколо" операцій запису [7].

Переваги Write-Around Cache порівняно з Cache-Aside:

– зменшення навантаження на кеш, оскільки операції запису не оновлюють кеш, це знижує кількість записів у кеш-пам'ять, що може бути корисним для систем із високою частотою записів та обмеженими ресурсами кешу;

– ефективне використання кешу для популярних даних, кеш зберігає лише ті дані, які запитуються для читання, що дозволяє ефективніше використовувати кеш для найпопулярніших або найчастіше запитуваних даних.

Недоліки Write-Around Cache порівняно з Cache-Aside:

– підвищена латентність при першому доступі до нових даних, оскільки нові або оновлені дані не зберігаються в кеші під час запису, перший запит до них вимагатиме звернення до основного сховища, що може збільшити час відгуку;

– ризик читання застарілих даних, якщо дані в основному сховищі оновилися, а в кеші все ще зберігається стара версія, це може призвести до неконсистентності та надання користувачам некоректної інформації;

– складність у підтримці консистентності даних, оскільки кеш не

оновлюється при записі, необхідно впроваджувати додаткові механізми для інвалідизації або оновлення кешу, щоб уникнути проблем із застарілими даними.

У контексті веб-застосунку "Телеграм-бот ІТКІ", де користувачі очікують негайного відображення змін та швидкого доступу до актуальної інформації, патерн Write-Around Cache може не відповідати вимогам системи. Затримка при першому читанні нових або оновлених даних може негативно вплинути на користувацький досвід.

Патерн Cache-Aside передбачає, що під час операцій запису дані зберігаються як в основному сховищі, так і в кеші. При читанні спочатку перевіряється кеш, і якщо дані відсутні, вони завантажуються з основного сховища та додаються до кешу. Це забезпечує швидкий доступ до даних та підтримує їх актуальність.

Переваги Cache-Aside у порівнянні з Write-Around Cache:

– швидкий доступ до нових та оновлених даних, оскільки кеш оновлюється під час запису, користувачі отримують негайний доступ до актуальної інформації без затримок;

– покращена консистентність даних, одночасне оновлення кешу та основного сховища знижує ризик читання застарілих даних;

– зменшення навантаження на основне сховище, часті читання відбуваються з кешу, що знижує кількість звернень до бази даних та покращує загальну продуктивність системи.

Враховуючи специфіку веб-застосунку "Телеграм-бот ІТКІ", де швидкодія та актуальність даних є критичними, патерн Cache-Aside є більш підходящим. Він забезпечує:

– негайне відображення змін для користувачів, користувачі можуть одразу бачити результати своїх дій, що покращує задоволеність від використання застосунку;

– зниження затримок при доступі до даних: Використання кешу для читання даних зменшує час відгуку системи;

– спрощене управління кешем, оновлення кешу під час запису спрощує підтримку консистентності та зменшує необхідність додаткових механізмів інвалідизації.

Таким чином, хоча Write-Around Cache може бути ефективним у системах з переважанням операцій запису та рідким доступом до даних, він не відповідає потребам веб-застосунку "Телеграм-бот ІТКІ". Патерн Cache-Aside забезпечує кращий баланс між продуктивністю та консистентністю даних, що є важливим для забезпечення високої якості користувацького досвіду та надійності системи.

Переходячи до подальшого аналізу альтернативних патернів кешування, варто детально розглянути патерн Write-Behind (або Write-Back) Cache. Цей підхід передбачає, що операції запису спочатку здійснюються в кеші, а не безпосередньо в основному сховищі даних. Оновлення основного сховища відбувається асинхронно, з певною затримкою або після накопичення певного обсягу змін. Таким чином, кеш стає основним місцем для операцій запису та читання, а база даних оновлюється у фоновому режимі.

Однією з головних переваг патерну Write-Behind Cache є підвищення швидкодії операцій запису. Оскільки запис до кешу зазвичай відбувається швидше, ніж до бази даних, користувачі відчувають миттєву реакцію системи на свої дії. Це особливо важливо для застосунків з високою інтенсивністю записів, де затримки можуть негативно вплинути на користувацький досвід [8].

Переваги патерну Write-Behind Cache:

– підвищення швидкодії записів, операції запису здійснюються в кеші з низькою латентністю, що забезпечує швидку реакцію системи;

– зменшення навантаження на основне сховище, асинхронне оновлення бази даних знижує кількість безпосередніх записів, дозволяючи основному сховищу ефективніше обробляти інші запити;

– оптимізація використання ресурсів, пакетне оновлення даних зменшує кількість транзакцій, що може підвищити загальну продуктивність

системи;

– кеш як єдине джерело даних, спрощує логіку читання та запису, оскільки всі операції відбуваються через кеш.

Проте цей патерн має суттєві недоліки, які можуть бути критичними для веб-застосунку "Телеграм-бот ІТКІ".

Недоліки патерну Write-Behind Cache:

– ризик втрати даних, у випадку збою кешу або системи до того, як зміни будуть записані в основне сховище, існує ймовірність втрати незбережених даних;

– проблеми з консистентністю даних, затримка між оновленням кешу та бази даних може призвести до того, що інші сервіси або компоненти отримуватимуть застарілу інформацію;

– складність реалізації, вимагає впровадження складних механізмів асинхронного запису, обробки помилок та забезпечення надійності передачі даних;

– ускладнене управління конфліктами, при одночасних змінах одних і тих самих даних різними користувачами може виникнути необхідність у складних механізмах синхронізації;

– високі вимоги до надійності кешу, кеш стає критичним компонентом системи, і його збій може призвести до серйозних проблем у роботі застосунку.

У контексті веб-застосунку "Телеграм-бот ІТКІ", де важливо забезпечити надійне та безпечне зберігання даних, ризик втрати інформації є неприйнятним. Користувачі очікують, що їхні дії будуть збережені та відображені коректно в будь-який момент часу. Втрата даних може серйозно підірвати довіру до системи та негативно вплинути на її репутацію.

Крім того, проблеми з консистентністю даних можуть призвести до того, що різні користувачі отримуватимуть суперечливу або застарілу інформацію. Це може спричинити плутанину, помилки в роботі бота та незадоволеність користувачів. Враховуючи, що веб-застосунок "Телеграм-бот ІТКІ" може

взаємодіяти з іншими сервісами або компонентами, затримка в оновленні бази даних може призвести до серйозних проблем з інтеграцією.

Складність реалізації патерну Write-Behind Cache також є значним недоліком. Необхідність впровадження надійних механізмів асинхронного запису, моніторингу черги змін та обробки можливих помилок ускладнює архітектуру застосунку. Це може призвести до збільшення витрат на розробку, підтримку та тестування системи.

Зважаючи на ці недоліки, патерн Write-Behind Cache не є оптимальним вибором для веб-застосунку "Телеграм-бот ІТКІ". Натомість, патерн Cache-Aside пропонує більш збалансований підхід, який поєднує в собі високу продуктивність, надійність та простоту реалізації.

2.3.2 Альтернативні патерни Claim-Check

У процесі оптимізації веб-застосунку "Телеграм-бот ІТКІ" особливу увагу було приділено впровадженню патерну Claim-Check для ефективного управління файлами та зменшення навантаження на систему. Однак сучасні програмні системи потребують гнучкості та здатності адаптуватися до різних умов експлуатації. Тому важливо розглянути альтернативні патерни, які можуть забезпечити подібні або навіть кращі результати в певних контекстах.

Аналіз альтернативних патернів дозволяє обрати найбільш оптимальне рішення для конкретних завдань, враховуючи особливості архітектури та вимоги до продуктивності, масштабованості та безпеки. Розгляд різних підходів також сприяє більш глибокому розумінню можливостей системи та шляхів її вдосконалення.

Серед можливих альтернатив патерну Claim-Check виділяються наступні патерни:

– Split and Aggregate Pattern;
– Content Enricher Pattern;
– Content Filter Pattern;

– Process Manager Pattern.

У процесі дослідження та вдосконалення веб-застосунку "Телеграм-бот ІТКІ" важливо розглянути альтернативні патерни до Claim-Check, зокрема патерн Split and Aggregate. Цей патерн пропонує розподіл великих завдань на менші, незалежні підзадачі з подальшим об'єднанням результатів. Він використовується для підвищення продуктивності та масштабованості систем, особливо коли необхідно обробляти великі обсяги даних або виконувати складні обчислення [9].

Патерн Split and Aggregate передбачає:

– розділення завдання, велике завдання ділиться на кілька менших, які можуть виконуватися паралельно;

– паралельна обробка, підзадачі розподіляються між різними обчислювальними ресурсами для одночасного виконання;

– агрегування результатів, після завершення обробки результати всіх підзадач об'єднуються для отримання кінцевого результату.

Переваги патерну Split and Aggregate:

– підвищення продуктивності, паралельна обробка зменшує загальний час виконання завдання;

– масштабованість, система легко адаптується до збільшення навантаження шляхом додавання нових ресурсів;

– гнучкість, незалежність підзадач спрощує оновлення та модифікацію системи;

– розподіленість, знижується ризик єдиної точки відмови завдяки розподілу завдань.

Недоліки патерну Split and Aggregate:

– складність реалізації, потребує додаткових зусиль для управління розподілом та синхронізацією;

– управління залежностями, можливі складнощі з підзадачами, що залежать одна від одної;

– додаткові накладні витрати, розподіл і об'єднання можуть додавати затримки;

– проблеми з узгодженістю даних, потрібно забезпечити цілісність даних при агрегуванні.

Однак, у контексті веб-застосунку "Телеграм-бот ІТКІ", застосування патерну Split and Aggregate є менш доцільним. Це пов'язано з тим, що даний застосунок не містить функціоналу, який вимагає розподілу великих обчислювальних завдань на підзадачі, таких як генерація статистики або аналітика. Основні завдання бота зосереджені на наданні доступу до сервісів через Telegram та управлінні даними користувачів, що не потребує складної паралельної обробки.

Натомість, патерн Claim-Check є більш необхідним для цього застосунку. Claim-Check передбачає зберігання файлів у зовнішньому сховищі та передачу між компонентами системи лише посилань на ці дані. Це дозволяє зменшити навантаження на мережу та підвищити ефективність передачі інформації.

Переваги патерну Claim-Check для веб-застосунку "Телеграм-бот ІТКІ":

– ефективне управління великими даними, зменшується обсяг даних, що передається між сервісами;

– покращення продуктивності, зменшується затримка при передачі файлів, що підвищує швидкодію бота;

– підвищення безпеки, централізоване зберігання даних дозволяє краще контролювати доступ;

– зниження навантаження на систему, менше ресурсів витрачається на передачу даних, що оптимізує роботу бота.

У веб-застосунку "Телеграм-бот ІТКІ" використання Claim-Check є особливо корисним для оптимізації процесів зберігання та обробки файлів. Наприклад, при передачі документів або зображень від адміністраторів при масових розсилках повідомлень, зберігання цих файлів у зовнішньому

сховищі (наприклад, Minio) і передача посилань дозволяє швидше та безпечніше оперувати даними.

Claim-Check краще підходить для даного застосунку через такі аспекти як:

– відсутність потреби в паралельній обробці, оскільки бот не виконує складних обчислень, переваги Split and Aggregate не будуть реалізовані;

– фокус на управлінні даними, основні проблеми бота пов'язані з ефективним зберіганням та передачею даних, що вирішується за допомогою Claim-Check;

– простота впровадження, Claim-Check менш складний у реалізації порівняно зі Split and Aggregate, що зменшує час та ресурси на розробку;

– підвищення безпеки та надійності, Centralізоване зберігання файлів спрощує контроль доступу та резервне копіювання.

Таким чином, у контексті вдосконалення архітектури веб-застосунку "Телеграм-бот ІТКІ", патерн Claim-Check є більш підходящим та ефективним рішенням. Він дозволяє вирішити виявлені проблеми з неефективним управлінням даними та оптимізувати процеси зберігання і передачі файлів. Використання Redis для кешування даних за патерном Cache-Aside та інтеграція Minio як сховища файлів відповідно до патерну Claim-Check забезпечить зниження навантаження на основну базу даних і підвищить швидкодію застосунку.

У підсумку, хоча патерн Split and Aggregate має свої переваги в певних умовах, для "Телеграм-бота ІТКІ" більш доцільно зосередитися на впровадженні патерну Claim-Check. Це рішення сприятиме підвищенню продуктивності, ефективності використання ресурсів та покращенню безпеки даних, що в кінцевому рахунку забезпечить кращий користувацький досвід і підвищить конкурентоспроможність застосунку.

У контексті вдосконалення архітектури веб-застосунку "Телеграм-бот ІТКІ" варто також розглянути патерн Content Enricher як альтернативу Claim-Check. Цей патерн дозволяє збагачувати повідомлення або дані додатковою

інформацією з зовнішніх джерел перед їх подальшою обробкою чи передачею. Застосування Content Enricher може покращити якість наданих сервісів, забезпечивши користувачів більш повною та актуальною інформацією [10].

Переваги патерну Content Enricher у порівнянні з Claim-Check полягають у можливості:

– збагачення даних, дозволяє доповнювати наявні дані новою інформацією, що підвищує їх цінність для кінцевого користувача;

– підвищення точності, отримання актуальних даних з зовнішніх джерел сприяє більш точній та релевантній відповіді на запити користувачів;

– гнучкість інтеграції, легко інтегрується з різними сервісами та API, що розширює функціональні можливості застосунку.

Однак, у контексті веб-застосунку "Телеграм-бот ІТКІ", застосування патерну Content Enricher може бути менш ефективним порівняно з Claim-Check через наступні причини:

– збільшення затримок, додаткові звернення до зовнішніх джерел можуть уповільнити роботу бота, що негативно вплине на користувацький досвід;

– складність реалізації, необхідність інтеграції з різними сервісами ускладнює архітектуру та вимагає додаткових ресурсів на розробку та підтримку;

– залежність від сторонніх сервісів, робота бота стає залежною від доступності та надійності зовнішніх джерел даних, що може призвести до непередбачуваних збоїв.

Порівняно з Content Enricher, патерн Claim-Check є більш доцільним для даного застосунку, оскільки він оптимізує управління великими обсягами даних та підвищує ефективність системи без додаткових затримок. Claim-Check дозволяє зберігати великі файли у зовнішньому сховищі та передавати між компонентами системи лише посилання на них, що знижує навантаження на мережу та серверні ресурси. Це особливо важливо для "Телеграм-бота

ІТКІ", де швидкодія та надійність є критичними факторами.

З огляду на специфіку застосунку, основні задачі якого не передбачають збагачення даних з зовнішніх джерел, впровадження Content Enricher не принесе суттєвих переваг. Натомість, використання Claim-Check сприятиме оптимізації процесів зберігання та передачі файлів, підвищуючи продуктивність та масштабованість системи.

Таким чином, аналіз альтернативних патернів показує, що Claim-Check є найбільш відповідним для вирішення наявних проблем веб-застосунку "Телеграм-бот ІТКІ". Він забезпечує ефективне управління даними, покращує швидкодію та знижує навантаження на системні ресурси, що в кінцевому результаті покращує користувацький досвід і підвищує надійність роботи бота.

Також доцільно розглянути патерн Content Filter як альтернативу Claim-Check. Цей патерн призначений для фільтрації вхідних повідомлень або даних, видаляючи або модифікуючи непотрібну чи небажану інформацію перед подальшою обробкою. Застосування Content Filter може підвищити безпеку та ефективність системи, забезпечуючи, що лише релевантні дані надходять до внутрішніх компонентів [11].

Переваги патерну Content Filter у порівнянні з Claim-Check полягають у наступному:

– покращення безпеки, фільтруючи шкідливий або небажаний контент, патерн допомагає захистити систему від потенційних атак та зловживань;

– зниження навантаження на систему, видаляючи зайву інформацію, зменшується обсяг даних для обробки, що підвищує продуктивність;

– підвищення якості даних, забезпечує, що тільки коректна та релевантна інформація потрапляє до основних сервісів, покращуючи загальну якість роботи;

– гнучкість налаштувань, можливість налаштувати фільтри під специфічні потреби та вимоги застосунку.

Однак, у випадку "Телеграм-бот ІТКІ", застосування патерну Content Filter може бути менш доцільним порівняно з Claim-Check через такі причини:

– обмежена потреба у фільтрації, якщо бот не обробляє великий обсяг неконтрольованих вхідних даних, необхідність у складному фільтруванні знижується;

– додаткові витрати на реалізацію, впровадження Content Filter може вимагати значних ресурсів для розробки та підтримки, що не виправдовує себе при низькій потребі;

– ризик впливу на користувацький досвід, надмірне або неправильне фільтрування може призвести до блокування корисної інформації, що негативно позначиться на взаємодії з користувачами.

Порівнюючи з Claim-Check, який спеціалізується на ефективному управлінні великими файлами та оптимізації передачі даних, Content Filter не вирішує основних проблем веб-застосунку "Телеграм-бот ІТКІ". Claim-Check дозволяє зберігати великі файли у зовнішньому сховищі та передавати між компонентами лише посилання на них, що знижує навантаження на мережу та підвищує швидкодію. Це особливо важливо для бота, де ефективність передачі даних та швидка реакція на запити користувачів є критичними.

З огляду на специфіку застосунку, основні задачі якого не передбачають інтенсивної фільтрації вхідних даних, впровадження патерну Content Filter не принесе суттєвих переваг. Натомість, фокусування на Claim-Check є більш раціональним, оскільки він безпосередньо вирішує виявлені проблеми з неефективним управлінням даними та оптимізує процеси зберігання і передачі файлів.

Таким чином, аналіз патерну Content Filter підтверджує, що Claim-Check є найбільш відповідним для покращення архітектури "Телеграм-бота ІТКІ". Застосування цього патерну сприятиме підвищенню продуктивності, ефективному використанню ресурсів та забезпеченню високого рівня безпеки даних, що в кінцевому підсумку покращить користувацький досвід і підвищить надійність роботи системи.

Варто також розглянути патерн Process Manager як альтернативу Claim-Check. Цей патерн призначений для управління складними бізнес-процесами, які включають кілька кроків або транзакцій, забезпечуючи їх узгодженість і цілісність. Process Manager діє як координатор, який відстежує стан процесу, керує послідовністю виконання завдань та обробляє можливі винятки чи збої [12].

Переваги патерну Process Manager у порівнянні з Claim-Check полягають у наступному:

– керування складними процесами, дозволяє організувати і контролювати багатокрокові транзакції або робочі процеси, забезпечуючи їх послідовне та узгоджене виконання;

– відновлення після збоїв, забезпечує механізми для обробки помилок і відновлення процесу з точки, де стався збій, що підвищує надійність системи;

– масштабованість, може координувати процеси, які виконуються на різних вузлах або сервісах, що сприяє розподіленій архітектурі;

– гнучкість налаштувань, дозволяє налаштовувати послідовність та логіку виконання процесів відповідно до бізнес-вимог.

Проте, у випадку веб-застосунку "Телеграм-бот ІТКІ", застосування патерну Process Manager може бути менш ефективним порівняно з Claim-Check через наступні причини:

– відсутність складних процесів, якщо бот не виконує багатокрокових транзакцій або складних бізнес-процесів, необхідність у Process Manager знижується;

– додаткова складність, впровадження Process Manager додає рівень складності до архітектури, що може бути невиправданим при простих сценаріях використання;

– затримки у виконанні, координація процесів може призводити до додаткових затримок, що негативно вплине на швидкодію бота;

– ресурсні витрати, потребує додаткових обчислювальних ресурсів для відстеження та управління процесами.

Порівнюючи з Claim-Check, який оптимізує управління великими файлами та передачею даних, Process Manager не адресує основних проблем веб-застосунку "Телеграм-бот ІТКІ". Claim-Check дозволяє зберігати великі файли у зовнішньому сховищі та обмінюватися лише посиланнями на них, що зменшує навантаження на мережу та прискорює взаємодію між компонентами системи. Це особливо важливо для бота, де швидкість відповіді та ефективність обробки запитів є критичними.

З огляду на специфіку застосунку, де основні задачі не вимагають складного управління процесами, впровадження патерну Process Manager може бути невиправданим. Натомість, фокус на Claim-Check дозволить вирішити актуальні проблеми з неефективним управлінням даними та оптимізувати роботу з файлами.

Таким чином, аналіз альтернативних патернів підтверджує, що Claim-Check є найбільш підходящим для покращення архітектури веб-застосунку "Телеграм-бот ІТКІ". Застосування цього патерну сприятиме:

– підвищенню продуктивності, швидша передача даних та зменшення навантаження на систему;

– оптимізації ресурсів, ефективне використання мережевих та обчислювальних ресурсів;

– забезпеченню безпеки, централізоване зберігання даних полегшує контроль доступу та захист інформації;

– покращенню користувацького досвіду, швидкодія та надійність бота сприяють задоволенню потреб користувачів.

У підсумку, хоча патерн Process Manager має свої переваги в управлінні складними процесами, його застосування в "Телеграм-боті ІТКІ" не є необхідним через відсутність відповідних вимог. Зосередження на впровадженні Claim-Check дозволить ефективно вирішити наявні проблеми та досягти цілей оптимізації, забезпечуючи високий рівень продуктивності та

надійності веб-застосунку.

2.4 Огляд технологій Redis та Minio

2.4.1 Особливості Redis для кешування

У сучасних веб-застосунках кешування відіграє критично важливу роль у забезпеченні високої швидкодії та масштабованості системи. Зі зростанням кількості користувачів і обсягу даних виникає потреба в ефективних механізмах зберігання та доступу до інформації. Кешування дозволяє зменшити навантаження на базу даних, скоротити час відгуку та підвищити загальну продуктивність застосунку.

Вибір ефективного механізму кешування є ключовим для успішної реалізації високопродуктивних та масштабованих систем. Існує безліч рішень для кешування, включаючи вбудовані механізми у фреймворках, таких як Spring Boot, а також сторонні інструменти, наприклад:

– Caffeine;
– Ehcache;
– Hazelcast;
– Memcached;
– Redis.

Кожне з цих рішень має свої особливості, переваги та обмеження, які впливають на їхню придатність для конкретних завдань.

Redis, як високопродуктивне in-memory сховище даних, пропонує розширені можливості для кешування в сучасних веб-застосунках. Завдяки своїй швидкодії, масштабованості та підтримці різноманітних структур даних, Redis стає привабливим вибором для розробників, які прагнуть оптимізувати роботу своїх систем [13].

Caffeine Cache є сучасною високопродуктивною бібліотекою кешування для Java, яка призначена для забезпечення максимальної ефективності та гнучкості в роботі з кешем у застосунках. Розроблена як наступник популярної

бібліотеки Guava Cache, Caffeine використовує передові алгоритми та структури даних для досягнення високої швидкодії та низької затримки при доступі до кешованих даних.

Caffeine Cache надає розробникам можливість легко інтегрувати кешування в свої Java-застосунки, забезпечуючи при цьому широкий спектр налаштувань та політик керування кешем. Однією з ключових особливостей Caffeine є використання адаптивних алгоритмів для управління кешем, які автоматично підлаштовуються під робоче навантаження та патерни доступу до даних [14].

Бібліотека підтримує різні стратегії виселення елементів з кешу, такі як:

– W-TinyLFU (Windowed Tiny Least Frequently Used) – поєднує переваги політик LRU (Least Recently Used) та LFU (Least Frequently Used), забезпечуючи високу точність та ефективність;

– Time-based expiration – встановлення часу життя (TTL) для елементів кешу, після якого вони автоматично видаляються;

– Size-based eviction – контроль розміру кешу на основі кількості елементів або обсягу пам'яті, що використовується.

Caffeine Cache також пропонує асинхронне завантаження даних, що дозволяє уникнути блокування потоків при запиті даних, яких немає в кеші. Це досягається за допомогою використання CompletableFuture та інших асинхронних механізмів Java.

Переваги Caffeine Cache:

– висока продуктивність, завдяки оптимізованим алгоритмам та використанню структур даних, таких як пробуджувані хеш-таблиці, Caffeine забезпечує мінімальні затримки при доступі до кешу;

– гнучкість налаштувань, підтримка різних політик виселення, налаштування розміру кешу, часового життя елементів та інших параметрів дозволяє адаптувати кешування під специфічні потреби застосунку;

– простота інтеграції, зручний та інтуїтивний API робить процес

впровадження кешування швидким та нескладним;

– асинхронність, можливість асинхронного завантаження даних з джерела, що підвищує масштабованість та реактивність застосунку;

– статистика та моніторинг, вбудовані засоби збору статистики дозволяють відстежувати ефективність кешування та оптимізувати налаштування;

– підтримка Java 8 та новіших версій, використання сучасних можливостей мови для підвищення ефективності та зручності використання.

Недоліки Caffeine Cache:

– локальність кешу, Caffeine працює в межах одного екземпляру JVM, що означає, що в розподілених системах кеш не синхронізується між різними інстансами застосунку;

– відсутність персистентності, кеш зберігається виключно в оперативній пам'яті та втрачається при перезапуску застосунку або виникненні збоїв;

– обмеження пам'яті, при великих обсягах даних кеш може споживати значну кількість оперативної пам'яті, що може вимагати додаткової оптимізації або збільшення ресурсів;

– не підходить для міжмовних застосунків, оскільки Caffeine написана на Java, її не можна використовувати безпосередньо в застосунках, написаних на інших мовах програмування;

– відсутність вбудованої підтримки кластеризації, для реалізації розподіленого кешування необхідно використовувати додаткові інструменти або власні рішення для синхронізації кешу між вузлами.

Caffeine Cache є чудовим вибором для Java-застосунків, які потребують високопродуктивного локального кешування з гнучкими налаштуваннями та низькою затримкою. Завдяки своїй простоті та ефективності, вона підходить для широкого спектру задач, від простих настільних програм до високонавантажених серверних застосунків.

Однак для систем, які вимагають розподіленого кешування або зберігання даних між перезапусками, Caffeine може бути недостатньо. У таких випадках варто розглянути інші рішення, такі як Redis або Hazelcast, які надають можливості розподіленого кешування та персистентності.

Ehcache є одним із найбільш популярних і зрілих механізмів кешування для Java-застосунків, який забезпечує потужні можливості для покращення продуктивності та масштабованості системи. Розроблений як відкрите рішення, Ehcache пропонує гнучкі та надійні засоби для локального та розподіленого кешування, а також підтримує персистентність даних, що дозволяє зберігати кеш між перезапусками застосунку [15].

Ehcache інтегрується з різними фреймворками та технологіями, такими як Spring, Hibernate та інші, що робить його привабливим вибором для широкого спектру застосунків. Він підтримує різні стратегії керування кешем, включаючи обмеження розміру, час життя елементів, стратегії витіснення та інші налаштування, які дозволяють точно налаштувати кешування відповідно до потреб застосунку.

Однією з ключових особливостей Ehcache є його архітектура, яка підтримує багаторівневе кешування. Це означає, що дані можуть зберігатися як у пам'яті (heap), так і поза пам'яттю (off-heap), а також на диску. Такий підхід дозволяє ефективно використовувати ресурси системи та забезпечувати високу продуктивність навіть при роботі з великими обсягами даних.

Ehcache також підтримує кластеризацію через Terracotta Server Array, що дозволяє синхронізувати кеші між декількома інстансами застосунку. Це особливо корисно в розподілених системах, де необхідно забезпечити консистентність даних та високу доступність сервісів.

Переваги Ehcache:

– простота інтеграції, легко інтегрується з Spring Boot та іншими фреймворками завдяки підтримці Spring Cache Abstraction;

– підтримка персистентності, можливість зберігати кешовані дані на диску, що забезпечує збереження стану кешу між перезапусками застосунку;

– розширені налаштування, підтримка різних політик витіснення (LRU, LFU, FIFO тощо), обмеження розміру кешу, налаштування часу життя елементів та інші гнучкі параметри;

– масштабованість та кластеризація, можливість розподіленого кешування через Terracotta Server, що забезпечує синхронізацію кешу між різними вузлами та високу доступність;

– відповідність стандартам, підтримка JSR-107 (Java Caching Standard), що спрощує інтеграцію та міграцію між різними реалізаціями кешу;

– моніторинг та управління, вбудовані засоби для відстеження стану кешу, статистики та налаштування параметрів в режимі реального часу;

– широка спільнота та підтримка, активна спільнота розробників, велика кількість документації, прикладів та ресурсів для навчання.

Недоліки Ehcache:

– складність налаштування, може вимагати більше часу та зусиль для початкового налаштування, особливо при використанні розподіленого кешування та персистентності;

– витрати ресурсів, використання персистентності та кластеризації може призвести до збільшення споживання пам'яті та процесорного часу;

– залежність від додаткових компонентів, для розподіленого кешування необхідно розгортати та підтримувати Terracotta Server, що додає складності до інфраструктури;

– ліцензування, деякі розширені функції, особливо пов'язані з кластеризацією та високою доступністю, можуть вимагати комерційної ліцензії.

– менша продуктивність у порівнянні з in-memory рішеннями, при використанні персистентності та дискового зберігання продуктивність може бути нижчою, ніж у чисто in-memory кешів, таких як Caffeine або Redis.

– відсутність асинхронного завантаження, на відміну від деяких інших кешів, Ehcache не підтримує асинхронне завантаження даних за

замовчуванням.

Ehcache є потужним інструментом для реалізації кешування в Java-застосунках, який надає широкі можливості для налаштування та масштабування. Його здатність працювати як у локальному режимі, так і в кластері робить його універсальним рішенням для різних сценаріїв використання.

Також варто звернути увагу на ліцензійні умови. Хоча базова функціональність Ehcache є безкоштовною та відкритою, деякі розширені можливості, особливо пов'язані з кластеризацією та високою доступністю через Terracotta Server, можуть потребувати придбання комерційної ліцензії. Це може вплинути на загальну вартість рішення та вимагати додаткових ресурсів для підтримки інфраструктури.

Ehcache підходить для застосунків, які потребують гнучкого та налаштованого кешування з можливістю персистентності та розподіленості. Однак, веб-застосунок "Телеграм-боті ІТКІ" не вимагає таких складних функцій та потребує більшу швидкодію як в in-memory кеш тому варто розглянути альтернативу як Redis.

Hazelcast є потужною платформою для розподілених обчислень та зберігання даних, яка надає розширені можливості для реалізації розподіленого кешування в Java-застосунках. Це відкрите програмне забезпечення, яке дозволяє створювати масштабовані та високопродуктивні системи, забезпечуючи при цьому простоту інтеграції та використання. Hazelcast надає інструменти для реалізації розподілених структур даних, таких як карти, черги, множини та інші, які автоматично синхронізуються між вузлами кластера.

Hazelcast дозволяє розробникам легко створювати розподілені кеші, які можуть масштабуватися горизонтально шляхом додавання нових вузлів до кластера. Завдяки цьому досягається висока доступність та відмовостійкість системи. Hazelcast використовує архітектуру peer-to-peer без єдиної точки відмови, що підвищує надійність та ефективність обміну даними між вузлами.

Платформа підтримує різні механізми розподілу даних, включаючи партиціонування та реплікацію, що дозволяє оптимізувати розподіл навантаження та забезпечити консистентність даних. Hazelcast також надає можливість інтеграції з різними фреймворками та технологіями, такими як Spring Boot, що спрощує процес впровадження кешування в застосунок [16].

Однією з важливих особливостей Hazelcast є підтримка багатьох мов програмування, включаючи Java, .NET, C++ та інші, що дозволяє використовувати його в міжмовних та гетерогенних системах. Крім того, Hazelcast надає клієнт-серверну та вбудовану режими роботи, що дозволяє налаштувати систему відповідно до специфічних потреб застосунку.

Hazelcast підтримує транзакційність та забезпечує ACID-властивості для операцій з даними, що важливо для критично важливих застосунків. Також платформа надає можливості для обробки потокових даних та виконання розподілених обчислень за допомогою Hazelcast Jet.

Переваги Hazelcast:

– масштабованість та відмовостійкість, можливість горизонтального масштабування шляхом додавання нових вузлів до кластера без переривання роботи системи;

– розподілені структури даних, підтримка розподілених карт, черг, множин та інших колекцій, які автоматично синхронізуються між вузлами;

– висока продуктивність, низька затримка доступу до даних завдяки зберіганню їх в оперативній пам'яті та ефективним алгоритмам розподілу;

– простота інтеграції, легка інтеграція з Spring Boot та іншими фреймворками, підтримка Spring Cache Abstraction;

– підтримка транзакційності, забезпечення ACID-властивостей для операцій з даними, підтримка розподілених транзакцій;

– гнучкість конфігурації, можливість налаштування різних параметрів системи, таких як партиціонування, реплікація, політики кешування;

– підтримка багатьох мов, клієнтські бібліотеки для Java, .NET, C++ та інших мов, що дозволяє використовувати Hazelcast у міжмовних системах;

– моніторинг та управління, вбудовані засоби для моніторингу стану кластера, збору статистики та налаштування параметрів у реальному часі;

– відсутність єдиної точки відмови, peer-to-peer архітектура забезпечує високу надійність та доступність системи;

– розширюваність, можливість додавання власних сервісів та обробників подій, інтеграція з іншими технологіями та сервісами.

Недоліки Hazelcast:

– складність налаштування, може вимагати значних зусиль для початкового налаштування та оптимізації, особливо в складних розподілених середовищах;

– витрати ресурсів, високі вимоги до оперативної пам'яті та процесорного часу при роботі з великими обсягами даних та великою кількістю вузлів;

– відсутність персистентності за замовчуванням, дані зберігаються в оперативній пам'яті, що може призвести до їх втрати при збоях або перезапусках; для забезпечення персистентності необхідно використовувати додаткові механізми;

– складність діагностики, у великих кластерах може бути складно відстежувати та діагностувати проблеми, пов'язані з розподілом даних та мережевими затримками;

– відсутність стандартних API для деяких функцій, може вимагати використання специфічних для Hazelcast API, що ускладнює міграцію або інтеграцію з іншими системами;

– можливі проблеми з консистентністю, у розподілених системах завжди існує ризик виникнення проблем з узгодженістю даних, особливо при високому навантаженні або збоях мережі;

– ліцензування, деякі розширені функції та інструменти доступні

лише в комерційній версії Hazelcast Enterprise, що може вплинути на бюджет проекту;

– крута крива навчання, для ефективного використання всіх можливостей Hazelcast необхідно глибоке розуміння розподілених систем та принципів роботи платформи;

– обмежена підтримка SQL, хоча Hazelcast надає можливості для виконання запитів, підтримка SQL може бути обмеженою порівняно з спеціалізованими базами даних.

Hazelcast є потужним рішенням для реалізації розподіленого кешування та обробки даних в пам'яті, що підходить для застосунків, які потребують високої масштабованості та продуктивності. Його архітектура дозволяє створювати системи без єдиної точки відмови, забезпечуючи при цьому високу доступність сервісів. Hazelcast добре інтегрується з Java-застосунками та надає широкі можливості для налаштування та розширення.

Проте, використання Hazelcast може бути пов'язане зі складнощами в налаштуванні та підтримці, особливо для команд без досвіду роботи з розподіленими системами. Високі вимоги до ресурсів та потенційні проблеми з консистентністю даних також можуть стати перешкодою для деяких проектів. Крім того, для доступу до розширених функцій може знадобитися придбання комерційної ліцензії, що варто враховувати при плануванні бюджету.

Hazelcast є потужним та гнучким інструментом для реалізації розподіленого кешування та обробки даних в пам'яті. Він підходить для великих та складних систем, які потребують високої продуктивності, масштабованості та відмовостійкості. Проте, його використання може бути складним та вимагати додаткових ресурсів для налаштування та підтримки. Тому Hazelcast як рішення для кешування не є оптимальним для веб-застосунку "Телеграм-боті ІТКІ".

Memcached є високопродуктивною, розподіленою системою кешування в пам'яті, яка використовується для прискорення динамічних веб-застосунків

шляхом зменшення навантаження на базу даних. Вона зберігає дані та об'єкти в оперативній пам'яті, що дозволяє швидко отримувати їх без необхідності звертатися до повільніших сховищ. Memcached був розроблений для покращення продуктивності великих систем та широко використовується такими компаніями, як Facebook, Twitter, Wikipedia та іншими.

Memcached працює за принципом простого ключ-значення, де дані зберігаються у вигляді пар ключ-значення в пам'яті. Він не підтримує складні структури даних або запити, як це роблять деякі інші кеші або бази даних. Натомість, його фокус полягає на максимально швидкому зберіганні та отриманні даних за ключем. Memcached використовує розподілену архітектуру, де дані розподіляються між декількома вузлами, що дозволяє масштабувати систему горизонтально.

Однією з головних переваг Memcached є його простота та легкість у використанні. Він підтримує клієнтські бібліотеки для багатьох мов програмування, включаючи Java, PHP, Python, Ruby та інші. Це робить його універсальним інструментом для кешування в різних типах застосунків. Memcached не вимагає складної конфігурації або налаштування, що дозволяє швидко розпочати його використання [17].

Переваги Memcached:

– висока продуктивність, Memcached зберігає дані в оперативній пам'яті, що забезпечує надзвичайно швидкий доступ до кешованих даних з мінімальною затримкою;

– простота використання, легкий у встановленні та конфігурації, має простий протокол текстових команд, який легко розуміти та використовувати;

– масштабованість, підтримує горизонтальне масштабування шляхом додавання нових вузлів до кластера, що дозволяє обробляти великі обсяги даних та навантаження;

– розподіленість, дані автоматично розподіляються між вузлами, що забезпечує балансування навантаження та ефективне використання ресурсів;

– підтримка багатьох мов, клієнтські бібліотеки доступні для широкого спектра мов програмування, що робить Memcached універсальним рішенням для різних проектів;

– відкритий код та безкоштовність, Memcached є програмним забезпеченням з відкритим кодом, доступним безкоштовно, що знижує витрати на впровадження;

– низьке споживання ресурсів, Memcached спроектований для ефективного використання оперативної пам'яті та процесорного часу, що дозволяє розгортати його на серверах з обмеженими ресурсами;

– стабільність та надійність, довгий час існування та активна спільнота розробників забезпечують стабільність та надійність роботи системи;

– підтримка простих операцій, швидкі операції встановлення, отримання та видалення даних за ключем роблять Memcached ідеальним для простих сценаріїв кешування.

Недоліки Memcached:

– відсутність персистентності, дані зберігаються лише в оперативній пам'яті і втрачаються при перезапуску сервера або збоях, що може бути критичним для деяких застосунків;

– обмежена функціональність, підтримує лише прості операції ключ-значення без можливості роботи зі складними структурами даних або виконання запитів;

– відсутність реплікації та відмовостійкості, немає вбудованого механізму реплікації даних між вузлами, що може призвести до втрати даних при виході з ладу вузла;

– проблеми з консистентністю, у розподілених системах можливі ситуації, коли дані не синхронізовані між клієнтами, що може призвести до непередбачуваної поведінки;

– обмеження на розмір ключів та значень, існують обмеження на максимальний розмір ключа та значення, що може бути недостатнім для деяких

сценаріїв;

– відсутність механізмів безпеки, Memcached не має вбудованих засобів аутентифікації та шифрування, що вимагає додаткових заходів для захисту даних;

– не підходить для складних даних, неможливість працювати зі складними структурами даних, такими як списки, множини або хеші, обмежує застосування Memcached;

– відсутність автоматичного балансування, Memcached не надає вбудованих засобів для автоматичного балансування навантаження між вузлами;

– необхідність керування клієнтськими бібліотеками, різні клієнтські бібліотеки можуть мати різні особливості та вимоги, що може ускладнювати розробку та підтримку.

При використанні Memcached важливо враховувати, що він не підтримує складні операції або структури даних. Це означає, що серіалізація та десеріалізація об'єктів повинні виконуватися ефективно, щоб не знижувати продуктивність системи. Крім того, оскільки дані зберігаються в пам'яті і не зберігаються, необхідно бути готовим до можливості їх втрати та розробити механізми для повторного заповнення кешу. Тому Memcached не є ефективним рішенням для веб-застосунку "Телеграм-боті ІТКІ".

Redis є високопродуктивним in-memory сховищем даних з відкритим кодом, яке підтримує різноманітні структури даних і надає розширені можливості для кешування, обробки та зберігання даних. Розроблений як NoSQL база даних, Redis поєднує в собі швидкодію пам'яті та функціональність персистентності, що дозволяє зберігати дані на диску та забезпечувати їх доступність після перезапуску системи.

Redis підтримує різні типи даних, включаючи рядки, хеші, списки, множини, відсортовані множини, бітові поля та геопросторові індекси. Це дозволяє розробникам використовувати найбільш підходящу структуру даних для конкретного завдання, що підвищує ефективність зберігання та обробки

інформації. Крім того, Redis надає можливості для виконання атомарних операцій, скриптів на мові Lua, транзакцій та публікації-підписки (Pub/Sub), що розширює сферу його застосування.

Однією з ключових особливостей Redis є його висока продуктивність. Завдяки зберіганню даних в оперативній пам'яті та ефективним алгоритмам обробки, Redis забезпечує дуже низьку затримку доступу до даних і високу пропускну здатність. Це робить його ідеальним вибором для задач, що вимагають швидкого доступу до даних, таких як кешування, керування сесіями, аналітика в реальному часі та інші.

Redis підтримує реплікацію даних, що забезпечує високу доступність та відмовостійкість системи. Основний вузол може мати декілька підлеглих, які автоматично синхронізуються з ним. У випадку збою основного вузла, підлеглий може стати новим основним, що мінімізує час простою системи. Redis також підтримує кластеризацію, що дозволяє розподіляти дані між декількома вузлами та масштабувати систему горизонтально.

Збереження даних у Redis реалізовано через механізми знімків (snapshotting) та журналювання (append-only file, AOF). Це дозволяє зберігати дані на диску та відновлювати їх після перезапуску або збою. Розробники можуть налаштувати частоту збереження даних та балансувати між продуктивністю та безпекою даних.

Переваги Redis:

– висока продуктивність, зберігання даних в оперативній пам'яті забезпечує дуже швидкий доступ з низькою затримкою;

– різноманітність структур даних, підтримка різних типів даних (рядки, хеші, списки, множини тощо) дозволяє ефективно зберігати та обробляти дані;

– збереження даних (persistence), можливість зберігати дані на диску та відновлювати їх після перезапуску або збою;

– реплікація та кластеризація, підтримка майстер-підлеглий реплікації та кластеризації для масштабування та високої доступності;

– підтримка транзакцій, можливість виконувати послідовність команд як єдину атомарну операцію;

– Lua-скрипти, підтримка виконання скриптів на мові Lua для реалізації складної логіки на стороні сервера;

– публікація-підписка (Pub/Sub), можливість реалізації систем обміну повідомленнями та сповіщень у реальному часі;

– гнучке налаштування, можливість налаштування різних параметрів, таких як політики виселення даних (LRU, LFU тощо), частота збереження даних, розмір пам'яті та інші;

– широка підтримка клієнтів, наявність клієнтських бібліотек для багатьох мов програмування, включаючи Java, що спрощує інтеграцію;

– активна спільнота та документація, велика спільнота розробників, детальна документація та численні ресурси для навчання;

– безпека, підтримка аутентифікації через паролі та TLS-шифрування для захисту даних при передачі;

Недоліки Redis:

– обмеження пам'яті, оскільки дані зберігаються в оперативній пам'яті, великий обсяг даних може вимагати значних ресурсів пам'яті;

– вартість масштабування, горизонтальне масштабування може бути дорогим через вимоги до пам'яті на кожному вузлі;

– складність налаштування кластерів, налаштування та управління кластером Redis може бути складним та вимагати додаткових знань;

– відсутність підтримки SQL, Redis не підтримує SQL-запити, що може ускладнити міграцію з реляційних баз даних;

– потенційні проблеми з персистентністю, у випадку неправильної конфігурації персистентності можлива втрата даних;

– обмежена транзакційність, транзакції в Redis не підтримують повну ізольованість (відсутність підтримки ACID повністю);

– однопоточна архітектура, основний процес Redis є однопоточним,

що може стати вузьким місцем при виконанні тривалих операцій;

– відсутність типів даних, всі дані зберігаються як байтові масиви, що може вимагати додаткової серіалізації/десеріалізації на стороні клієнта;

– відсутність вбудованої підтримки шардінгу в окремих версіях, хоча Redis Cluster вирішує цю проблему, він додає складність в управлінні.

Redis є потужним інструментом для кешування, який поєднує в собі швидкодію та розширені можливості зберігання даних. Завдяки підтримці різних структур даних та функцій, Redis може бути використаний не лише для кешування, але й для інших задач, таких як керування сесіями, обробка черг, статистика в реальному часі та інше.

У контексті веб-застосунку "Телеграм-бот ІТКІ" використання Redis для кешування є найбільш оптимальним рішенням з кількох причин. По-перше, Redis забезпечує високу швидкодію, що критично важливо для застосунків, які повинні швидко реагувати на запити користувачів.

По-друге, Redis підтримує різноманітні структури даних, що дозволяє зберігати складні об'єкти та оптимізувати доступ до них. У випадку "Телеграм-бота ІТКІ" це може бути корисним для зберігання сесій користувачів, налаштувань, історії взаємодій та інших даних, необхідних для персоналізації та ефективної роботи бота.

По-третє, Redis надає можливості реплікації та кластеризації, що забезпечує масштабованість та високу доступність системи. Це важливо для застосунків з великим навантаженням або тих, які повинні працювати без перерв навіть при збоях окремих вузлів.

Крім того, Redis легко інтегрується зі Spring Boot та підтримує стандартні механізми кешування, що спрощує розробку та підтримку застосунку. Наявність активної спільноти та великої кількості ресурсів робить процес навчання та вирішення проблем більш зручним.

Зважаючи на всі переваги Redis, можна зробити висновок, що він є найкращим рішенням для кешування даних у веб-застосунку "Телеграм-бот ІТКІ". Він забезпечує необхідну швидкодію, масштабованість та

функціональність, що дозволить значно покращити продуктивність та надійність веб-застосунку, а також підвищити задоволеність користувачів.

2.4.2 Особливості Minio для зберігання файлів

Minio – це високопродуктивна, розподілена об'єктна система зберігання даних, сумісна з Amazon S3 API. Вона розроблена для забезпечення масштабованого та надійного зберігання великих обсягів неструктурованих даних, що робить її особливо корисною для сучасних веб-застосунків, які потребують ефективного управління файлами та медіаконтентом. Minio відзначається своєю простотою в розгортанні та налаштуванні, що дозволяє швидко інтегрувати його в існуючу інфраструктуру без значних зусиль.

Альтернативами Minio є такі рішення для об'єктного зберігання, як Amazon S3, Google Cloud Storage, Microsoft Azure Blob Storage та відкриті системи на кшталт Ceph і OpenStack Swift. Ці сервіси також пропонують можливості для зберігання великих обсягів даних з високою доступністю та надійністю. Однак кожне з цих рішень має свої особливості, переваги та обмеження [18].

Переваги Minio перед альтернативами:

– відкритий вихідний код та безкоштовність, Minio є open-source рішенням, що дозволяє уникнути витрат на ліцензування та надає можливість модифікації коду під специфічні потреби;

– сумісність з S3 API, підтримка стандартного API забезпечує легку інтеграцію з існуючими застосунками та інструментами, що працюють з Amazon S3;

– простота розгортання, Minio можна швидко налаштувати на локальних серверах або в хмарному середовищі, що прискорює процес впровадження;

– легка масштабованість, система дозволяє додавати нові вузли для збільшення обсягу зберігання та підвищення продуктивності;

– висока продуктивність, оптимізована для роботи з великими файлами та забезпечує швидкий доступ до даних.

Недоліки Minio у порівнянні з альтернативами:

– обмежена функціональність у порівнянні з хмарними сервісами, відсутність додаткових сервісів, таких як автоматичне резервне копіювання, аналітика або глобальне розподілення даних, які пропонують великі хмарні провайдери;

– відповідальність за інфраструктуру, підтримка та адміністрування серверів Minio лягає на команду розробників або IT-відділ, що може вимагати додаткових ресурсів;

– менша географічна розподіленість, на відміну від глобальних хмарних сервісів, дані зберігаються в обмеженому наборі дата-центрів, що може впливати на швидкодію для користувачів з різних регіонів.

У контексті веб-застосунку "Телеграм-бот ІТКІ", Minio надає оптимальне рішення для асинхронного завантаження файлів, прикріплених до повідомлень для масових розсилок. Використання Minio дозволяє уникнути завантаженню файлу підчас виконання масових розсилок, що відповідає патерну Claim-Check та підвищує ефективність обробки даних. Це забезпечує наступні переваги для застосунку:

– зниження навантаження на сервер, відокремлення зберігання файлів від основного застосунку дозволяє оптимізувати використання ресурсів та покращити загальну продуктивність системи;

– покращення безпеки, зберігання файлів в окремому сховищі з чіткими правилами доступу підвищує рівень захисту даних;

– гнучкість та масштабованість, Minio дозволяє легко збільшувати обсяг зберігання без значних змін в архітектурі застосунку;

– економічна ефективність, відсутність витрат на ліцензування та можливість використання на існуючій інфраструктурі знижують загальні витрати на впровадження.

Враховуючи специфічні потреби "Телеграм-бот ІТКІ", Minio є найбільш доцільним вибором для зберігання файлів. Його використання сприяє оптимізації процесів зберігання та обробки файлів, підвищує продуктивність та безпеку системи, а також забезпечує необхідну гнучкість для подальшого розвитку та масштабування веб-застосунку.

2.5 Висновки по розділу 2

У розділі 2 було представлено проектування оновленої архітектури веб-застосунку "Телеграм-бот ІТКІ" із застосуванням патернів Cache-Aside та Claim-Check, метою яких є підвищення ефективності системи та оптимізація використання її ресурсів. Проектування архітектури базувалося на результатах аналізу поточних проблем системи, що були визначені у розділі 1. Зокрема, для реалізації нового підходу було додано два спеціалізованих сервіси, Redis та Minio, які інтегруються через REST API для забезпечення швидкої взаємодії з веб-застосунком.

Redis, впроваджений для реалізації патерну Cache-Aside, слугує високошвидкісним сховищем часто запитуваних даних, що дозволяє суттєво знизити навантаження на основну базу даних. Використання Redis забезпечує ефективне управління кешем, де веб-застосунок вирішує, коли звертатися до кешу, а коли – безпосередньо до бази даних, зменшуючи затримки у відповідях на запити. Окрім цього, Redis дозволяє зберігати тимчасові дані з функцією автоматичного видалення, що важливо для зменшення обсягу зайвих даних у системі.

Другий впроваджений сервіс, Minio, інтегровано для зберігання файлів-вкладень, що надсилаються адміністраторами в повідомленнях. Minio як об’єктне сховище реалізує патерн Claim-Check, де великі файли зберігаються поза основною базою даних, а додатку передаються лише посилання або ідентифікатори файлів. Такий підхід дозволяє зменшити обсяг переданих даних, підвищити ефективність обробки масових розсилок і оптимізувати використання мережевих ресурсів. Завдяки Claim-Check патерну, система

може обробляти більші обсяги даних, не перевантажуючи основний сервіс та забезпечуючи швидкий доступ до інформації.

Проведений аналіз альтернативних патернів Cache-Aside та Claim-Check показав, що обрані технології та підходи найбільш ефективні для вирішення задач проекту. Порівняння з іншими патернами для кешування і зберігання даних дозволило переконатися, що Redis та Minio відповідають усім вимогам для забезпечення швидкодії, надійності та масштабованості системи.

Таким чином, результати проектування архітектури з використанням Redis та Minio підтверджують доцільність впровадження обраних патернів. Оновлена архітектура забезпечує гнучкість, масштабованість і продуктивність, підвищує надійність зберігання та доступу до даних і сприяє подальшому розвитку та вдосконаленню веб-застосунку. Вибрані рішення створюють передумови для забезпечення швидкого доступу користувачів до частих запитів і оптимальної обробки великих файлів, що покращує загальний користувацький досвід і підвищує рівень функціональності системи.

3 АНАЛІЗ ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

3.1 Опис програмної реалізації

Реалізація веб-застосунку "Телеграм-бот ІТКІ" базується на використанні контейнеризації через Docker, що забезпечує модульність і гнучкість у розгортанні та управлінні компонентами системи. Основна структура застосунку описана у файлі конфігурації docker-compose.yml, який задає необхідні сервіси, змінні середовища та послідовність запуску контейнерів, що дозволяє ефективно керувати інфраструктурою застосунку.

Для впровадження нової архітектури було додано два автономних сервіси – Redis та Minio, які відповідають за оптимізацію управління даними та файлами. У конфігурації docker-compose для реалізації живого середовища (Додаток А docker-compose-live-reload.yml) ці сервіси описані як незалежні, щоб уникнути залежності від інших компонентів і забезпечити їхню автономну роботу. Це дозволяє запускати Redis та Minio паралельно з основними компонентами застосунку без прив’язки до конкретних частин системи, підвищуючи надійність і масштабованість загальної архітектури.

Основний API застосунку було модифіковано шляхом додавання двох нових залежностей – Redis та Minio. Такий підхід забезпечує інтеграцію патернів Cache-Aside та Claim-Check, що відповідають за оптимізацію процесів кешування та управління файлами. Redis використовується для кешування тимчасових даних, скорочуючи кількість запитів до основної бази даних і покращуючи швидкодію застосунку, особливо під час процесу аутентифікації. Minio ж служить сховищем для управління файлами відповідно до патерну Claim-Check, дозволяючи зменшити навантаження на базу даних за рахунок зберігання великих об’єктів поза її межами.

Оновлений порядок початку роботи сервісів можна описати таким чином:

1) Postgres DB instance (postgres), Redis server (redis), Minio server (minio);

2) Java REST API (api);

3) Node-RED telegram bot (node-red), Angular frontend (frontend).

3.2 Аналіз механізму кешування даних

3.2.1 Інтеграція Redis у застосунок

Для інтеграції Redis-серверу до проекту, було створено окремий сервіс в docker-compose-live-reload.yml файлі. Сервіс має власну перевірку стану Redis-серверу. Ця перевірка виконує команду «redis-cli ping». Якщо дана команда повертає «PONG» [19] протягом 5 секунд, тоді сервіс вважається здоровим і починають запускатися сервіси які потребують Redis-сервер (Java REST API). Якщо на протязі 5 секунд команда не повертає «PONG» 5 разів підряд, тоді сервіс вважається нездоровим і сервіси, що залежать від Redis-серверу не будуть запускатися. Час між перевірками Redis-серверу налаштовано на 10 секунд.

В середовищі Docker було додано кілька нових змінних (Додаток А .env_sample) для конфігурування інтеграції Java REST API з Redis-сервером:

– REDIS_HOSTNAME – домене ім’я серверу в віртуальній мережі Docker;

– REDIS_PORT – порт підключення, згідно дефолтного налаштування серверу;

– REDIS_FEATURE – змінна, що приймає логічне значення true або false, використовується в умовах на Java REST API, якщо задано false, тоді новий алгоритм кешування не буде використовуватися.

Для надання можливості з’єднання Java REST API з Redis-сервером було додано дві додаткових залежності до maven проекту, а саме «org.springframework.boot:spring-boot-starter-data-redis» та «redis.clients:jedis».

В самому проекті було додано клас конфігурації RedisConfig для створення RedisTemplate для кожного типу операцій з сутностями в Redis. Загалом, було створено два RedisTemplate: userByLoginCache – для оптимізації login ендпоінту та tokenBlackListCache – для реалізації додаткового захисту

токенів аутентифікації.

Для запису сутностей до Redis-серверу було змінено класи-моделі сутностей в БД. А саме надано дозвіл на серилізацію (перетворення у потік байтів) об'єктів цих класів. Це було зроблено за рахунок реалізації інтерфейсу Serializable.

Об'єкт userByLoginCache було параметризовано такими класами як String та User, що значить ключем доступу до інформації буде рядок, в даному випадку логін користувача. А значенням (або самими даними) буде сутність користувача з БД. Таким чином, API може зберігати та отримувати сутність користувача за його логіном. Забезпечення цих операцій є необхідним для реалізації алгоритму патерну Cache-Aside.

3.2.2 Аналіз алгоритму роботи з кешем

Згідно з патерном Cache-Aside, Java REST API повинно отримувати дані про користувача в кілька етапів:

– спроба отримати дані з кешу;

– якщо в кешу наявні дані, тоді передати дані з кешу;

– якщо в кешу даних немає, тоді зробити запит до БД і повернути дані з БД;

– невідстаючі дані записати до кешу.

Враховуючи, що ця оптимізація напрямлена на покращення продуктивності /login ендпоінту, який використовується для отримання аутентифікаційного токену, за час зберігання сутності у кешу можна використати час дії самого аутентифікаційного токену. Така практика, особливо корисна буде для сервісу Node-RED, який використовує /login ендпоінт для отримання аутентифікаційного токену для взаємодії з Java REST API.

Тому час зберігання сутності користувача в кеші було встановлено як 5JWT_TOKEN_TTL, де JWT_TOKEN_TTL – це змінна середовища, що відповідає за час дії аутентифікаційного токену. В проекті ця змінна дорівнює

одній годині, тобто час зберігання сутності користувача буде 5 годин.

За рахунок цього, Node-RED буде робити 5 запитів до кешу з 6, у випадку, коли жоден клієнт не буде аутентифікуватися у додатку. Якщо будуть додаткові запити від інших клієнтів, тоді Node-RED потенційно може брати інформацію лише з кешу, що значно знижує кількість запитів до серверу БД.

За цих умов, алгоритм реалізації патерну Cache-Aside, буде виглядати наступним чином:

– якщо змінна середовища REDIS_FEATURE дорівнює false, тоді дані будуть отримуватися і передаватися одразу з БД;

– якщо змінна середовища REDIS_FEATURE дорівнює true, тоді робиться спроба отримати значення з кешу;

– якщо дані в кеші наявні, тоді сутність користувача віддається одразу з кешу;

– якщо даних в кеші не має, тоді запит на отримання даних виконується до БД;

– відсутні дані в кеші записуються до кешу з часом існування 5 годин.

Даний алгоритм зображено на Рис. 3.1.

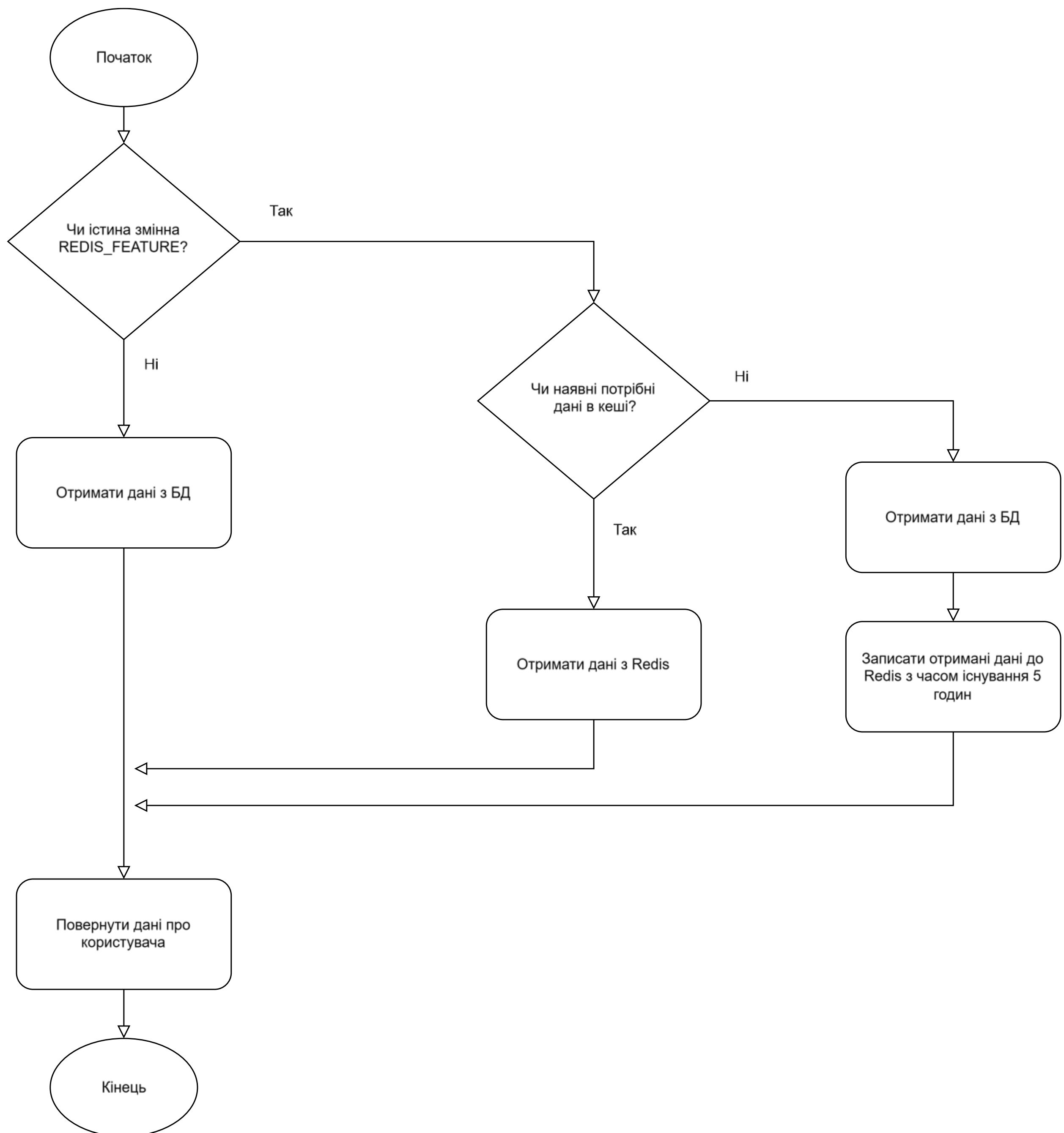

Рисунок 3.1 – Алгоритм-реалізація патерну Cache-Aside

3.3 Аналіз механізму зберігання та обробки файлів

3.3.1 Інтеграція Minio у застосунок

Для інтеграції Minio до проекту було створено окремий Dockerfile кастомного образу Minio.

Цей Dockerfile складається з двох стадій:

– перша стадія – встановлення Minio Client на Alpine;

– друга стадія – встановлення Minio та його підготовка.

На першій стадії використовується образ alpine:3.12. На нього встановлюється Minio Client для виконання потрібних команд.

На другій стадії використовується офіційний образ Minio. На цій стадії копіюється Minio Client з попередньої стадії, копіюється окремий скрипт з командами (create-bucket.sh) і виконується при запуску контейнеру.

Сам create-bucket.sh виконує такі дії як:

– запускає встановлення Minio з портом 9001 для консолі та текою data для зберігання файлів;

– очікує поки Minio запуститься;

– конфігурує Minio Client згідно з налаштуваннями Minio;

– за допомогою Minio Client, створює новий бакет з назвою my-data;

– підтримує контейнер запущеним за допомогою команди tail.

В файлі docker-compose-live-reload.yml створюється окремий сервіс minio на основі створеного Dockerfile. В якості платформи для підняття контейнеру використовується linux/x86_64. З контейнеру надається доступ до портів 9000 (для API взаємодії зі сховищем) та 9001 (консольний порт для перевірки вмісту сховища та адміністрування Minio з зовні). На рівні сервісу в docker-compose-live-reload.yml також конфігуруються ключі доступу до Minio, а саме MINIO_ACCESS_KEY та MINIO_SECRET_KEY.

Перевірка роботи Minio відбувається за рахунок команди «curl -f http://localhost:9000/minio/health/live». Ця команда виконує запит по посиланню «http://localhost:9000/minio/health/live» і якщо відповіддю буде не статус код 200, тоді curl поверне не нульовий код відповіді і спроба перевірити стан Minio буде провальною. Інтервал між перевірками встановлено в 30 секунд, максимальний час відповіді команди встановлено в 10 секунд, а кількість невдалих спроб після яких сервіс буде вважатися за несправний – 5.

Для Minio було створено кілька нових змінних середовища:

– MINIO_URL – посилання на кореневий ендпоінт Minio

(використовується тільки на Java REST API);

– MINIO_ACCESS_KEY – публічний ключ доступу до Minio;

– MINIO_SECRET_KEY – секретний ключ доступу до Minio.

Саме змінні середовища використовуються для конфігурування ключів доступу на самому сервісі.

До Java REST API було додано бібліотеку «io.minio:minio» для організації підключення до API Minio. Для взаємодії з Minio створено клас конфігурації MinioConfig що створює minioClient, який виконує CRUD операції з об’єктами в Minio.

Для паралельного завантаження файлів до Minio було створено UploadToMinioController, який забезпечує ендпоінт POST /minio/upload для завантаження файлів до Minio. При відповіді, цей ендпоінт повертає назву файлу всередині бакету. Назва файлів в бакеті будується по шаблону "[d] s", де d – це поточна дата у вигляді timestamp, а s – оригінальна назва файлу завантаженого на бакет. Такий підхід вирішує проблему з унікальністю назв файлів всередині бакету.

Для здійснення масових розсилок повідомлень з використанням назви файлу було створено окремий контролер (TelegramAsyncController). Цей контролер має аналогічний набір ендпоінтів які використовувалися для масових розсилок у синхронному форматі. Для маркування нових ендпоінтів було використано префікс «v2» в шляху (наприклад ендпоінт надсилання повідомлення з фотографією: «/tg/broadcast/photo» та «/tg/v2/broadcast/photo»).

При надсиланні повідомлення через нові асинхронні ендпоінти, з фронтенду передається назва файлу і припис до нього. Java REST API приймає запит, створює Virtual Thread для асинхронного виконання задачі і повертає користувачу статус код «OK». В віртуальному потоці REST API виконує завантаження файлу з Minio по назві файлу, і циклічне надсилання повідомлень користувачам. Такий підхід робить надсилання повідомлень більш надійним, гнучким для подальшого розвитку архітектури та більш швидким і зручним для адміністраторів, що взаємодіють з Angular frontend.

3.3.2 Аналіз обробки файлів за патерном Claim-Check

Обробка файлів за патерном Claim-Check передбачає роботу з файлами наступним чином:

– відправник надсилає файл до системи;

– система завантажує файл до зовнішнього сховища і генерує посилання на завантажений файл, саме це посилання і є Claim-Check;

– Claim-Check надсилається на обробку;

– отримувач Claim-Check виконує завантаження;

– отримувач здійснює обробку надісланого файлу.

Враховуючи область, де буде використовуватися даний патерн, алгоритм обробки запиті на надсилання масових розсилок з адміністративної платформи (Angular frontend) буде виглядати наступним чином:

– адміністратор прикріпляє файл до форми надсилання повідомлень;

– замість збереження шляху файлу для подальшого його завантаження на сервіс, адміністративна платформа одразу починає завантаження файлу не чекаючи надсилання його адміністратором;

– після завантаження файлу через ендпоінт /minio/upload фронтенд отримує назву файлу з зовнішнього сховища;

– адміністратор надсилає повідомлення для масової розсилки;

– Java REST API приймає запит;

– створює віртуальний потік для виконання задачі;

– надсилає статус код «ОК»;

– паралельно у віртуальному потоці відбувається завантаження файлу та циклічне надсилання повідомлень користувачам.

Даний алгоритм зображено на Рис 3.2.

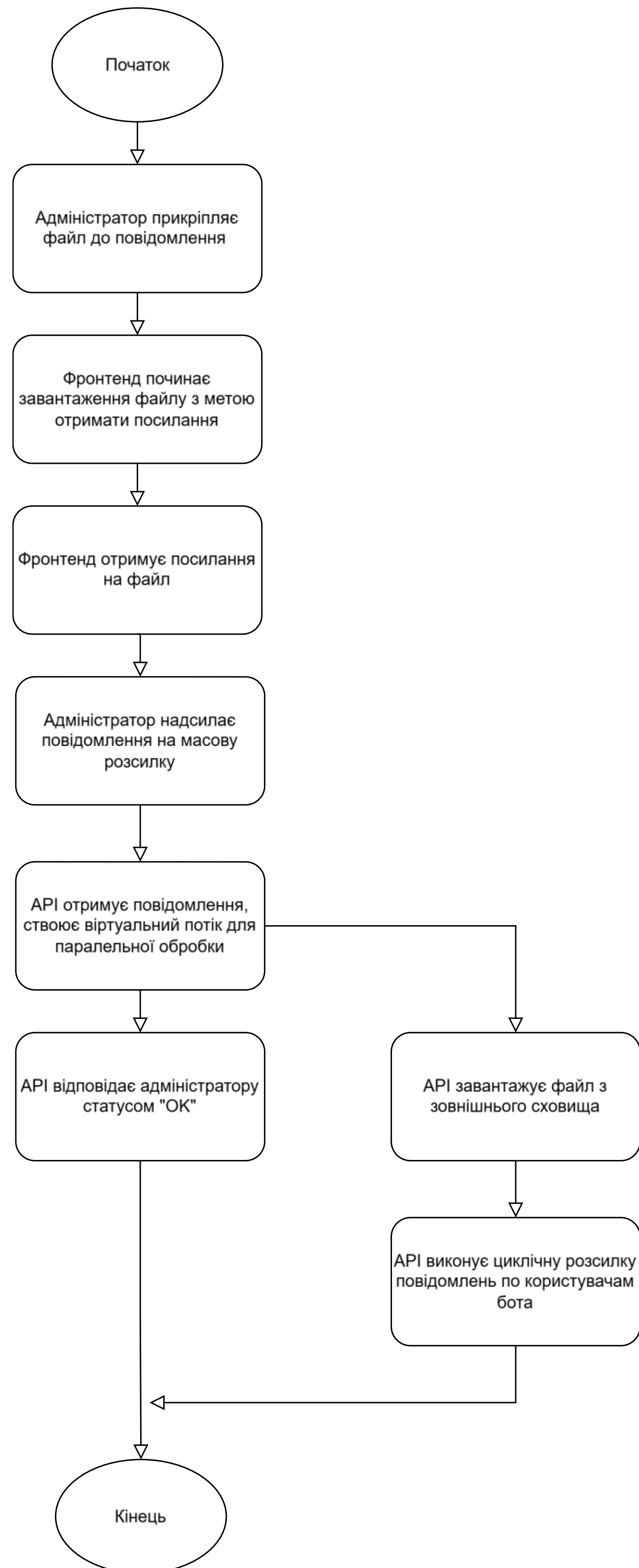

Рисунок 3.2 – Алгоритм-реалізація патерну Claim-Check

3.4 Аналіз додаткового захисту аутентифікаційних токенів

Однією з критичних проблем безпеки веб-застосунку "Телеграм-бот ІТКІ" є можливість компрометації аутентифікаційних токенів, зокрема authToken та refreshToken. У випадку, якщо зловмисник отримує доступ до цих токенів, він може виконувати всі дії від імені адміністратора, що створює серйозну загрозу для цілісності та конфіденційності системи. Ця вразливість пов'язана з природою JWT-токенів (JSON Web Tokens), які містять основну інформацію про користувача та підпис сервера. Оскільки JWT-токени існують незалежно від сеансу та контролюються лише строком дії, вони залишаються дійсними навіть після завершення сеансу адміністратора. Детальний аналіз цієї проблеми було проведено у підрозділі «1.3 Аналіз проблематики безпеки веб-застосунку».

Для вирішення цієї проблеми пропонується впровадити механізм відкликання токенів. Суть цього підходу полягає в зберіганні відкликаних аутентифікаційних токенів після завершення сеансу адміністратора в окремому безпечному сховищі. При кожній спробі використання аутентифікаційного токена або токена оновлення сеансу система повинна виконувати додаткову перевірку на наявність цього токена в сховищі відкликаних токенів. Якщо токен виявлено в цьому сховищі, він вважається недійсним, і доступ користувачу не надається.

Зважаючи на обмежений строк дії JWT-токенів, зберігання відкликаних токенів є ефективним лише протягом періоду їх дійсності. Тому важливо використовувати сховище, яке підтримує автоматичне видалення даних після закінчення встановленого часу життя (TTL). У цьому контексті Redis є оптимальним вибором завдяки його високій продуктивності, підтримці TTL та можливості швидкого доступу до даних. Redis дозволяє ефективно зберігати відкликані токени та забезпечує миттєвий доступ до них під час перевірки автентичності.

Оновлений алгоритм аутентифікації з використанням механізму відкликання токенів буде виглядати наступним чином (Рис. 3.3).

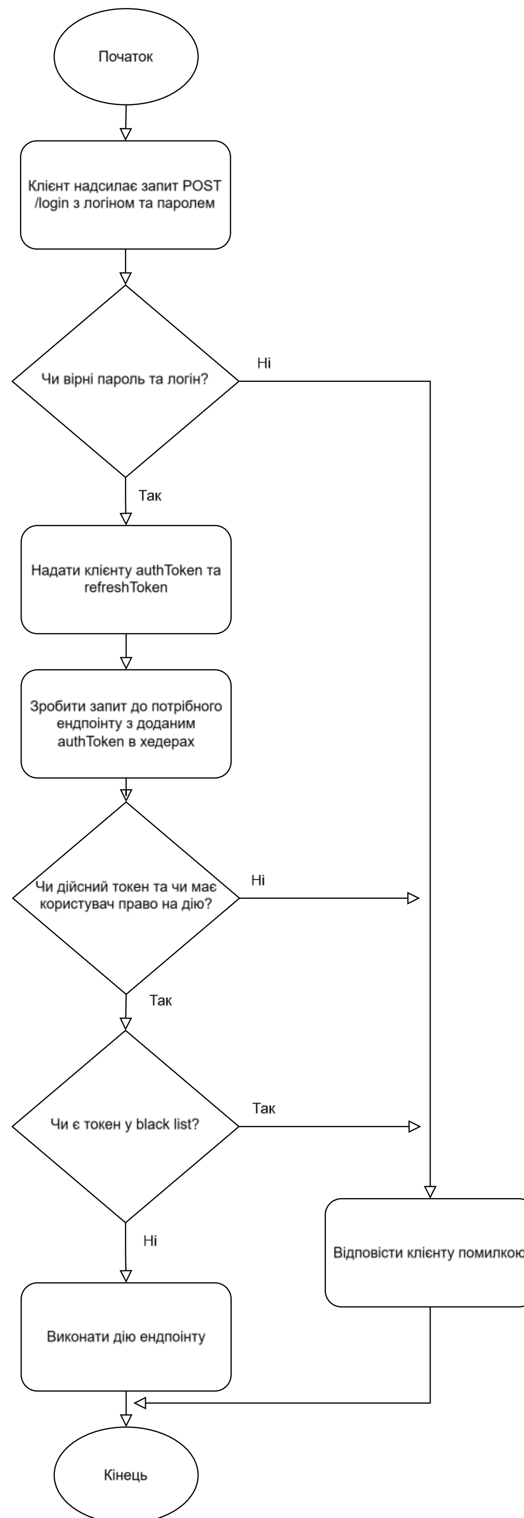

Рисунок 3.3 – Алгоритм аутентифікації на API з зовнішнім сховищем

На Рис 3.3 можна побачити, що перевірка зовнішнього сховища

відбувається після перевірки токену на валідність. Такий підхід дозволить зменшити кількість звернень до Redis.

Для забезпечення збереження токенів в Redis було створено tokenBlackListCache. Redis зберігає дані способом схожим з HashMap. Тобто, зберігає дані парами (ключ - значення). У випадку створення чорного списку токенів, можна скористатися принципом роботи HashSet – використати токен аутентифікації в якості ключу, а в якості значення використовувати заглушку, що не буде використовувати пам'ять сховища (null). В такому випадку, для перевірки вмісту сховища, можна використати операцію hasKey, що поверне одразу значення істини у випадку наявності ключа в сховищі.

В якості TTL для збереження токену в сховищі, було використано час життя кожного токену (для токену аутентифікації – одна година, для токену оновлення сеансу – один рік).

3.5 Перевірка результатів оптимізації додатку

3.5.1 Перевірка результатів отримання даних користувача з кешу

Для перевірки швидкості отримання даних про користувача, було додано додаткове логування, що показує час, що було витрачено на отримання даних (Рис 3.4).

```
c.itki.api.service.impl.UserServiceImpl  : findByLoginWithCache took 2 ms
c.itki.api.service.impl.UserServiceImpl  : findByLoginWithCache took 2 ms
```

Рисунок 3.4 – Приклад логів отримання даних про користувача

В якості пристрою для тестування було використано Apple MacBook Pro 14” 2021.

Після 240 запитів було отримано наступні дані (Таблиця 3.1).

Таблиця 3.1 – Статистика отримання інформації про користувача

Параметр	З Cache-Aside	Без Cache-Aside
Мінімальний час, мс	0	1
Максимальний час, мс	3	42
Середній час, мс	0.7	1.96

Зі значень таблиці 3.1 можна підвести підсумки, що використання Cache-Aside зменшило пікові затримки, а середній час отримання інформації зменшився на ~ 64.3%.

3.5.2 Перевірка результатів відповідей /login ендпоінту з кешем

Для тестування /login ендпоінту було використано JMeter з наступною конфігурацією:

– Number of Threads – 1;
– Ramp-up period – 0.1;
– Loop Count – 10000.

Таблиця 3.2 – Статистика відповіді на запит до /login ендпоінту

Параметр	З Cache-Aside	Без Cache-Aside
Кількість запитів, шт	10000	10000
Мінімальний час, мс	75	76
Максимальний час, мс	163	243
Середній час, мс	77	79

З цієї статистики можна побачити, що приріст в швидкодії є навіть на рівні Tomcat. Пікова затримка зменшилася на ~ 33%, а середня на ~ 3%.

Також якщо подивитися на графік затримки відповіді відносно часу запиту, можна побачити тенденцію до покращення продуктивності, як на пікових так і на середніх значеннях.

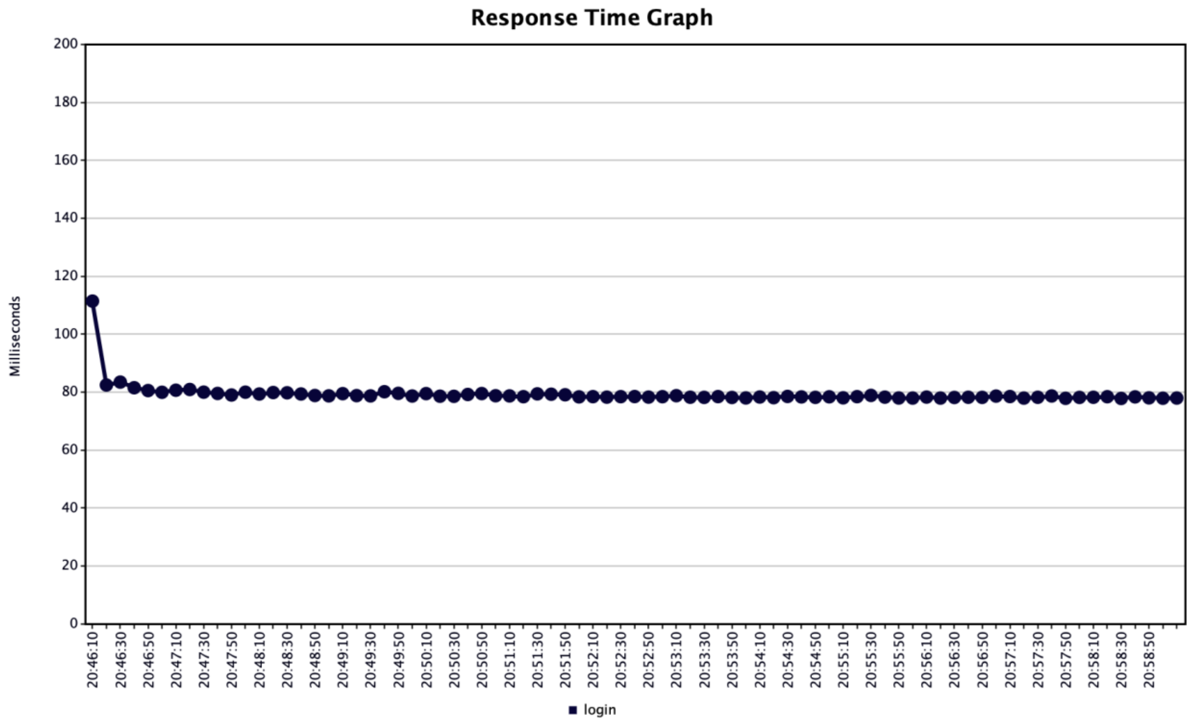

Рисунок 3.5 – Графік затримки відповіді для API без Cache-Aside

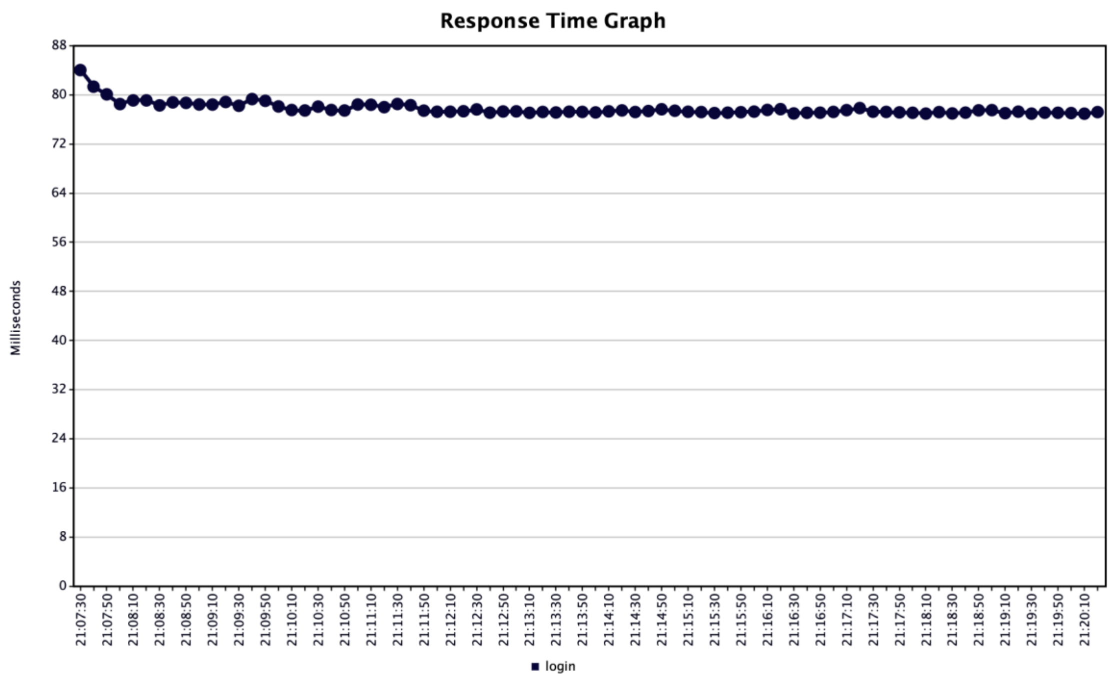

Рисунок 3.6 – Графік затримки відповіді для API з Cache-Aside

Окрім цього, тенденція при використанні Cache-Aside буде ще більше помітною при збільшенні об’ємів сховища PostgreSQL, бо на відміну від Redis, сховище PostgreSQL не буде звільнятися автоматично, а при налаштуванні

видалення за рахунок розпорядків та stored procedure, продуктивність в час роботи цих процедур буде зменшуватися.

3.5.3 Перевірка результатів асинхронного запиту масової розсилки

На відміну від попередніх запитів, запити масових розсилок напряму пов’язані з Telegram API, тому тестування було проведено через Postman з вибіркою в 2 запити. При тестуванні до додатку було підключено 2 користувачі телеграм боту. Для тестування було використано ендпоінти: /tg/broadcast/photo – з синхронним надсиланням файлів до Telegram API та /tg/v2/broadcast/photo з асинхронним надсиланням файлів і використанням патерну Claim-Check. Обидва ендпоінти використовуються для масової розсилки повідомлення з однією фотографією та приписом до неї.

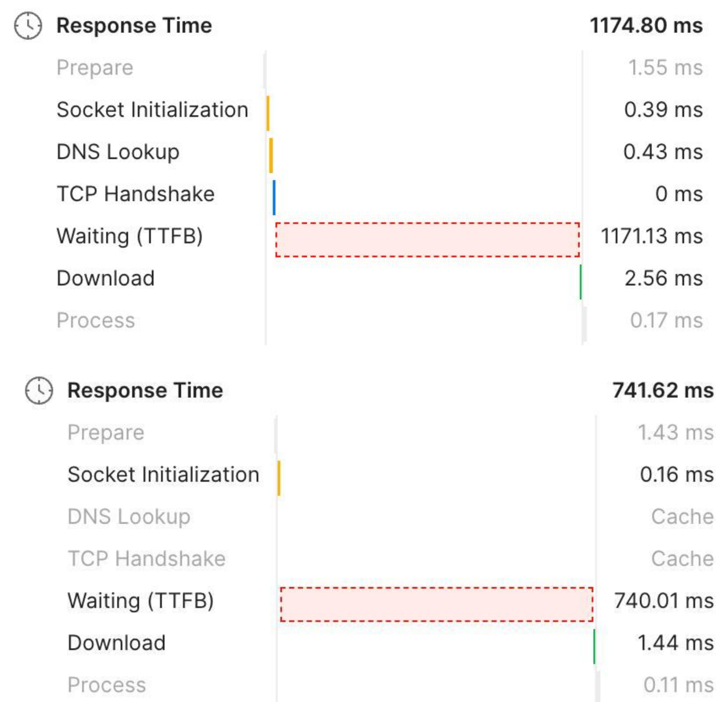

Рисунок 3.7 – Час відповіді синхронної масової розсилки

З двох запитів до ендпоінту /tg/broadcast/photo (Рис 3.7) можна побачити, що середній час відповіді 958.21 мс.

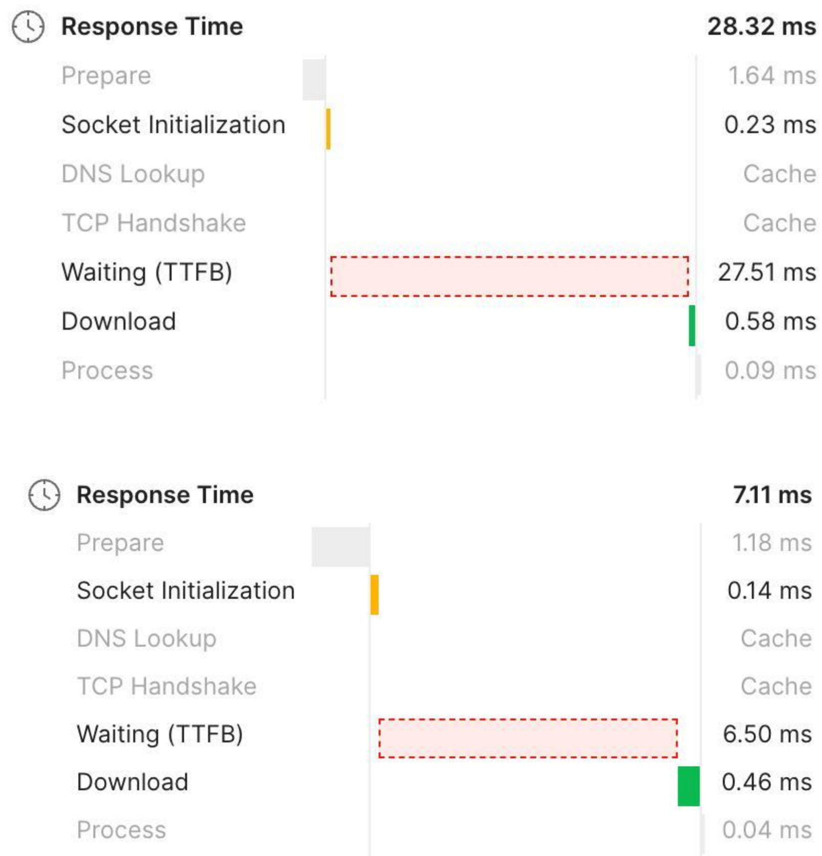

Рисунок 3.8 – Час відповіді асинхронної масової розсилки

Як можна побачити на Рис. 3.8 середній час відповіді при запиті на /tg/v2/broadcast/photo складає 17.7 мс, що менше синхронної розсилки приблизно в 54 рази.

Для масової розсилки було використано наступний файл (Рис.3.9).

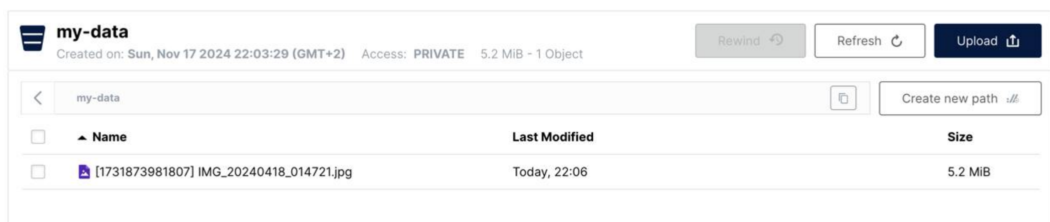

Рисунок 3.9 – Файл, що було використано для масової розсилки

3.6 Висновки по розділу 3

У розділі 3 було практично реалізовано оновлену архітектуру веб-застосунку "Телеграм-бот ІТКІ" з впровадженням патернів Cache-Aside та

Claim-Check, розроблених у попередньому розділі. Інтеграція сервісів Redis та Minio дозволила оцінити ефективність запропонованих рішень та їх вплив на продуктивність системи. Результати тестування продемонстрували суттєве покращення швидкодії: використання Cache-Aside зменшило пікові затримки, а середній час отримання інформації скоротився приблизно на 64.3%. Навіть на рівні сервера Tomcat було відзначено приріст швидкодії—пікова затримка зменшилася на 33%, а середня—на 3%. Ці показники підтверджують ефективність кешування даних за допомогою Redis та доцільність застосування патерну Cache-Aside.

У контексті масових розсилок повідомлень було впроваджено патерн Claim-Check, що продемонстрував значні переваги. Порівняльне тестування ендпоінтів /tg/broadcast/photo (синхронна розсилка) та /tg/v2/broadcast/photo (асинхронна розсилка з використанням Claim-Check) показало, що середній час відповіді зменшився з 958.21 мс до 17.7 мс—приблизно в 54 рази. Це свідчить про ефективність обробки файлів та оптимізацію взаємодії з Telegram API завдяки використанню Minio як об'єктного сховища.

Отримані результати підтверджують, що впровадження патернів Cache-Aside та Claim-Check позитивно впливає на продуктивність і масштабованість веб-застосунку. Зниження затримок і підвищення швидкодії не лише покращують користувацький досвід, але й підвищують надійність та ефективність системи. Впроваджені рішення забезпечують гнучкість архітектури, створюючи передумови для подальшого розвитку та вдосконалення застосунку в умовах зростаючих вимог до якості програмного забезпечення.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи було досліджено та успішно впроваджено патерни Cache-Aside та Claim-Check у веб-застосунку "Телеграм-бот ІТКІ". Основною метою роботи було підвищення продуктивності, масштабованості та безпеки системи шляхом оптимізації її архітектури та усунення виявлених недоліків.

Аналіз існуючої системи виявив, що монолітна архітектура з синхронною обробкою запитів, неефективним управлінням даними та недосконалими механізмами аутентифікації призводила до затримок у роботі та потенційних вразливостей у безпеці. Це негативно впливало на користувацький досвід та обмежувало можливості подальшого розвитку застосунку.

Впровадження патерну Cache-Aside з використанням Redis дозволило оптимізувати доступ до даних і прискорити процеси аутентифікації. Тестування показало, що використання кешування суттєво зменшило затримки при отриманні інформації користувача. Зокрема, середній час отримання даних скоротився приблизно на 64.3%, що свідчить про ефективність обраного підходу та покращення швидкодії системи.

Патерн Claim-Check, реалізований через інтеграцію Minio як об'єктного сховища, забезпечив ефективне зберігання та обробку файлів, особливо при масових розсилках. Тестування асинхронної розсилки повідомлень показало, що середній час відповіді зменшився з 958.21 мс до 17.7 мс, тобто приблизно в 54 рази. Це підтверджує, що впровадження патерну Claim-Check значно підвищило продуктивність системи та дозволило оптимізувати використання ресурсів.

Додатково було розроблено механізм захисту аутентифікаційних токенів, який підвищив рівень безпеки системи та запобіг несанкціонованому доступу після виходу користувача. Це сприяло посиленню захисту даних користувачів і підвищило довіру до застосунку.

Отримані результати підтверджують, що Cache-Aside та Claim-Check

патернів значно покращило продуктивність, масштабованість та безпеку веб-застосунку "Телеграм-бот ІТКІ". Покращення швидкодії та ефективності обробки даних не лише підвищило якість користувацького досвіду, але й створило гнучку та надійну основу для подальшого розвитку системи.

ПЕРЕЛІК ПОСИЛАНЬ

1. Caching Strategy: Read-Through Pattern. URL: https://www.enjoyalgorithms.com/blog/read-through-caching-strategy (дата звернення: 19.11.2024).

2. Journal of Information Systems & Operations Management, Vol. 18 No. 1, May 2024. Romanian-American University Publishing House. URL: https://web.rau.ro/websites/jisom/Vol.18%20No.1%20-%202024/JISOM%2018.1.pdf (дата звернення: 19.11.2024).

3. Claim-Check pattern – Azure Architecture Center. URL: https://learn.microsoft.com/en-us/azure/architecture/patterns/claim-check (дата звернення: 19.11.2024).

4. Caching Strategies and How to Choose the Right One – CodeAhoy. URL: https://codeahoy.com/2017/08/11/caching-strategies-and-how-to-choose-the-right-one/ (дата звернення: 19.11.2024).

5. Caching patterns – Database Caching Strategies Using Redis. URL: https://docs.aws.amazon.com/whitepapers/latest/database-caching-strategies-using-redis/caching-patterns.html (дата звернення: 19.11.2024).

6. What Is Read-Through Cache? – ITU Online IT Training. URL: https://www.ituonline.com/tech-definitions/what-is-read-through-cache/ (дата звернення: 19.11.2024).

7. Write-Around Caching Strategy – EnjoyAlgorithms. URL: https://www.enjoyalgorithms.com/blog/write-around-caching-pattern (дата звернення: 19.11.2024).

8. Read-Through, Write-Through, Write-Behind, and Refresh-Ahead Caching – Oracle Coherence 3.3 User Guide. URL: https://docs.oracle.com/cd/E14447_01/coh.330/coh33ug/readthroughwritethrough.htm (дата звернення: 19.11.2024).

9. Wijeweera, B. Tips Before Implementing a Split-Aggregate Scenario with WSO2 EI. URL: https://buddhimawijeweera.wordpress.com/2022/05/30/tips-

before-implementing-a-split-aggregate-scenario-with-wso2-ei/ (дата звернення: 19.11.2024).

10. Content Enricher – Enterprise Integration Patterns. URL: https://www.enterpriseintegrationpatterns.com/patterns/messaging/DataEnricher.html (дата звернення: 19.11.2024).

11. Content Filter – Enterprise Integration Patterns. URL: https://www.enterpriseintegrationpatterns.com/patterns/messaging/ContentFilter.html (дата звернення: 19.11.2024).

12. Process Manager – Enterprise Integration Patterns. URL: https://www.enterpriseintegrationpatterns.com/patterns/messaging/ProcessManager.html (дата звернення: 19.11.2024).

13. Community Edition – Redis Documentation. URL: https://redis.io/docs/latest/get-started/ (дата звернення: 19.11.2024).

14. Caffeine Cache – Javarevisited. URL: https://medium.com/javarevisited/caffeine-cache-f106cee91925 (дата звернення: 19.11.2024).

15. Ehcache – Java's Most Widely-Used Cache. URL: https://www.ehcache.org/ (дата звернення: 19.11.2024).

16. Caching Data – Hazelcast Documentation. URL: https://docs.hazelcast.com/hazelcast/5.5/cache/overview (дата звернення: 19.11.2024).

17. Memcached – A Distributed Memory Object Caching System. URL: https://memcached.org/ (дата звернення: 19.11.2024).

18. MinIO – S3 Compatible Storage for AI. URL: https://min.io/ (дата звернення: 19.11.2024).

19. PING – Redis Documentation. URL: https://redis.io/docs/latest/commands/ping/ (дата звернення: 19.11.2024).

ДОДАТОК А. ОНОВЛЕННІ ФАЙЛИ ЗАСТОСУНКУ

.env_sample

```
# Node-RED
TZ=Europe/Kiev
API_HOST=host.docker.internal:8080
API_LOGIN=admin
API_PASSWORD=admin
# add your telegram token and username here
TELEGRAM_TOKEN=_
TELEGRAM_USERNAME=_
NMU_API=https://www.nmu.org.ua/ua/content/student_life/students/schedule


# DB properties
POSTGRES_USER=root
POSTGRES_PASSWORD=12345
POSTGRES_DB=itki_db

# REDIS
REDIS_HOSTNAME=redis
REDIS_PORT=6379
REDIS_FEATURE=true

# API
SPRING_LOCAL_PORT=8080
SPRING_DOCKER_PORT=8080
DEBUG_PORT=5005
JWT_TOKEN_TTL=3600000
REFRESH_TOKEN_TTL_IN_DAYS=365
MINIO_URL=http://minio:9000

# MINIO
MINIO_ACCESS_KEY=987654321
MINIO_SECRET_KEY=123456789
```

docker-compose-live-reload.yml

```
version: '3'
services:
  # DB
  postgres:
    platform: linux/x86_64
    image: 'postgres:15-alpine'
    container_name: postgres
    env_file: ./.env
```

```
    ports:
      - "5432:5432"
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U $POSTGRES_USER -d $POSTGRES_DB"]
      interval: 10s
      timeout: 5s
      retries: 5

  # CACHE
  redis:
    image: redis:latest
    container_name: redis
    hostname: redis
    ports:
      - "6379:6379"
    healthcheck:
      test: [ "CMD", "redis-cli", "ping" ]
      interval: 10s
      timeout: 5s
      retries: 5

  # FILE STORAGE
  minio:
    platform: linux/x86_64
    container_name: minio
    env_file: ./.env
    build:
      context: ./minio
      dockerfile: Dockerfile
    ports:
      - "9000:9000"
      - "9001:9001"
    environment:
      MINIO_ACCESS_KEY: $MINIO_ACCESS_KEY
      MINIO_SECRET_KEY: $MINIO_SECRET_KEY
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:9000/minio/health/live"]
      interval: 30s
      timeout: 10s
      retries: 5

  # API
  api:
    depends_on:
      postgres:
        condition: service_healthy
      redis:
        condition: service_healthy
      minio:
        condition: service_healthy
    image: itki-bot-api
    build:
```

```
    context: ./itki-bot-api
    dockerfile: Dockerfile.live
  env_file: ./.env
  container_name: api
  ports:
    - $SPRING_LOCAL_PORT:$SPRING_DOCKER_PORT
    - $DEBUG_PORT:$DEBUG_PORT
  environment:
    - spring.datasource.url=jdbc:postgresql://postgres:5432/$POSTGRES_DB
    - spring.datasource.username=$POSTGRES_USER
    - spring.datasource.password=$POSTGRES_PASSWORD
    - security.jwt.token.expire-length=$JWT_TOKEN_TTL
    - spring.jpa.hibernate.ddl-auto=validate
    - JAVA_TOOL_OPTIONS=-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=*:5005
  healthcheck:
    test: [ "CMD-SHELL", "curl -X GET localhost:8080/success" ]
    interval: 10s
    timeout: 5s
    retries: 5

# Node RED
node-red:
  depends_on:
    api:
      condition: service_healthy
  container_name: node-red
  image: node-red
  environment:
    - TZ=Europe/Kiev
  build: node-red/.
  ports:
    - "1880:1880"
  env_file:
    - .env
  volumes:
    - ./node-red/config:/root/.node-red:delegated
    - /root/.node-red/node_modules

# FRONT
frontend:
  container_name: frontend
  image: frontend
  build:
    context: ./frontend
    dockerfile: Dockerfile.live
  ports:
    - "4200:4200"
  volumes:
      - ./frontend:/app:delegated
  depends_on:
    api:
      condition: service_healthy
```

frontend/src/app/auth/auth-interceptor.ts

```
import {
  HttpErrorResponse,
  HttpEvent,
  HttpHandler,
  HttpInterceptor,
  HttpRequest,
  HttpStatusCode,
} from '@angular/common/http';
import { Injectable } from '@angular/core';
import {
  BehaviorSubject, Observable, throwError,
} from 'rxjs';
import {
  catchError, filter, retry, switchMap, take,
} from 'rxjs/operators';
import {AuthService} from "./auth.service";
import {TokenService} from "./token.service";
import {TokenDto} from "./token-dto";

const refreshPathname = '/refresh';
const loginPathname = '/login'

@Injectable()
export class AuthInterceptor implements HttpInterceptor {
  private isRefreshing = false;

  private refreshTokenSubject = new BehaviorSubject<string>('');

  constructor(
    private authService: AuthService,
    private tokenService: TokenService,
  ) {
  }

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    const token = this.tokenService.getAccessToken();

    if (token != null) {
      req = this.addToken(req, token);
    }

    return next.handle(req).pipe(
      catchError((error) => {
        const url = new URL(error.url).pathname;
```

```
      if (req.url.endsWith(loginPathname)) {
        return next.handle(req);
      }

      if (this.isForbiddenStatus(error)
        && url === refreshPathname && this.tokenService.getRefreshToken() !== '') {
        this.authService.logout();

        return throwError(error);
      }

      if (this.isForbiddenStatus(error) && this.tokenService.getRefreshToken() !== '') {
        return this.refreshToken(req, next);
      }
      return throwError(error);
    }),
    retry(3),
  );
}

private isForbiddenStatus(error: any) {
  return error instanceof HttpErrorResponse
    && error.status === HttpStatusCode.Forbidden;
}

private addToken(request: HttpRequest<any>, token: string) {
  return request.clone({
    setHeaders: {
      Authorization: `Bearer ${token}`,
    },
  });
}

private refreshToken(
  request: HttpRequest<any>,
  next: HttpHandler,
): Observable<HttpEvent<any>> {
  if (!this.isRefreshing) {
    this.isRefreshing = true;
    this.refreshTokenSubject.next('');

    return this.authService.refreshToken().pipe(
      switchMap((loginResponseDto: TokenDto) => {
        const newAccessToken = loginResponseDto.token;

        this.refreshTokenSubject.next(newAccessToken);
        this.isRefreshing = false;

        return next.handle(this.addToken(request, newAccessToken));
      }),
    );
  }
```

```
    return this.refreshTokenSubject.pipe(
      filter((token) => token !== ''),
      take(1),
      switchMap((token) => next.handle(this.addToken(request, token))),
    );
  }
}
```

frontend/src/app/auth/auth.service.ts

```
import { Injectable } from '@angular/core';
import { environment } from '../../environments/environment';
import { HttpClient, HttpHeaders } from "@angular/common/http";
import { TokenService } from "./token.service";
import { LoginRequestDto } from "./login-request-dto";
import { TokenDto } from "./token-dto";
import { finalize, Observable, tap } from "rxjs";
import { ROUTES } from "../../constants/routes.constants";
import { Router } from "@angular/router";


@Injectable({
  providedIn: 'root'
})
export class AuthService {
  private baseUrl = environment.apiBaseUrl;

  private readonly loginUrl = `${this.baseUrl}/login`;
  private readonly refreshUrl = `${this.baseUrl}/refresh`;
  private readonly logoutUrl = `${this.baseUrl}/logout`;
  private readonly httpOptions = {
    headers: new HttpHeaders({ 'Content-Type': 'application/json' }),
  };

  constructor(
    private httpClient: HttpClient,
    private tokenService: TokenService,
    private router: Router,

  ) {
  }

  authenticate(loginRequestDto: LoginRequestDto): Observable<TokenDto> {
    return this.httpClient.post<TokenDto>(
      this.loginUrl,
      loginRequestDto,
      this.httpOptions,
```

```
    );
  }

  refreshToken(): Observable<TokenDto> {
    return this.httpClient.post<TokenDto>(
      this.refreshUrl,
      this.tokenService.getRefreshToken(),
      this.httpOptions,
    ).pipe(tap((loginResponseDto: TokenDto) => {
      this.tokenService.saveTokens(
        loginResponseDto.token,
        loginResponseDto.refreshToken,
      );
    }));
  }

  logout(): void {
    const tokenDto: TokenDto = {
      token: this.tokenService.getAccessToken(),
      refreshToken: this.tokenService.getRefreshToken()
    };
    this.httpClient.post<boolean>(this.logoutUrl, tokenDto, this.httpOptions)
      .pipe(
        finalize(() => {
          this.tokenService.deleteTokens();
          this.router.navigate([ROUTES.login]);
        })
      ).subscribe();
  }
}
```

frontend/src/app/auth/token-dto.ts

```
export interface TokenDto {
  token: string;
  refreshToken: string;
}
```

frontend/src/app/dto/telegram-send-file-request-dto.ts

```
export class TelegramSendFileRequestDto {
  caption: string;
  filenames: Array<string>;
```

```
  constructor(
    caption: string,
    filenames: Array<string>,
  ) {
    this.caption = caption;
    this.filenames = filenames;
  }
}
```

frontend/src/app/modal/exit-modal/exit-modal.component.ts

```
import { Component } from '@angular/core';
import { NzModalService } from 'ng-zorro-antd/modal';
import {AuthService} from '../../auth/auth.service';

@Component({
  selector: 'app-exit-nzModalService',
  templateUrl: './exit-modal.component.html',
  styleUrls: ['./exit-modal.component.scss']
})
export class ExitModalComponent {
  constructor(
    private nzModalService: NzModalService,
    private authService: AuthService,
  ) {
  }

  showModal(): void {
    this.nzModalService.closeAll();
    this.nzModalService.confirm({
      nzTitle: 'Вихід з облікового запису',
      nzWidth: '40%',
      nzMaskClosable: false,
      nzOnCancel: () => this.handleCancel(),
      nzOnOk: () => this.handleOk(),
      nzContent: ExitModalComponent,
      nzClosable: false,
      nzOkText: 'Гаразд',
      nzCancelText: 'Скасувати',
    });
  }

  handleOk(): void {
    this.authService.logout();
  }

  handleCancel(): void {
    this.nzModalService.closeAll();
```

```
  }
}
```

frontend/src/app/page/login/login.component.ts

```
import { Component, OnInit } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';
import { AuthService } from '../../auth/auth.service';
import { LoginRequestDto } from '../../auth/login-request-dto';
import { TokenService } from '../../auth/token.service';
import { TokenDto } from '../../auth/token-dto';
import { NzMessageService } from 'ng-zorro-antd/message';
import { ROUTES } from '../../../constants/routes.constants';

@Component({
  selector: 'app-login',
  templateUrl: './login.component.html',
  styleUrls: ['./login.component.css']
})
export class LoginComponent implements OnInit {
  loginForm!: FormGroup;
  showPassword = false;

  constructor(
    private fb: FormBuilder,
    private authService: AuthService,
    private nzMessageService: NzMessageService,
    private tokenService: TokenService,
    ) { }

  ngOnInit(): void {
    this.loginForm = this.fb.group({
      username: ['', [Validators.required]],
      password: ['', [Validators.required]]
    });
  }

  get username() {
    return this.loginForm.get('username');
  }

  get password() {
    return this.loginForm.get('password');
  }

  toggleShowPassword() {
    this.showPassword = !this.showPassword;
  }
```

```
  getValidationStatus(controlName: string): string {
    const control = this.loginForm.get(controlName);
    if (control) {
      return control.touched && control.invalid ? 'error' : control.dirty ? 'success' : '';
    }
    return 'error';
  }


  onSubmit(): void {
    if (this.loginForm.valid) {
      const { username, password } = this.loginForm.value;
      this.authService.authenticate(new LoginRequestDto(username, password)).subscribe(
        (loginResponseDto: TokenDto) => {
          this.tokenService.saveTokens(loginResponseDto.token, loginResponseDto.refreshToken)
          window.location.href = ROUTES.menu
        },
        () => {
          this.nzMessageService.error('Невірний логін або пароль, спробуйте знов')
        },
      )
    } else {
      Object.values(this.loginForm.controls).forEach(control => {
        if (control.invalid) {
          control.markAsDirty();
          control.updateValueAndValidity({ onlySelf: true });
        }
      });
    }
  }
}
```

frontend/src/app/page/send-message/form/send-file/send-file.component.ts

```
import { Component, OnDestroy, OnInit } from '@angular/core';
import { UntypedFormBuilder, UntypedFormGroup, Validators } from '@angular/forms';
import { NzUploadFile } from 'ng-zorro-antd/upload';
import { NzMessageService } from 'ng-zorro-antd/message';
import { MessageService } from '../../../../service/message.service';
import { Subscription } from 'rxjs';

@Component({
  selector: 'app-send-file',
  templateUrl: './send-file.component.html',
  styleUrls: ['./send-file.component.scss']
```

```
})
export class SendFileComponent implements OnInit, OnDestroy {
  private subscriptions: Subscription[] = [];
  private filename: string = '';
  captionForm!: UntypedFormGroup;
  uploading = false;
  fileList: NzUploadFile[] = [];

  constructor(
    private nzMessageService: NzMessageService,
    private messageService: MessageService,
    private formBuilder: UntypedFormBuilder,
) {}

  ngOnInit(): void {
    this.captionForm = this.formBuilder.group({
      caption: [null, [Validators.maxLength(1000)]]
    });
  }

  ngOnDestroy(): void {
    this.subscriptions.forEach(
      subscription => subscription.unsubscribe()
    );
  }

  beforeUpload = (file: NzUploadFile): boolean => {
    this.fileList = this.fileList.concat(file);
    this.uploadToMinio(file);
    return false;
  };

  handleUpload(): void {
    this.uploading = true;
    const caption = this.captionForm.value.caption ? this.captionForm.value.caption : '';
    const subscription = this.messageService.sendFileWithMinio(caption.trim(), this.filename).subscribe(
      () => {
        this.uploading = false;
        this.fileList = [];
        this.filename = '';
        this.captionForm.reset();
        this.nzMessageService.success('Ви успішно надіслали повідомлення');
      }
    );
    this.subscriptions.push(subscription);
  }

  uploadToMinio(file: NzUploadFile): Subscription {
    this.uploading = true;
    return this.messageService.uploadToMinio(file).subscribe(
      (fileResponse) => {
        this.uploading = false;
```

```
      this.filename = fileResponse.filenames[0];
    }, (error) => {
      console.log(`error = ${JSON.stringify(error)}`)
    }
   )
  }
}
```

frontend/src/app/page/send-message/form/send-photo-group/send-photo-group.component.html

```
<nz-row nzJustify="center">
  <nz-upload
    nzType="drag"
    [nzShowButton]="true"
    nzListType="picture"
    nzAccept="image/*"
    [nzSize]="10240"
    [nzMultiple]="true"
    [nzDisabled]="fileList.length === 50"
    [(nzFileList)]="fileList" [nzBeforeUpload]="beforeUpload"
    [nzRemove]="onRemoveFile"

  >
    <p class="ant-upload-drag-icon">
      <span nz-icon nzType="inbox"></span>
    </p>
    <p class="ant-upload-text">Щоб завантажити клікніть або перетягніть файл</p>
    <p class="ant-upload-hint">
      Підтримуються зображення. Максимальна кількість фото 50, максимальний розмір - 10MB
    </p>
  </nz-upload>
</nz-row>
<nz-row nzJustify="end" nzAlign="bottom">
  <button
    nz-button
    [nzType]="'primary'"
    [nzLoading]="uploading"
    (click)="handleUpload()"
    [disabled]="fileList.length === 0"
    class="upload-btn"
  >
    {{ uploading ? 'Завантаження' : 'Надіслати' }}
  </button>
</nz-row>
```

frontend/src/app/page/send-message/form/send-photo-group/send-photo-group.component.ts

```
import { Component, OnDestroy } from '@angular/core';
import { Subscription } from 'rxjs';
import { NzUploadFile } from 'ng-zorro-antd/upload';
import { NzMessageService } from 'ng-zorro-antd/message';
import { MessageService } from '../../../../service/message.service';

@Component({
  selector: 'app-send-photo-group',
  templateUrl: './send-photo-group.component.html',
  styleUrls: ['./send-photo-group.component.scss']
})
export class SendPhotoGroupComponent implements OnDestroy {
  private subscriptions: Subscription[] = [];
  private filenameMinioIdMap: Map<string, string> = new Map<string, string>();
  uploading = false;
  fileList: NzUploadFile[] = [];

  constructor(
    private nzMessageService: NzMessageService,
    private messageService: MessageService,
  ) {}

  ngOnDestroy(): void {
    this.subscriptions.forEach(
      subscription => subscription.unsubscribe()
    );
  }

  beforeUpload = (file: NzUploadFile): boolean => {
    this.fileList = this.fileList.concat(file);
    this.uploadToMinio(file);
    return false;
  };



  onRemoveFile = (file: NzUploadFile): boolean => {
    this.filenameMinioIdMap.delete(file.uid);
    return true;
  };

  handleUpload(): void {
    this.uploading = true;
    const filenames: string[] = [... this.filenameMinioIdMap.values()];
    const subscription = this.messageService.sendPhotoGroupWithMinio(filenames).subscribe(
      () => {
```

```
        this.uploading = false;
        this.fileList = [];
        this.filenameMinioIdMap.clear();
        this.nzMessageService.success('Ви успішно надіслали повідомлення');
      }
    );
    this.subscriptions.push(subscription);
  }

  uploadToMinio(file: NzUploadFile): Subscription {
    this.uploading = true;
    return this.messageService.uploadToMinio(file).subscribe(
      (fileResponse) => {
        this.uploading = false;
        this.filenameMinioIdMap.set(file.uid, fileResponse.filenames[0]);
      }, (error) => {
        console.log(`error = ${JSON.stringify(error)}`)
      }
    )
  }
}
```

frontend/src/app/page/send-message/form/send-photo/send-photo.component.html

```
<form nz-form [formGroup]="captionForm">
  <nz-form-item nzJustify="center" class="caption">
    <nz-form-label nzSpan="4" nzFor="caption">Припис до фото</nz-form-label>
    <nz-form-control nzSpan="8">
      <nz-textarea-count [nzMaxCharacterCount]="1000">
      <textarea
        nz-input
        rows="4"
        type="text"
        name="caption"
        id="caption"
        formControlName="caption"
        placeholder="Це поле не обов'язкове"
      ></textarea>
      </nz-textarea-count>
    </nz-form-control>
  </nz-form-item>
  <nz-form-item nzJustify="center">
    <nz-upload
      nzType="drag"
      [nzShowButton]="true"
      nzListType="picture"
```

```
      nzAccept="image/*"
      [nzSize]="10240"
      [nzDisabled]="fileList.length !== 0"
      [(nzFileList)]="fileList" [nzBeforeUpload]="beforeUpload"
    >
      <p class="ant-upload-drag-icon">
        <span nz-icon nzType="inbox"></span>
      </p>
      <p class="ant-upload-text">Щоб завантажити клікніть або перетягніть файл</p>
      <p class="ant-upload-hint">
        Підтримуються зображення. Максимальна кількість фото 1, максимальний розмір - 10MB
      </p>
    </nz-upload>
  </nz-form-item>
  <nz-form-item nzJustify="end" nzAlign="bottom">
    <button
      nz-button
      [nzType]="'primary'"
      [nzLoading]="uploading"
      (click)="handleUpload()"
      [disabled]="fileList.length === 0 || captionForm.invalid"
      class="upload-btn"
    >
      {{ uploading ? 'Завантаження' : 'Надіслати' }}
    </button>
  </nz-form-item>
</form>
```

frontend/src/app/page/send-message/form/send-photo/send-photo.component.ts

```
import { Component, OnDestroy, OnInit } from '@angular/core';
import {NzUploadFile} from 'ng-zorro-antd/upload';
import { NzMessageService } from 'ng-zorro-antd/message';
import { MessageService } from '../../../../service/message.service';
import { UntypedFormBuilder, UntypedFormGroup, Validators } from '@angular/forms';
import { Subscription } from 'rxjs';

@Component({
  selector: 'app-send-photo',
  templateUrl: './send-photo.component.html',
  styleUrls: ['./send-photo.component.scss']
})
export class SendPhotoComponent implements OnInit, OnDestroy {
  private subscriptions: Subscription[] = [];
  private filename: string = '';
  captionForm!: UntypedFormGroup;
```

```
uploading = false;
fileList: NzUploadFile[] = [];

constructor(
  private nzMessageService: NzMessageService,
  private messageService: MessageService,
  private formBuilder: UntypedFormBuilder,
  ) {}

ngOnInit(): void {
  this.captionForm = this.formBuilder.group({
    caption: [null, [Validators.maxLength(1000)]]
  });
}

ngOnDestroy(): void {
  this.subscriptions.forEach(
    subscription => subscription.unsubscribe()
  );
}

beforeUpload = (file: NzUploadFile): boolean => {
  this.fileList = this.fileList.concat(file);
  this.uploadToMinio(file);
  return false;
};

handleUpload(): void {
  this.uploading = true;
  const caption = this.captionForm.value.caption ? this.captionForm.value.caption : '';
  const subscription = this.messageService.sendPhotoWithMinio(caption.trim(), this.filename).subscribe(
    () => {
      this.uploading = false;
      this.fileList = [];
      this.filename = '';
      this.captionForm.reset();
      this.nzMessageService.success('Ви успішно надіслали повідомлення');
    }
  );
  this.subscriptions.push(subscription);
}

uploadToMinio(file: NzUploadFile): Subscription {
  this.uploading = true;
  return this.messageService.uploadToMinio(file).subscribe(
    (fileResponse) => {
      this.uploading = false;
      this.filename = fileResponse.filenames[0];
    }, (error) => {
        console.log(`error = ${JSON.stringify(error)}`)
    }
  )
```

```
  }
}
```

frontend/src/app/page/send-message/form/send-text/send-text.component.ts

```
import { Component, OnDestroy, OnInit } from '@angular/core';
import { UntypedFormBuilder, UntypedFormGroup, Validators } from '@angular/forms';
import { NzMessageService } from 'ng-zorro-antd/message';
import { MessageService } from '../../../../service/message.service';
import { Subscription } from 'rxjs';

@Component({
  selector: 'app-send-text',
  templateUrl: './send-text.component.html',
  styleUrls: ['./send-text.component.scss']
})
export class SendTextComponent implements OnInit, OnDestroy {
  textForm!: UntypedFormGroup;
  private messageSubscription!: Subscription;

  constructor(
    private formBuilder: UntypedFormBuilder,
    private nzMessageService: NzMessageService,
    private messageService: MessageService,
  ) {}

  ngOnInit(): void {
    this.textForm = this.formBuilder.group({
      text: [null, [Validators.required,
        Validators.maxLength(4000),
        Validators.pattern(/^(?!\s*$).+/),
      ]],
    });
  }

  ngOnDestroy(): void {
    if (this.messageSubscription) {
      this.messageSubscription.unsubscribe();
    }
  }

  sendMessage() {
    if (this.textForm.valid) {
      const { text } = this.textForm.value;
      this.messageSubscription= this.messageService.sendAsyncTextMessage(text.trim()).subscribe(
        () => {
```

```
        this.textForm.reset();
        this.nzMessageService.success("Ви успішно надіслали повідомлення")
      }
    );
  } else {
    Object.values(this.textForm.controls).forEach(control => {
      if (control.invalid) {
        control.markAsDirty();
        control.updateValueAndValidity({ onlySelf: true });
      }
    });
  }
 }
}
```

frontend/src/app/service/message.service.ts

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';
import { environment } from '../../environments/environment';
import {TelegramSendFileRequestDto} from "../dto/telegram-send-file-request-dto";

@Injectable({
  providedIn: 'root'
})
export class MessageService {
  private readonly apiUrl = environment.apiBaseUrl;
  private readonly sendTextMessageUrl = `${this.apiUrl}/tg/broadcast/text`;
  private readonly sendAsyncTextMessageUrl = `${this.apiUrl}/tg/broadcast/text`;
  private readonly sendPhotoMessageUrl = `${this.apiUrl}/tg/broadcast/photo`;
  private readonly sendPhotoMessageWithMinioUrl = `${this.apiUrl}/tg/v2/broadcast/photo`;
  private readonly sendDocumentMessageUrl = `${this.apiUrl}/tg/broadcast/file`;
  private readonly sendDocumentMessageWithMinioUrl = `${this.apiUrl}/tg/v2/broadcast/file`;
  private readonly sendPhotoGroupMessageUrl = `${this.apiUrl}/tg/broadcast/mediaGroup/photo`;
  private readonly sendPhotoGroupMessageWithMinioUrl = `${this.apiUrl}/tg/v2/broadcast/mediaGroup/photo`;
  private readonly uploadToMinioUrl = `${this.apiUrl}/minio/upload`;


  constructor(
    private httpClient: HttpClient,
  ) { }

  sendTextMessage(text: string): Observable<void> {
    return this.httpClient.post<void>(
      this.sendTextMessageUrl,
      text,
      );
```

```
}

sendAsyncTextMessage(text: string): Observable<void> {
  return this.httpClient.post<void>(
    this.sendAsyncTextMessageUrl,
    text,
  );
}

sendPhoto(text: string, photo: any): Observable<void> {
  const formData = new FormData();
  formData.append('caption', text);
  formData.append('photo', photo);
  return this.httpClient.post<void>(
    this.sendPhotoMessageUrl,
    formData
  );
}

sendPhotoWithMinio(text: string, photo: string): Observable<void> {
  const requestDto = new TelegramSendFileRequestDto(text, [photo])
  return this.httpClient.post<void>(
    this.sendPhotoMessageWithMinioUrl,
    requestDto
  );
}

sendFile(text: string, document: any): Observable<void> {
  const formData = new FormData();
  formData.append('caption', text);
  formData.append('document', document);
  return this.httpClient.post<void>(
    this.sendDocumentMessageUrl,
    formData
  );
}

sendFileWithMinio(text: string, document: string): Observable<void> {
  const requestDto = new TelegramSendFileRequestDto(text, [document])
  return this.httpClient.post<void>(
    this.sendDocumentMessageWithMinioUrl,
    requestDto
  );
}

sendPhotoGroup(photos: any[]): Observable<void> {
  const formData = new FormData();
  photos.forEach(
    photo => formData.append('photo', photo)
  );
  return this.httpClient.post<void>(
    this.sendPhotoGroupMessageUrl,
```

```
      formData
    );
  }

  sendPhotoGroupWithMinio(photos: string[]): Observable<void> {
    const requestDto = new TelegramSendFileRequestDto('', [... photos])
    return this.httpClient.post<void>(
      this.sendPhotoGroupMessageWithMinioUrl,
      requestDto
    );
  }

  uploadToMinio(file: any): Observable<any> {
    const formData = new FormData();
    formData.append('file', file);
    return this.httpClient.post<string>(
      this.uploadToMinioUrl,
      formData
    )
  }
}
```

itki-bot-api/Dockerfile

```
FROM openjdk:21-oracle AS builder
WORKDIR /home/api
COPY . /home/api
RUN ./mvnw clean package

# Run application
FROM openjdk:21-oracle
COPY --from=builder /home/api/target/*.jar app.jar
ENTRYPOINT ["java", "-jar", "/app.jar"]
EXPOSE 5005
```

itki-bot-api/Dockerfile.live

```
FROM openjdk:21-oracle
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} app.jar
ENTRYPOINT ["java", "-jar", "/app.jar"]
EXPOSE 5005
```

itki-bot-api/pom.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.3.0</version>
    <relativePath/>
    <!-- lookup parent from repository -->
  </parent>
  <groupId>com.example</groupId>
  <artifactId>itki-bot-api</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <name>itki-bot-api</name>
  <description>itki-bot-api</description>
  <properties>
    <java.version>21</java.version>
    <maven.checkstyle.plugin.version>3.4.0</maven.checkstyle.plugin.version>
    <maven.checkstyle.plugin.configLocation>
      checkstyle.xml
         </maven.checkstyle.plugin.configLocation>
  </properties>
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>jakarta.validation</groupId>
        <artifactId>jakarta.validation-api</artifactId>
        <version>3.1.0</version>
      </dependency>
    </dependencies>
  </dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-security</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
      <groupId>org.liquibase</groupId>
```

```
    <artifactId>liquibase-core</artifactId>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt</artifactId>
    <version>0.9.1</version>
</dependency>
<dependency>
    <groupId>jakarta.validation</groupId>
    <artifactId>jakarta.validation-api</artifactId>
</dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-validation</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
            <scope>runtime</scope>
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>org.postgresql</groupId>
            <artifactId>postgresql</artifactId>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.telegram</groupId>
            <artifactId>telegrambots</artifactId>
            <version>6.9.7.1</version>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-test</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-redis</artifactId>
        </dependency>
        <dependency>
            <groupId>redis.clients</groupId>
```

```
				<artifactId>jedis</artifactId>
			</dependency>
			<dependency>
				<groupId>io.minio</groupId>
				<artifactId>minio</artifactId>
				<version>8.5.11</version>
			</dependency>
	</dependencies>

	<build>
		<plugins>
			<plugin>
				<groupId>org.springframework.boot</groupId>
				<artifactId>spring-boot-maven-plugin</artifactId>
				<configuration>
					<excludes>
						<exclude>
							<groupId>org.projectlombok</groupId>
							<artifactId>lombok</artifactId>
						</exclude>
					</excludes>
				</configuration>
			</plugin>
			<plugin>
				<groupId>org.apache.maven.plugins</groupId>
				<artifactId>maven-checkstyle-plugin</artifactId>
				<version>${maven.checkstyle.plugin.version}</version>
				<executions>
					<execution>
						<phase>compile</phase>
						<goals>
							<goal>check</goal>
						</goals>
					</execution>
				</executions>
				<configuration>
					<configLocation>${maven.checkstyle.plugin.configLocation}</configLocation>
					<encoding>UTF-8</encoding>
					<consoleOutput>true</consoleOutput>
					<failsOnError>true</failsOnError>
					<linkXRef>false</linkXRef>
				</configuration>
			</plugin>
		</plugins>
	</build>

</project>
```

itki-bot-api/src/main/java/com/itki/api/config/AsyncConfig.java

```
package com.itki.api.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.ThreadFactory;

@Configuration
public class AsyncConfig {
  private static final int THREAD_POOL_SIZE = 10;

  @Bean(destroyMethod = "shutdown")
  public ExecutorService getExecutor() {
    ThreadFactory virtualThreadFactory = Thread.ofVirtual().factory();
    return Executors.newFixedThreadPool(THREAD_POOL_SIZE, virtualThreadFactory);
  }
}
```

itki-bot-api/src/main/java/com/itki/api/config/MinioConfig.java

```
package com.itki.api.config;

import io.minio.MinioClient;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class MinioConfig {
  public static final String BUCKET_NAME = "my-data";
  @Value("${MINIO_ACCESS_KEY}")
  private String accessKey;
  @Value("${MINIO_SECRET_KEY}")
  private String secretKey;
  @Value("${MINIO_URL}")
  private String minioUrl;

  @Bean
  public MinioClient minioClient() {
    return MinioClient.builder()
        .endpoint(minioUrl)
        .credentials(accessKey, secretKey)
        .build();
  }
```

```
}
```

itki-bot-api/src/main/java/com/itki/api/config/PasswordEncoderConfig.java

```
package com.itki.api.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;

@Configuration
public class PasswordEncoderConfig {
  @Bean
  @Primary
  public PasswordEncoder getEncoder() {
    return new BCryptPasswordEncoder();
  }
}
```

itki-bot-api/src/main/java/com/itki/api/config/RedisConfig.java

```
package com.itki.api.config;

import com.itki.api.model.User;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.redis.connection.RedisStandaloneConfiguration;
import org.springframework.data.redis.connection.jedis.JedisConnectionFactory;
import org.springframework.data.redis.core.RedisTemplate;

@Configuration
public class RedisConfig {
  @Value("${REDIS_HOSTNAME}")
  private String redisHostName;
  @Value("${REDIS_PORT}")
  private int redisPort;

  @Bean
  JedisConnectionFactory jedisConnectionFactory() {
```

```
    RedisStandaloneConfiguration redisStandaloneConfiguration =
        new RedisStandaloneConfiguration(redisHostName, redisPort);
    return new JedisConnectionFactory(redisStandaloneConfiguration);
  }

  @Bean
  public RedisTemplate<String, User> userByLoginCache(
      JedisConnectionFactory jedisConnectionFactory
  ) {
    RedisTemplate<String, User> template = new RedisTemplate<>();
    template.setConnectionFactory(jedisConnectionFactory);
    return template;
  }

  @Bean
  public RedisTemplate<String, String> tokenBlackListCache(
      JedisConnectionFactory jedisConnectionFactory
  ) {
    RedisTemplate<String, String> template = new RedisTemplate<>();
    template.setConnectionFactory(jedisConnectionFactory);
    return template;
  }
}
```

itki-bot-api/src/main/java/com/itki/api/config/SecurityConfig.java

```
package com.itki.api.config;

import com.itki.api.jwt.JwtTokenFilter;
import com.itki.api.jwt.JwtTokenProvider;
import lombok.RequiredArgsConstructor;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.HttpMethod;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configurers.AbstractHttpConfigurer;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.web.SecurityFilterChain;
import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;
import org.springframework.web.cors.CorsConfiguration;
import org.springframework.web.cors.CorsConfigurationSource;
import org.springframework.web.cors.UrlBasedCorsConfigurationSource;
import java.util.List;

@Configuration
@RequiredArgsConstructor
public class SecurityConfig {
  private final JwtTokenProvider jwtTokenProvider;
```

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
  http.cors(cors -> cors.configurationSource(corsConfigurationSource()));
  http.httpBasic(AbstractHttpConfigurer::disable);
  http.logout(AbstractHttpConfigurer::disable);
  http.authorizeHttpRequests(authManager -> authManager
      .requestMatchers(
          HttpMethod.GET,
          "/questions",
          "/curators",
          "/curators/**",
          "/groups",
          "/telegram-users"
      ).hasAnyRole("ADMIN", "USER")
      .requestMatchers(
          HttpMethod.POST,
          "/questions",
          "/curators",
          "/groups",
          "/telegram-users",
          "/tg/**"
      ).hasRole("ADMIN")
      .requestMatchers(
          HttpMethod.DELETE,
          "/questions/**",
          "/curators/**",
          "/groups/**",
          "/telegram-users/**"
      ).hasRole("ADMIN")
      .requestMatchers("/success").permitAll()
      .requestMatchers("/login", "/refresh", "/logout").permitAll()
      .anyRequest().authenticated()
  );
  http.csrf(AbstractHttpConfigurer::disable).sessionManagement(
      session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS)
  );
  http.addFilterBefore(
      new JwtTokenFilter(jwtTokenProvider),
      UsernamePasswordAuthenticationFilter.class
  );
  return http.build();
}

@Bean
public CorsConfigurationSource corsConfigurationSource() {
  final CorsConfiguration configuration = new CorsConfiguration();
  configuration.setAllowedOriginPatterns(List.of("*"));
  configuration.setAllowedMethods(List.of("*"));
  configuration.setAllowCredentials(true);
  configuration.setAllowedHeaders(List.of("*"));
  configuration.setExposedHeaders(List.of("Access-Control-Allow-Origin",
```

```
        "Access-Control-Allow-Methods", "Access-Control-Allow-Headers", "Access-Control-Max-Age",
        "Access-Control-Request-Headers", "Access-Control-Request-Method"));
    final UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
    source.registerCorsConfiguration("/**", configuration);
    return source;
  }
}
```

itki-bot-api/src/main/java/com/itki/api/controller/AuthenticationController.java

```
package com.itki.api.controller;

import com.itki.api.dto.LoginRequestDto;
import com.itki.api.dto.TokenDto;
import com.itki.api.service.AuthenticationService;
import jakarta.validation.Valid;
import lombok.RequiredArgsConstructor;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;
import java.util.Optional;

@RestController
@RequiredArgsConstructor
public class AuthenticationController {
  private final AuthenticationService authenticationService;

  @PostMapping("/login")
  public ResponseEntity<TokenDto> login(
      @Valid @RequestBody LoginRequestDto loginRequestDto
  ) {
    Optional<TokenDto> loginResponseDto = authenticationService
        .login(loginRequestDto.getLogin(), loginRequestDto.getPassword());
    return getLoginResponseDto(loginResponseDto);
  }

  @PostMapping("/refresh")
  public ResponseEntity<TokenDto> refresh(@RequestBody String refreshToken) {
    Optional<TokenDto> loginResponseDto = authenticationService.login(refreshToken);
    return getLoginResponseDto(loginResponseDto);
  }

  @PostMapping("/logout")
  public ResponseEntity<Boolean> logout(@Valid @RequestBody TokenDto tokenDto) {
```

```
    authenticationService.logout(tokenDto.getToken(), tokenDto.getRefreshToken());
    return ResponseEntity.ok(true);
  }

  private ResponseEntity<TokenDto> getLoginResponseDto(
      Optional<TokenDto> loginResponseDto
  ) {
    if (loginResponseDto.isPresent()) {
      return ResponseEntity.ok(loginResponseDto.get());
    }
    return ResponseEntity.status(HttpStatus.FORBIDDEN).build();
  }
}
```

itki-bot-api/src/main/java/com/itki/api/controller/QuestionController.java

```
package com.itki.api.controller;

import com.itki.api.dto.QuestionRequestDto;
import com.itki.api.dto.QuestionResponseDto;
import com.itki.api.mapper.QuestionDtoMapper;
import com.itki.api.model.Question;
import com.itki.api.service.AnswerService;
import com.itki.api.service.QuestionService;
import lombok.RequiredArgsConstructor;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import java.util.List;
import java.util.stream.Collectors;

@RestController
@RequestMapping("/questions")
@RequiredArgsConstructor
public class QuestionController {
  private final QuestionService questionService;
  private final AnswerService answerService;
  private final QuestionDtoMapper questionDtoMapper;

  @GetMapping
```

```
  public List<QuestionResponseDto> getAll() {
    return questionService.findAll()
        .stream()
        .map(questionDtoMapper::toResponse)
        .collect(Collectors.toList());
  }

  @PostMapping
  public QuestionResponseDto save(@RequestBody QuestionRequestDto questionRequestDto) {
    Question question = questionDtoMapper.toModel(questionRequestDto);
    answerService.save(question.getAnswer());
    questionService.save(question);
    return questionDtoMapper.toResponse(question);
  }

  @DeleteMapping("/{id}")
  public ResponseEntity<HttpStatus> deleteById(@PathVariable Long id) {
    Question question = questionService.findById(id);
    questionService.deleteById(question.getId());
    answerService.deleteById(question.getAnswer().getId());
    return ResponseEntity.ok().build();
  }
}
```

itki-bot-api/src/main/java/com/itki/api/controller/TelegramAsyncController.java

```
package com.itki.api.controller;

import com.itki.api.config.MinioConfig;
import com.itki.api.dto.SendFileToTelegramRequestDto;
import com.itki.api.service.TelegramBroadcastService;
import io.minio.GetObjectArgs;
import io.minio.MinioClient;
import lombok.RequiredArgsConstructor;
import lombok.SneakyThrows;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import java.io.InputStream;
import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.ExecutorService;
```

```
@RestController
@RequestMapping("/tg/v2")
@RequiredArgsConstructor
public class TelegramAsyncController {
  private final TelegramBroadcastService telegramBroadcastService;
  private final MinioClient minioClient;
  private final ExecutorService executorService;

  @PostMapping("/broadcast/photo")
  public ResponseEntity<HttpStatus> sendBroadcastPhoto(
      @RequestBody
      SendFileToTelegramRequestDto requestDto
  ) {
    executorService.execute(() -> {
      InputStream photoFromMinio = getFileFromMinio(requestDto.filenames().get(0));
      telegramBroadcastService.sendPhoto(
          requestDto.caption(),
          photoFromMinio,
          requestDto.filenames().get(0)
      );
    });
    return ResponseEntity.ok().build();
  }

  @PostMapping("/broadcast/file")
  @SneakyThrows
  public ResponseEntity<HttpStatus> sendBroadcastFile(
      @RequestBody
      SendFileToTelegramRequestDto requestDto
  ) {
    executorService.execute(() -> {
      InputStream fileFromMinio = getFileFromMinio(requestDto.filenames().get(0));
      telegramBroadcastService.sendFile(
          requestDto.caption(),
          fileFromMinio, requestDto.filenames().get(0)
      );
    });
    return ResponseEntity.ok().build();
  }

  @PostMapping("/broadcast/mediaGroup/photo")
  @SneakyThrows
  public ResponseEntity<HttpStatus> sendBroadcastPhotoGroup(
      @RequestBody
      SendFileToTelegramRequestDto requestDto
  ) {
    executorService.execute(() -> {
      Map<String, InputStream> filenameInputStreamMap = new HashMap<>();
      for (String filename : requestDto.filenames()) {
        InputStream fileFromMinio = getFileFromMinio(filename);
        filenameInputStreamMap.put(filename, fileFromMinio);
      }
```

```
    telegramBroadcastService.sendPhotoMediaGroup(filenameInputStreamMap);
   });
   return ResponseEntity.ok().build();
  }

  @PostMapping("/broadcast/text")
  public ResponseEntity<HttpStatus> sendBroadcastText(
     @RequestBody String text
  ) {
   executorService.execute(() -> {
    telegramBroadcastService.sendTextMessage(text);
   });
   return ResponseEntity.ok().build();
  }

  private InputStream getFileFromMinio(String filename) {
   try {
    return minioClient.getObject(
      GetObjectArgs.builder()
        .bucket(MinioConfig.BUCKET_NAME)
        .object(filename)
        .build()
    );
   } catch (Exception e) {
    throw new RuntimeException("Error getting file"
      + filename
      + " from Minio", e);
   }
  }
}
```

itki-bot-api/src/main/java/com/itki/api/controller/TelegramController.java

```
package com.itki.api.controller;

import com.itki.api.service.TelegramBroadcastService;
import lombok.RequiredArgsConstructor;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.multipart.MultipartFile;
```

```
@RestController
@RequestMapping("/tg")
@RequiredArgsConstructor
public class TelegramController {
  private final TelegramBroadcastService telegramBroadcastService;

  @PostMapping("/broadcast/file")
  public ResponseEntity<HttpStatus> sendBroadcastFile(
      @RequestParam("document") MultipartFile multipartFile,
      @RequestParam String caption
  ) {
    telegramBroadcastService.sendFile(caption, multipartFile);
    return ResponseEntity.ok().build();
  }

  @PostMapping("/broadcast/photo")
  public ResponseEntity<HttpStatus> sendBroadcastPhoto(
      @RequestParam("photo") MultipartFile multipartFile,
      @RequestParam String caption
  ) {
    telegramBroadcastService.sendPhoto(caption, multipartFile);
    return ResponseEntity.ok().build();
  }

  @PostMapping("/broadcast/text")
  public ResponseEntity<HttpStatus> sendBroadcastText(
      @RequestBody String text
  ) {
    telegramBroadcastService.sendTextMessage(text);
    return ResponseEntity.ok().build();
  }

  @PostMapping("/broadcast/mediaGroup/photo")
  public ResponseEntity<HttpStatus> sendBroadcastPhotoGroup(
      @RequestParam("photo") MultipartFile[] multipartFiles
  ) {
    telegramBroadcastService.sendPhotoMediaGroup(multipartFiles);
    return ResponseEntity.ok().build();
  }
}
```

itki-bot-api/src/main/java/com/itki/api/controller/UploadToMinioController.java

```
package com.itki.api.controller;

import com.itki.api.config.MinioConfig;
```

```
import com.itki.api.dto.UploadResponse;
import io.minio.MinioClient;
import io.minio.PutObjectArgs;
import lombok.RequiredArgsConstructor;
import lombok.SneakyThrows;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestHeader;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.multipart.MultipartFile;
import java.util.List;

@Controller
@RequestMapping("/minio")
@RequiredArgsConstructor
public class UploadToMinioController {
  private final MinioClient minioClient;

  @PostMapping("/upload")
  @SneakyThrows
  public ResponseEntity<UploadResponse> uploadToMinio(@RequestHeader MultipartFile file) {
    String filename = String.format(
        "[%d] %s",
        System.currentTimeMillis(),
        file.getOriginalFilename()
    );
    PutObjectArgs.Builder putObjectArgs = PutObjectArgs.builder()
        .bucket(MinioConfig.BUCKET_NAME)
        .object(filename)
        .stream(file.getInputStream(), file.getSize(), -1);
    minioClient.putObject(putObjectArgs.build());
    return ResponseEntity.ok().body(new UploadResponse(List.of(filename)));
  }
}
```

itki-bot-api/src/main/java/com/itki/api/dto/LoginRequestDto.java

```
package com.itki.api.dto;

import jakarta.validation.constraints.NotBlank;
import lombok.Data;

@Data
public class LoginRequestDto {
  @NotBlank
  private String login;
  @NotBlank
```

```
  private String password;
}
```

itki-bot-api/src/main/java/com/itki/api/dto/SendFileToTelegramRequestDto.java

```
package com.itki.api.dto;

import java.util.List;

public record SendFileToTelegramRequestDto(
    List<String> filenames,
    String caption
) {
}
```

itki-bot-api/src/main/java/com/itki/api/dto/TokenDto.java

```
package com.itki.api.dto;

import java.util.List;

public record UploadResponse(List<String> filenames) {
}
```

itki-bot-api/src/main/java/com/itki/api/jwt/JwtTokenFilter.java

```
package com.itki.api.jwt;

import io.jsonwebtoken.JwtException;
import java.io.IOException;
import java.util.Optional;
import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import org.springframework.http.HttpStatus;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.context.SecurityContextHolder;
```

```
import org.springframework.web.filter.OncePerRequestFilter;

public class JwtTokenFilter extends OncePerRequestFilter {
  private final JwtTokenProvider jwtTokenProvider;

  public JwtTokenFilter(JwtTokenProvider jwtTokenProvider) {
    this.jwtTokenProvider = jwtTokenProvider;
  }

  @Override
  protected void doFilterInternal(
    HttpServletRequest request,
    HttpServletResponse response,
    FilterChain filterChain
  ) throws IOException, ServletException {
    Optional<String> token = jwtTokenProvider.resolveTokenFromRequest(request);
    if (token.isPresent()) {
      try {
        jwtTokenProvider.validateToken(token.get());
        Authentication auth = jwtTokenProvider.getAuthentication(token.get());
        SecurityContextHolder.getContext().setAuthentication(auth);
      } catch (JwtException e) {
        response.setStatus(HttpStatus.FORBIDDEN.value());
      }
    }
    filterChain.doFilter(request, response);
  }
}
```

itki-bot-api/src/main/java/com/itki/api/jwt/JwtTokenProvider.java

```
package com.itki.api.jwt;

import com.itki.api.service.TokenBlackListService;
import io.jsonwebtoken.Claims;
import io.jsonwebtoken.JwtException;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;
import jakarta.annotation.PostConstruct;
import java.time.Duration;
import java.util.Base64;
import java.util.Date;
import java.util.Optional;
import jakarta.servlet.http.HttpServletRequest;
import lombok.RequiredArgsConstructor;
import lombok.extern.log4j.Log4j2;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
```

```
import org.springframework.security.core.Authentication;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.stereotype.Component;

@Log4j2
@RequiredArgsConstructor
@Component
public class JwtTokenProvider {
 private static final int HEADER_OFFSET = 7;
 @Value("${security.jwt.token.expire-length}")
 private long accessTokenValidPeriod;
 @Value("#{T(java.time.Duration).ofDays('${REFRESH_TOKEN_TTL_IN_DAYS}')}")
 private Duration refreshTokenValidDuration;

 @Value("${security.jwt.token.secret-key}")
 private String secretKey;
 private final UserDetailsService userDetailsService;
 private final TokenBlackListService tokenBlackListService;

 @PostConstruct
 protected void init() {
  secretKey = Base64.getEncoder().encodeToString(secretKey.getBytes());
 }

 public String createAccessToken(String profileName) {
  return buildToken(profileName, accessTokenValidPeriod);
 }

 public String createRefreshToken(String profileName) {
  return buildToken(profileName, refreshTokenValidDuration.toMillis());
 }

 public Authentication getAuthentication(String token) {
  UserDetails userDetails = this.userDetailsService.loadUserByUsername(getUsername(token));
  return new UsernamePasswordAuthenticationToken(
   userDetails, "", userDetails.getAuthorities());
 }

 public String getUsername(String token) {
  return Jwts.parser().setSigningKey(secretKey).parseClaimsJws(token).getBody().getSubject();
 }

 public Optional<String> resolveTokenFromRequest(HttpServletRequest req) {
  String bearerToken = req.getHeader("Authorization");
  return resolveToken(bearerToken);
 }

 public Optional<String> resolveToken(String token) {
  if (token != null && token.startsWith("Bearer ")) {
   return Optional.of(token.substring(HEADER_OFFSET));
  }
```

```
    return Optional.empty();
  }

  public void validateToken(String token) {
    if (Jwts.parser().setSigningKey(secretKey).parseClaimsJws(token)
      .getBody().getExpiration().before(new Date())) {
      throw new JwtException("Jwt token is expired");
    } else if (tokenBlackListService.isTokenInBlackList(token)) {
      throw new JwtException("Jwt token was used");
    }
  }

  private String buildToken(String profileName, long refreshTokenValidDuration) {
    Claims claims = Jwts.claims().setSubject(profileName);
    Date now = new Date();
    Date validity = new Date(now.getTime() + refreshTokenValidDuration);
    return Jwts.builder()
        .setClaims(claims)
        .setIssuedAt(now)
        .setExpiration(validity)
        .signWith(SignatureAlgorithm.HS256, secretKey)
        .compact();
  }
}
```

itki-bot-api/src/main/java/com/itki/api/model/Answer.java

```
package com.itki.api.model;

import lombok.Getter;
import lombok.Setter;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.SequenceGenerator;
import jakarta.persistence.Table;
import java.io.Serializable;

@Getter
@Setter
@Entity
@Table(name = "answer")
public class Answer implements Serializable {
  @Id
  @GeneratedValue(generator = "answer_id_seq", strategy = GenerationType.SEQUENCE)
  @SequenceGenerator(
      name = "answer_id_seq",
```

```
    sequenceName = "answer_id_seq",
    allocationSize = 1
  )
  private Long id;
  private String text;
}
```

itki-bot-api/src/main/java/com/itki/api/model/Curator.java

```
package com.itki.api.model;

import lombok.Getter;
import lombok.Setter;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.SequenceGenerator;
import jakarta.persistence.Table;
import java.io.Serializable;

@Getter
@Setter
@Entity
@Table(name = "curator")
public class Curator implements Serializable {
  @Id
  @GeneratedValue(generator = "curator_id_seq", strategy = GenerationType.SEQUENCE)
  @SequenceGenerator(
    name = "curator_id_seq",
    sequenceName = "curator_id_seq",
    allocationSize = 1
  )
  private Long id;
  private String name;
  private String lastName;
  private String additionalName;
  private String department;
  private String position;
}
```

itki-bot-api/src/main/java/com/itki/api/model/Group.java

```
package com.itki.api.model;
```

```
import lombok.Getter;
import lombok.Setter;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.ManyToOne;
import jakarta.persistence.SequenceGenerator;
import jakarta.persistence.Table;
import java.io.Serializable;

@Getter
@Setter
@Entity
@Table(name = "[group]")
public class Group implements Serializable {
  @Id
  @GeneratedValue(generator = "group_id_seq", strategy = GenerationType.SEQUENCE)
  @SequenceGenerator(
      name = "group_id_seq",
      sequenceName = "group_id_seq",
      allocationSize = 1
  )
  private Long id;
  private String name;
  @ManyToOne
  @JoinColumn(name = "curator_id")
  private Curator curator;
}
```

itki-bot-api/src/main/java/com/itki/api/model/Question.java

```
package com.itki.api.model;

import lombok.Getter;
import lombok.Setter;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.OneToOne;
import jakarta.persistence.SequenceGenerator;
import jakarta.persistence.Table;
import java.io.Serializable;

@Getter
```

```
@Setter
@Entity
@Table(name = "question")
public class Question implements Serializable {
  @Id
  @GeneratedValue(generator = "question_id_seq", strategy = GenerationType.SEQUENCE)
  @SequenceGenerator(
      name = "question_id_seq",
      sequenceName = "question_id_seq",
      allocationSize = 1
  )
  private Long id;
  private String text;
  @OneToOne
  private Answer answer;
}
```

itki-bot-api/src/main/java/com/itki/api/model/Role.java

```
package com.itki.api.model;

import jakarta.persistence.Entity;
import jakarta.persistence.EnumType;
import jakarta.persistence.Enumerated;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.SequenceGenerator;
import jakarta.persistence.Table;
import lombok.Getter;
import lombok.Setter;
import java.io.Serializable;

@Getter
@Setter
@Entity
@Table(name = "role")
public class Role implements Serializable {
  @Id
  @GeneratedValue(generator = "role_id_seq", strategy = GenerationType.SEQUENCE)
  @SequenceGenerator(
      name = "role_id_seq",
      sequenceName = "role_id_seq",
      allocationSize = 1
  )
  private Long id;
  @Enumerated(EnumType.STRING)
  private RoleName name;
```

```
  public enum RoleName {
    ADMIN, USER
  }
}
```

itki-bot-api/src/main/java/com/itki/api/model/TelegramUser.java

```
package com.itki.api.model;

import lombok.Getter;
import lombok.Setter;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.SequenceGenerator;
import jakarta.persistence.Table;
import java.io.Serializable;

@Getter
@Setter
@Entity
@Table(name = "telegram_user")
public class TelegramUser implements Serializable {
  @Id
  @GeneratedValue(generator = "telegram_user_id_seq", strategy = GenerationType.SEQUENCE)
  @SequenceGenerator(
      name = "telegram_user_id_seq",
      sequenceName = "telegram_user_id_seq",
      allocationSize = 1
  )
  private Long id;
  private String firstName;
  private String lastName;
  private String username;
  private String externalChatId;
}
```

itki-bot-api/src/main/java/com/itki/api/model/User.java

```
package com.itki.api.model;

import lombok.Getter;
```

```
import lombok.Setter;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.JoinTable;
import jakarta.persistence.ManyToMany;
import jakarta.persistence.SequenceGenerator;
import jakarta.persistence.Table;
import java.io.Serializable;
import java.util.Set;

@Getter
@Setter
@Entity
@Table(name = "[user]")
public class User implements Serializable {
  @Id
  @GeneratedValue(generator = "user_id_seq", strategy = GenerationType.SEQUENCE)
  @SequenceGenerator(
    name = "user_id_seq",
    sequenceName = "user_id_seq",
    allocationSize = 1
  )
  private Long id;
  private String login;
  private String password;
  @ManyToMany
  @JoinTable(
    name = "user_role",
    joinColumns = @JoinColumn(name = "user_id"),
    inverseJoinColumns = @JoinColumn(name = "role_id")
  )
  private Set<Role> roles;
}
```

itki-bot-api/src/main/java/com/itki/api/service/AuthenticationService.java

```
package com.itki.api.service;

import com.itki.api.dto.TokenDto;
import java.util.Optional;

public interface AuthenticationService {
  Optional<TokenDto> login(String login, String password);
```

```
  Optional<TokenDto> login(String refreshToken);

  void logout(String token, String refreshToken);
}
```

itki-bot-api/src/main/java/com/itki/api/service/TelegramBroadcastService.java

```
package com.itki.api.service;

import org.springframework.web.multipart.MultipartFile;
import java.io.InputStream;
import java.util.Map;

public interface TelegramBroadcastService {
  void sendFile(String caption, MultipartFile file);

  void sendFile(String caption, InputStream file, String filename);

  void sendPhoto(String caption, MultipartFile photo);

  void sendPhoto(String caption, InputStream photo, String filename);

  void sendTextMessage(String text);

  void sendPhotoMediaGroup(MultipartFile[] photos);

  void sendPhotoMediaGroup(Map<String, InputStream> filenameInputStreamMap);
}
```

itki-bot-api/src/main/java/com/itki/api/service/TokenBlackListService.java

```
package com.itki.api.service;

public interface TokenBlackListService {
  void putTokenToBlackList(String token);

  void putRefreshTokenToBlackList(String token);

  boolean isTokenInBlackList(String token);
}
```

itki-bot-api/src/main/java/com/itki/api/service/impl/AuthenticationServiceImpl.java

```
package com.itki.api.service.impl;

import com.itki.api.dto.TokenDto;
import com.itki.api.jwt.JwtTokenProvider;
import com.itki.api.model.User;
import com.itki.api.service.AuthenticationService;
import com.itki.api.service.TokenBlackListService;
import com.itki.api.service.UserService;
import java.util.Optional;
import lombok.RequiredArgsConstructor;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.stereotype.Service;

@Service
@RequiredArgsConstructor
public class AuthenticationServiceImpl implements AuthenticationService {
  private final UserService userService;
  private final PasswordEncoder passwordEncoder;
  private final JwtTokenProvider jwtTokenProvider;
  private final TokenBlackListService tokenBlackListService;

  @Override
  public Optional<TokenDto> login(String login, String password) {
    Optional<User> user = userService.findByLogin(login);
    if (user.isEmpty() || !passwordEncoder.matches(password, user.get().getPassword())) {
      return Optional.empty();
    }
    return Optional.of(getLoginResponse(user.get()));
  }

  @Override
  public Optional<TokenDto> login(String refreshToken) {
    jwtTokenProvider.validateToken(refreshToken);
    Optional<User> user = userService
        .findByLogin(jwtTokenProvider.getUsername(refreshToken));
    return user.map(this::getLoginResponse);
  }

  @Override
  public void logout(String token, String refreshToken) {
    jwtTokenProvider.validateToken(token);
    jwtTokenProvider.validateToken(refreshToken);
    tokenBlackListService.putTokenToBlackList(token);
```

```
    tokenBlackListService.putRefreshTokenToBlackList(refreshToken);
  }

  private TokenDto getLoginResponse(User user) {
    String accessToken = jwtTokenProvider.createAccessToken(user.getLogin());
    String refreshToken = jwtTokenProvider.createRefreshToken(user.getLogin());
    TokenDto tokenDto = new TokenDto();
    tokenDto.setToken(accessToken);
    tokenDto.setRefreshToken(refreshToken);
    return tokenDto;
  }


}
```

itki-bot-api/src/main/java/com/itki/api/service/impl/CrudServiceImpl.java

```
package com.itki.api.service.impl;

import com.itki.api.service.CrudService;
import java.util.List;
import lombok.RequiredArgsConstructor;
import org.springframework.data.jpa.repository.JpaRepository;

@RequiredArgsConstructor
public abstract class CrudServiceImpl<T> implements CrudService<T> {
  private final JpaRepository<T, Long> repository;
  private final String entityName;

  @Override
  public T findById(Long id) {
    return repository.findById(id)
      .orElseThrow(() -> new RuntimeException(
      "No entity: '%s' with id: %d".formatted(entityName, id)
      ));
  }

  @Override
  public List<T> findAll() {
    return repository.findAll();
  }

  @Override
  public T save(T entity) {
    return repository.save(entity);
  }
```

```
  @Override
  public void deleteById(Long id) {
    repository.deleteById(id);
  }

  @Override
  public void deleteAll() {
    repository.deleteAll();
  }
}
```

itki-bot-api/src/main/java/com/itki/api/service/impl/TelegramBroadcastServiceImpl.java

```
package com.itki.api.service.impl;

import com.itki.api.service.TelegramBroadcastService;
import com.itki.api.service.TelegramUserService;
import lombok.RequiredArgsConstructor;
import lombok.SneakyThrows;
import lombok.extern.log4j.Log4j2;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Service;
import org.springframework.web.multipart.MultipartFile;
import org.telegram.telegrambots.bots.TelegramWebhookBot;
import org.telegram.telegrambots.meta.api.methods.BotApiMethod;
import org.telegram.telegrambots.meta.api.methods.send.SendDocument;
import org.telegram.telegrambots.meta.api.methods.send.SendMediaGroup;
import org.telegram.telegrambots.meta.api.methods.send.SendMessage;
import org.telegram.telegrambots.meta.api.methods.send.SendPhoto;
import org.telegram.telegrambots.meta.api.objects.InputFile;
import org.telegram.telegrambots.meta.api.objects.Update;
import org.telegram.telegrambots.meta.api.objects.media.InputMedia;
import org.telegram.telegrambots.meta.api.objects.media.InputMediaPhoto;
import org.telegram.telegrambots.meta.exceptions.TelegramApiException;
import org.telegram.telegrambots.meta.exceptions.TelegramApiRequestException;
import java.io.IOException;
import java.io.InputStream;
import java.util.Arrays;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

@Service
@RequiredArgsConstructor
@Log4j2
```

```
public class TelegramBroadcastServiceImpl
    extends TelegramWebhookBot implements TelegramBroadcastService {
  private static final String BLOCKING_MESSAGE = """
      Error sending message: \
      [403] Forbidden: bot was blocked by the user\
      """;
  private final TelegramUserService telegramUserService;
  @Value("${TELEGRAM_TOKEN}")
  private String telegramToken;
  @Value("${TELEGRAM_USERNAME}")
  private String telegramBotUsername;

  @Override
  @SneakyThrows
  public void sendFile(String caption, MultipartFile file) {
    List<String> externalChatIds = telegramUserService.getAllExternalChatIds();
    SendDocument sendDocument = new SendDocument();
    for (String externalChatId: externalChatIds) {
      sendFileToChat(
          caption,
          this.toInputFile(file.getInputStream(), file.getOriginalFilename()),
          externalChatId,
          sendDocument
      );
    }
  }

  @Override
  @SneakyThrows
  public void sendFile(String caption, InputStream file, String filename) {
    List<String> externalChatIds = telegramUserService.getAllExternalChatIds();
    SendDocument sendDocument = new SendDocument();
    for (String externalChatId: externalChatIds) {
      sendFileToChat(
          caption,
          this.toInputFile(file, filename),
          externalChatId,
          sendDocument
      );
    }
  }

  @Override
  @SneakyThrows
  public void sendPhoto(String caption, MultipartFile photo) {
    SendPhoto sendPhoto = new SendPhoto();
    List<String> externalChatIds = telegramUserService.getAllExternalChatIds();
    for (String externalChatId: externalChatIds) {
      sendPhotoToSpecialChat(
          caption,
          this.toInputFile(photo.getInputStream(), photo.getOriginalFilename()),
          externalChatId,
```

```
      sendPhoto
    );
  }
}

@Override
@SneakyThrows
public void sendPhoto(String caption, InputStream photo, String filename) {
  SendPhoto sendPhoto = new SendPhoto();
  List<String> externalChatIds = telegramUserService.getAllExternalChatIds();
  for (String externalChatId: externalChatIds) {
    sendPhotoToSpecialChat(caption, this.toInputFile(photo, filename), externalChatId, sendPhoto);
  }
}

@Override
@SneakyThrows
public void sendTextMessage(String text) {
  SendMessage sendMessage = new SendMessage();
  List<String> externalChatIds = telegramUserService.getAllExternalChatIds();
  for (String externalChatId: externalChatIds) {
    sendMessage.setChatId(externalChatId);
    sendMessage.setText(text);
    try {
      execute(sendMessage);
    } catch (TelegramApiRequestException telegramApiRequestException) {
      handleUserBlocking(externalChatId, telegramApiRequestException);
    }
  }
}

@Override
@SneakyThrows
public void sendPhotoMediaGroup(MultipartFile[] photos) {
  SendMediaGroup sendMediaGroup = new SendMediaGroup();
  List<String> externalChatIds = telegramUserService.getAllExternalChatIds();
  for (String externalChatId: externalChatIds) {
    sendMediaGroup.setMedias(
        Arrays.stream(photos).map(this::toInputPhotoMedia).collect(Collectors.toList())
    );
    sendMediaGroup.setChatId(externalChatId);
    try {
      execute(sendMediaGroup);
    } catch (TelegramApiRequestException telegramApiRequestException) {
      handleUserBlocking(externalChatId, telegramApiRequestException);
    }
  }
}

@Override
@SneakyThrows
public void sendPhotoMediaGroup(Map<String, InputStream> filenameInputStreamMap) {
```

```
  SendMediaGroup sendMediaGroup = new SendMediaGroup();
  List<String> externalChatIds = telegramUserService.getAllExternalChatIds();
  for (String externalChatId: externalChatIds) {
    sendMediaGroup.setMedias(
        filenameInputStreamMap.entrySet().stream()
            .map(entry -> this.toInputPhotoMedia(entry.getValue(), entry.getKey()))
            .collect(Collectors.toList())
    );
    sendMediaGroup.setChatId(externalChatId);
    try {
      execute(sendMediaGroup);
    } catch (TelegramApiRequestException telegramApiRequestException) {
      handleUserBlocking(externalChatId, telegramApiRequestException);
    }
  }
}

@Override
public String getBotUsername() {
  return telegramBotUsername;
}

@Override
public String getBotToken() {
  return telegramToken;
}

@Override
public BotApiMethod<?> onWebhookUpdateReceived(Update update) {
  return null;
}

@Override
public String getBotPath() {
  return null;
}

@SneakyThrows
private InputFile toInputFile(InputStream inputStream, String filename) {
  return new InputFile(inputStream, filename);
}

@SneakyThrows
private InputMedia toInputPhotoMedia(MultipartFile multipartFile) {
  return toInputPhotoMedia(
      multipartFile.getInputStream(),
      multipartFile.getOriginalFilename()
  );
}

@SneakyThrows
private InputMedia toInputPhotoMedia(InputStream inputStream, String filename) {
```

```
    InputMediaPhoto inputMediaPhoto = new InputMediaPhoto();
    inputMediaPhoto.setMedia(inputStream, filename);
    return inputMediaPhoto;
  }

  private void handleUserBlocking(
      String externalChatId,
      TelegramApiRequestException telegramApiRequestException
  ) throws TelegramApiRequestException {
    if (telegramApiRequestException.getMessage().contains(BLOCKING_MESSAGE)) {
      log.info("user with externalChatId '{}' was deleted", externalChatId);
      telegramUserService.deleteByExternalChatId(externalChatId);
    } else {
      throw telegramApiRequestException;
    }
  }

  private void sendPhotoToSpecialChat(
      String caption,
      InputFile photo,
      String externalChatId,
      SendPhoto sendPhoto
  ) throws TelegramApiException {
    sendPhoto.setPhoto(photo);
    sendPhoto.setCaption(caption);
    sendPhoto.setChatId(externalChatId);
    try {
      execute(sendPhoto);
    } catch (TelegramApiRequestException telegramApiRequestException) {
      handleUserBlocking(externalChatId, telegramApiRequestException);
    }
  }

  private void sendFileToChat(
      String caption,
      InputFile file,
      String externalChatId,
      SendDocument sendDocument) throws IOException, TelegramApiException {
    sendDocument.setChatId(externalChatId);
    sendDocument.setDocument(file);
    sendDocument.setCaption(caption);
    try {
      execute(sendDocument);
    } catch (TelegramApiRequestException telegramApiRequestException) {
      handleUserBlocking(externalChatId, telegramApiRequestException);
    }
  }
}
```

itki-bot-api/src/main/java/com/itki/api/service/impl/TokenBlackListServiceImpl.java

```
package com.itki.api.service.impl;

import com.itki.api.service.TokenBlackListService;
import lombok.RequiredArgsConstructor;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.stereotype.Service;
import java.time.Duration;

@Service
@RequiredArgsConstructor
public class TokenBlackListServiceImpl implements TokenBlackListService {
  private final RedisTemplate<String, String> tokenBlackListCache;
  @Value("#{T(java.time.Duration).ofMillis('${JWT_TOKEN_TTL}')}")
  private Duration blackListAuthTokenTtl;
  @Value("#{T(java.time.Duration).ofDays('${REFRESH_TOKEN_TTL_IN_DAYS}')}")
  private Duration blackListRefreshTokenTtl;

  @Override
  public void putTokenToBlackList(String token) {
    tokenBlackListCache.opsForValue().set(token, null, blackListAuthTokenTtl);
  }

  @Override
  public void putRefreshTokenToBlackList(String token) {
    tokenBlackListCache.opsForValue().set(token, null, blackListRefreshTokenTtl);
  }

  @Override
  public boolean isTokenInBlackList(String token) {
    return Boolean.TRUE.equals(tokenBlackListCache.hasKey(token));
  }
}
```

itki-bot-api/src/main/java/com/itki/api/service/impl/UserServiceImpl.java

```
package com.itki.api.service.impl;

import com.itki.api.model.User;
import com.itki.api.repository.UserRepository;
import com.itki.api.service.UserService;
```

```
import java.time.Duration;
import java.util.Optional;
import lombok.extern.log4j.Log4j2;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.data.redis.core.ValueOperations;
import org.springframework.stereotype.Service;

@Service
@Log4j2
public class UserServiceImpl extends CrudServiceImpl<User> implements UserService {
  @Value("#{T(java.time.Duration).ofMillis('${JWT_TOKEN_TTL}').multipliedBy(5)}")
  private Duration userCacheExpiration;

  @Value("${REDIS_FEATURE}")
  private Boolean redisFeature;
  private final UserRepository userRepository;
  private final ValueOperations<String, User> valueOps;



  public UserServiceImpl(
      UserRepository userRepository,
      @Qualifier("userByLoginCache")
      RedisTemplate<String, User> userByLoginCache
  ) {
    super(userRepository, User.class.getSimpleName());
    this.userRepository = userRepository;
    this.valueOps = userByLoginCache.opsForValue();
  }

  @Override
  public Optional<User> findByLogin(String login) {
    if (redisFeature) {
      long startOperationTime = System.currentTimeMillis();
      Optional<User> byLoginWithCache = findByLoginWithCache(login);
      log.info(
          "findByLoginWithCache took {} ms",
          System.currentTimeMillis() - startOperationTime
      );
      return byLoginWithCache;
    }
    long startOperationTime = System.currentTimeMillis();
    Optional<User> userByLogin = userRepository.findUserByLogin(login);
    log.info(
        "findByLogin without cache took {} ms",
        System.currentTimeMillis() - startOperationTime
    );
    return userByLogin;
  }

  private Optional<User> findByLoginWithCache(String login) {
```

```
    Optional<User> user = Optional.ofNullable(valueOps.get(login));
    if (user.isEmpty()) {
      user = userRepository.findUserByLogin(login);
      user.ifPresent(u -> valueOps.set(login, u, userCacheExpiration));
    }
    return user;
  }
}
```

itki-bot-api/src/main/resources/application.properties

```
spring.datasource.url=jdbc:postgresql://localhost:5432/itki_db
spring.datasource.driverClassName=org.postgresql.Driver
spring.datasource.username=root
spring.datasource.password=12345
spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.hibernate.ddl-auto=validate
security.jwt.token.expire-length=3600000
security.jwt.token.secret-key=secret
spring.jpa.open-in-view=false

# remote debug
spring.devtools.remote.secret=itki
spring.devtools.restart.poll-interval=2s
spring.devtools.restart.quiet-period=1s

spring.servlet.multipart.max-file-size=50MB
spring.servlet.multipart.max-request-size=500MB
```

minio/Dockerfile

```
# Stage 1: Use an Alpine image to install the Minio Client
FROM alpine:3.12 as mc-builder

# Install wget and download Minio Client
RUN apk add --no-cache wget && \
    wget https://dl.min.io/client/mc/release/linux-amd64/mc && \
    chmod +x mc

# Stage 2: Use the official Minio image as the base image
FROM minio/minio

# Copy the Minio Client from the previous stage
COPY --from=mc-builder /mc /usr/bin/
```

```
# Copy the startup script into the container
COPY create-bucket.sh /usr/bin/

# Make the script executable
RUN chmod +x /usr/bin/create-bucket.sh

# Entry point to run the Minio server and create the bucket
ENTRYPOINT ["/usr/bin/create-bucket.sh"]
```

minio/create-bucket.sh

```
#!/bin/sh

# Start Minio server in the background
minio server /data --console-address ":9001" &

# Wait for Minio server to be up
while ! curl -s http://localhost:9000/minio/health/live; do
  echo "Waiting for Minio server..."
  sleep 3
done

# Configure Minio client with the Minio server
mc alias set myminio http://localhost:9000 $MINIO_ACCESS_KEY $MINIO_SECRET_KEY

# Create the bucket
mc mb myminio/my-data

# Keep the container running
tail -f /dev/null
```