НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:
В.о. завідувача кафедри
_____ Артем ВОЛОКИТА
(підпис)
“__” _____2026р.

Дипломний проєкт

на здобуття ступеня бакалавра
за освітньо-професійною програмою “Інженерія програмного забезпечення комп’ютерних систем”
спеціальності 121 “Інженерія програмного забезпечення”

на тему: Система автоматизації надсилання повідомлень у месенджери із онлайн-медіаплатформ

Виконав : студент 4 курсу, групи ІМ-21
(шифр групи)
_____Потаскалов Олесь Вадимович_____
(прізвище, ім’я, по батькові) (підпис)

Керівник _____доцент кафедри, к.т.н. Верба О. А _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант (нормоконтроль) асистент кафедри Нікольський С. С._____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент __доцент кафедри ТПЗА, к.т.н. Дунаєва Т. А._____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2026 р.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)

Освітньо-професійна програма

“Інженерія програмного забезпечення комп’ютерних систем”

спеціальності 121 “Інженерія програмного забезпечення”

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
_____ Артем ВОЛОКИТА
(підпис)
“__” _____2026р.

ЗАВДАННЯ
на бакалаврський дипломний проєкт студента

Потаскалова Олеся Вадимовича

1. Тема проєкту *Система автоматизації надсилання повідомлень у месенджери із онлайн-медіаплатформ*
керівник проєкту Верба Олександр Андрійович, доцент, к.т.н.,
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від __ травня 2026року №__-с

2. Термін здачі студентом закінченого проєкту *30 травня 2026р.*

3. Вихідні дані до проєкту *технічна документація, теоретичні дані.*

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

Розділ 1. Аналіз предметної області та існуючих рішень.

Розділ 2. Вимоги до системи та архітектурні рішення.

Розділ 3. Проектування та реалізація системи.

Розділ 4. Аналіз та демонстрація розробленої системи.

5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень) діаграма потоку обробки Twitch-подій, ER-діаграма сутностей бази даних, діаграма потоку обробки Twitch-подій.

6. Консультанта проєкту, з вказівкою розділів проєкту, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Нікольський С. С.		

7. Дата видачі завдання «24» вересня 2025 р.

Календарний план

№ п/п	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту	Примітки
1.	*Затвердження теми проєкту*	*08.09.2025-24.09.2025*	
2.	*Вивчення та аналіз завдання*	*24.09.2025-30.03.2026*	
3.	*Розробка архітектури та загальної структури системи*	*30.03.2026-14.04.2026*	
4.	*Розробка структур окремих підсистем*	*14.04.2026-04.05.2026*	
5.	*Програмна реалізація системи*	*27.05.2026-05.06.2026*	
6.	*Оформлення пояснювальної записки*	*11.05.2026-06.06.2026*	
7.	*Захист програмного продукту*	05.06.2026	
8.	*Передзахист*	08.06.2026	
9.	*Захист*	18.06.2026	

Студент-дипломник _____ Олесь ПОТАСКАЛОВ
(підпис)

Керівник проєкту _____ Олександр ВЕРБА
(підпис)

АНОТАЦІЯ

У бакалаврській роботі проаналізовано існуючі системи автоматизованого надсилання повідомлень з онлайн-медіаплатформ у месенджери. На основі проведеного аналізу та виявлених потреб кінцевих користувачів сформульовано спільні недоліки наявних рішень. Отримані результати стали підґрунтям для визначення функціональних та нефункціональних вимог до власної системи автоматизації сповіщень.

У результаті розроблено програмний продукт, що автоматизує процес сповіщень і забезпечує ефективну комунікацію між творцями медіа-контенту та їхньою аудиторією. Система реалізована мовою TypeScript на базі фреймворку NestJS 11, як основу для мікросервісної аріхтектури для BackEnd складової проєкту, і організована як монорепозиторій з чотирьох мікросервісів: API, Webhook Receiver, Notification Worker та Discord Bot. Разом з цим, веб-інтерфес, розроблений на основі NextJS 16. Для зберігання даних використовується СУБД PostgreSQL 16 з TypeORM; кешування та дедуплікація реалізовані засобами Redis 7; міжсервісний обмін повідомленнями забезпечує RabbitMQ 3; для синхронної внутрішньої комунікації між сервісами у форматі запит-відповідь застосовано TCP RPC. Розгортання системи здійснюється засобами Docker Compose.

Ключові слова: автоматизація сповіщень, мікросервісна архітектура, Twitch EventSub, Discord, webhook, RabbitMQ, Redis, TCP RPC, NestJS, TypeScript, PostgreSQL.

ANNOTATION

This bachelor's thesis analyses existing systems for automated notification delivery from online media platforms to messengers. Based on the analysis conducted and the identified needs of end users, the common shortcomings of available solutions were established. These findings formed the basis for defining the functional and non-functional requirements for a purpose-built notification automation system.

As a result, a software product was developed that automates the notification process and enables efficient communication between content creators and their audiences. The system is implemented in TypeScript using the NestJS 11 framework and is structured as a monorepo comprising three microservices: API, Webhook Receiver, and Discord Bot. PostgreSQL 16 with TypeORM is used for data persistence; caching and deduplication are handled by Redis 7; inter-service message publishing is managed by RabbitMQ 3; and TCP RPC is applied for synchronous request-response communication between services. The system is deployed using Docker Compose.

Key words: notification automation, microservices architecture, Twitch EventSub, Discord, webhook, RabbitMQ, Redis, TCP RPC, NestJS, TypeScript, PostgreSQL

справки	Формат	*Значення*	*Найменування*	Кіл. листів	№ екземпля	*Додаток*
			Документація загальна			
			Знову розроблена			
	А4	***ІАЛЦ.467200.002 ТЗ***	Система автоматизації	4		
			надсилання повідомлень у			
			месенджери із онлайн-медіаплатформ			
			Технічне завдання			
	А4	***ІАЛЦ.467200.003 ПЗ***	Система автоматизації	109		
			надсилання повідомлень у			
			месенджери із онлайн-медіаплатформ			
			Пояснювальна записка			
	А4	***ІАЛЦ.467200.004 Д1***	Система автоматизації	1		
			надсилання повідомлень у			
			месенджери із онлайн-медіаплатформ			
			Діаграма компонентів та			
			взаємодії підсистем			
	А4	***ІАЛЦ.4672008.005 Д2***	Система автоматизації	1		
			надсилання повідомлень у			
			месенджери із онлайн-медіаплатформ			
			ER-діаграма сутностей			
			бази даних			
	А4	***ІАЛЦ.467200.006 Д3***	Система автоматизації	1		
			надсилання повідомлень у			
			месенджери із онлайн-медіаплатформ			
			Діаграма потоку обробки			
			Twitch-подій			
	А4	***ІАЛЦ.467200.007 Д4***	Система автоматизації	38		
			надсилання повідомлень у			
			месенджери із онлайн-медіаплатформ			
			Текст програмного коду			

					ІАЛЦ.467200.001 ОА
Зм	***Лист***	***№ докум.***	***Підп***	***Дата***	
Розроб		Потаскалов О. В.			*Система автоматизації надсилання повідомлень у месенджери із онлайн-медіаплатформ* Опис альбому
Перев		Верба О.А.			

Літ.	Аркуш	Аркушів
	1	1

НТУУ "КПІ" ФІОТ ІМ-21

ТЕХНІЧНЕ ЗАВДАННЯ
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Система автоматизації надсилання повідомлень у месенджери із онлайн-медіаплатформ*»

Київ – 2026

ЗМІСТ

1 НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання поширюється на розробку системи автоматизації надсилання повідомлень у месенджери з онлайн-медіаплатформ, а також на подальшу підтримку та вдосконалення розробленої системи.

Областю застосування системи є адміністрування спільнот у месенджерах та чат-платформах, зокрема Discord, де виникає потреба в оперативному інформуванні учасників про події на стримінгових платформах. Система орієнтована на використання творцями медіа-контенту, адміністраторами серверів та організаторами онлайн-спільнот, які прагнуть автоматизувати процес сповіщень без ручного дублювання оголошень.

2 ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки даної системи є завдання для виконання роботи кваліфікаційно-освітнього рівня «бакалавр інженерії програмного забезпечення», який був затверджений факультетом “Інформатики та обчислювальної техніки” кафедрою обчислювальної техніки Національного технічного Університету України «Київський Політехнічний інститут ім. Ігоря Сікорського».

3 МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою даної роботи є проектування та реалізація системи автоматизованого надсилання подієвих сповіщень з онлайн-медіаплатформ у месенджери на основі мікросервісної архітектури з використанням механізму push-сповіщень.

Призначенням системи є усунення необхідності ручного інформування аудиторії шляхом автоматичної доставки повідомлень у реальному часі при настанні заданих подій на платформі – таких як початок або завершення трансляції, зміна категорії чи рейд. Система забезпечує гнучке управління

підписками через кредитну модель та підтримує розширення на нові платформи-джерела і платформи-одержувачі без зміни існуючої кодової бази

4 ДЖЕРЕЛА РОЗРОБКИ

Джерелами розробки даного дипломного проекту є офіційна документація Twitch EventSub API та Discord Developer API, технічна документація використаних фреймворків і бібліотек (NestJS, TypeORM, RabbitMQ, Redis, Zod), науково-технічна література з проектування мікросервісних систем, а також публікації та статті у мережі Інтернет, присвячені архітектурним паттернам побудови розподілених систем доставки повідомлень у реальному часі

5 ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розробленого продукту

Розроблена система має виконувати такі вимоги:

- Забезпечити реєстрацію та автентифікацію користувачів через електронну пошту або Discord OAuth 2.0.
- Надати можливість користувачам створювати, переглядати та видаляти підписки на події онлайн-медіаплатформ через REST API та Discord slash-команди.
- Забезпечити автоматичне надсилання сповіщень у Discord-канали при настанні підписаних подій у реальному часі, без використання механізму polling.
- Надати можливість користувачам самостійно налаштовувати вміст повідомлення (текст, embed, колір) для кожного призначення окремо.
- Підтримувати два механізми доставки повідомлень: через Discord Bot та через Discord Incoming Webhook.
- Реалізувати кредитну модель управління підписками з перевіркою ліміту перед додаванням нового призначення.

5.2. Вимоги до програмного забезпечення

- Операційна система: Windows 10/11, macOS 12 або вище, Ubuntu 22.04 LTS або інший дистрибутив Linux з підтримкою Docker.
- Docker Engine версії 24.0 або вище та Docker Compose версії 2.20 або вище.
- Node.js версії 20 LTS або вище (для локальної розробки без контейнеризації).
- Доступ до мережі Інтернет для взаємодії з Twitch API та Discord API.
- Публічно доступна IP-адреса або домен з підтримкою HTTPS для реєстрації Twitch EventSub webhook-підписок.

5.3. Вимоги до апаратної частини

- ЦП не менше ніж Intel® Core i3 8-го покоління або AMD Ryzen 3 3000-ї серії (2 фізичні ядра, тактова частота від 2.0 ГГц).
- Оперативна пам'ять (RAM) не менше ніж 4 ГБ.
- Постійна пам'ять (ROM) не менше ніж 20 ГБ вільного дискового простору.
- Мережевий інтерфейс зі стабільним з'єднанням не менше ніж 10 Мбіт/с.

6 ЕТАПИ РОЗРОБКИ

Назва етапів виконання	Термін виконання
Затвердження теми роботи	08.09.2025-24.09.2025
Вивчення та аналіз завдання	24.09.2025-30.03.2026
Розробка архітектури та загальної структури системи	30.03.2026-14.04.2026
Розробка структур окремих частин системи	14.04.2026-04.05.2026
Програмна реалізація системи	27.04.2026-05.06.2026
Виправлення помилок	07.06.2026-10.06.2026
Оформлення пояснювальної записки	11.05.2026-06.06.2026

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Система автоматизації надсилання повідомлень у месенджери із онлайн-медіаплатформ*»

Київ – 2026

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

AMQP	Advanced Message Queuing Protocol (Розширений протокол черги повідомлень)
API	Application Programming Interface (Прикладний програмний інтерфейс)
CI/CD	Continuous Integration / Continuous Deployment (Безперервна інтеграція та безперервне розгортання)
DI	Dependency Injection (Впровадження залежностей)
DTO	Data Transfer Object (Об'єкт передачі даних)
ENV	Environment Variables (Змінні середовища)
FK	Foreign Key (Зовнішній ключ)
HMAC	Hash-based Message Authentication Code (Код автентифікації повідомлень на основі хешу)
HTTP	HyperText Transfer Protocol (Протокол передачі гіпертексту)
HTTPS	HyperText Transfer Protocol Secure (Захищений протокол передачі гіпертексту)
JWT	JSON Web Token (Токен на основі формату JSON)
OAuth	Open Authorization (Відкритий протокол авторизації)
ORM	Object-Relational Mapping (Об'єктно-реляційне відображення)
PK	Primary Key (Первинний ключ)
REST	Representational State Transfer (Передача репрезентативного стану)
RPC	Remote Procedure Call (Виклик віддаленої процедури)
SHA	Secure Hash Algorithm (Алгоритм безпечного хешування)
TCP	Transmission Control Protocol (Протокол керування передачею)
TTL	Time To Live (Час життя запису)

URL	Uniform Resource Locator (Уніфікований вказівник ресурсу)
UUID	Universally Unique Identifier (Universally Unique Identifier – універсальний унікальний ідентифікатор)
СУБД	Система управління базами даних
JRPC	JSON Remote Procedure Call (Різновид RPC, в якому серіалізація даних запиту та відповіді здійснюється у форматі JSON; використовується для синхронної внутрішньої комунікації між сервісами у форматі запит-відповідь)
ACID	Atomicity, Consistency, Isolation, Durability (Атомарність, узгодженість, ізольованість, довговічність)
ARIA	Accessible Rich Internet Applications (Доступні насичені інтернет-застосунки)
WCAG	Web Content Accessibility Guidelines (Настанови з доступності веб-контенту)
UI	User Interface (Користувацький інтерфейс)
UX	User Experience (Користувацький досвід)
IoC	Inversion of Control (Інверсія керування)
SSR	Server-Side Rendering (Серверний рендеринг)
SPA	Single Page Application (Односторінковий застосунок)
CSS	Cascading Style Sheets (Каскадні таблиці стилів)
SCSS	Sassy CSS (Розширений синтаксис CSS)
SQL	Structured Query Language (Мова структурованих запитів)
SSL	Secure Sockets Layer (Рівень захищених з'єднань)
IDE	Integrated Development Environment (Інтегроване середовище розробки)
JSON	JavaScript Object Notation (Текстовий формат обміну даними)
SLA	Service Level Agreement (Угода про рівень обслуговування)
LTS	Long-Term Support (Довгострокова підтримка)

AOT	Ahead-of-Time compilation (Компіляція до виконання)
MVC	Model-View-Controller (Модель-вигляд-контролер)
RSC	React Server Components (Серверні компоненти React)
SWR	Stale-While-Revalidate (Застарілий-під-час-ревалідації)
CORS	Cross-Origin Resource Sharing (Спільне використання ресурсів між різними джерелами)

ВСТУП

З поширенням стримінгових платформ та зростанням аудиторії контент-мейкерів виникає потреба в ефективних інструментах для сповіщення спільноти про нові трансляції та події. Месенджери та чат-платформи, зокрема Discord, стали основним середовищем комунікації для геймерів і стримерів, проте ручне інформування підписників залишається трудомістким процесом.

Актуальність теми визначається відсутністю гнучкого, відкритого та масштабованого рішення для автоматизованого надсилання сповіщень про події на онлайн-медіаплатформах у месенджери. Існуючі комерційні сервіси пропонують лише фіксовані тарифні плани без гнучкого управління лімітами, не надають доступу до API та не підтримують розширення на нові платформи силами спільноти.

Метою дипломної роботи є проектування та реалізація системи автоматизації надсилання повідомлень у месенджери з онлайн-медіаплатформ на основі мікросервісної архітектури з використанням асинхронних черг повідомлень.

Для досягнення мети визначено такі завдання:

- провести аналіз предметної області та існуючих систем-аналогів;
- сформувати функціональні та нефункціональні вимоги до системи;
- обгрунтувати архітектурні рішення та вибір технологічного стеку;
- спроектувати та реалізувати мікросервісну систему відповідно до сформованих вимог;
- провести тестування системи та оцінити її якість.

Предметом дослідження є методи, архітектурні паттерни та інструменти для побудови надійних, масштабованих систем доставки повідомлень у реальному часі.

Практична цінність роботи полягає у розробці повністю функціональної системи, що реалізує: обробку Twitch EventSub webhook-подій без polling,

гнучку кредитну модель управління підписками, ізольовану мікросервісну архітектуру з RabbitMQ, а також підтримку як Bot, так і Webhook-режимів доставки у Discord.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ІСНУЮЧИХ РІШЕНЬ

1.1 Огляд предметної області

Онлайн-стримінг як індустрія демонструє стрімке зростання: за даними аналітичних компаній, Twitch щоденно охоплює понад 30 мільйонів активних глядачів [4]. Контент-мейкери формують лояльну аудиторію через соціальні мережі та тематичні спільноти, основу яких у сфері геймingy складає Discord – платформа з понад 600 мільйонами зареєстрованих користувачів [5].

Ключовою проблемою є розрив між подією на стримінговій платформі (початок трансляції, зміна категорії, рейд тощо) та своєчасним інформуванням підписників у месенджері. Ручне дублювання оголошень є неефективним, а нерегулярні сповіщення знижують залученість аудиторії та конверсію переглядів.

Технічно задача автоматизованого сповіщення реалізується через два підходи:

- Polling – система з фіксованим інтервалом запитує платформу про зміни стану стримера. Метод простий у реалізації, але має затримку (залежить від інтервалу) та генерує надлишкове навантаження на API, разом з цим має обмеження на кількість зазначених для запиту каналів для polling -у, що створює потребу у розділені запитів на декілька, разом з цим збільшує потребу у кількості викликів, що у результаті призведе до технічного обмеження зі сторони сервісу, у нашому випадку Twitch.
- Webhook / Push-сповіщення – платформа сама надсилає HTTP-запит на зареєстрований URL при настанні події. Метод забезпечує мінімальну затримку та знижує навантаження, проте вимагає публічно доступного сервера та коректної обробки підписок. Звісно разом з цим ми

зіштовхаємося з іншим обмеженням, яке полягає у тому, що на один зареєстрований акаунт сервісу Twitch – ми маємо обмеження у кількість підписок у рамках вартості кожного залученої події, яка підписана на Push-сповіщення. Вартість таких підписок може бути знижена, за рахунок авторизації власників каналу за допомогою Twitch OAuth 2.0 через відповідно вказаний акаунт платформи.

Twitch реалізує Webhook підхід через механізм EventSub – уніфіковану систему підписок на події з верифікацією через HMAC-SHA256 підпис [1]. Саме EventSub є основою для побудови надійних push-систем сповіщення.

Discord надає два механізми доставки повідомлень від зовнішніх сервісів: Discord Bot API (відправка через бот у конкретний канал сервера) та Incoming Webhooks (відправка через заздалегідь сконфігурований webhook-URL без необхідності присутності бота на сервері) [3].

1.2 Аналіз існуючих систем-аналогів

На сьогодні існують сервіси, що вирішують задачу автоматизованого надсилання повідомлень за подіями з медіаплатформ у месенджери, зокрема Discord. Попри відмінності у функціональності та моделях доступу, основний принцип роботи залишається схожим: система відстежує задану подію та доставляє сповіщення у визначений канал. Переважна більшість таких сервісів надає обмежену кількість безкоштовних підписок, а їхній функціонал здебільшого зводиться до відправки простих текстових повідомлень. Це звужує можливості для побудови повноцінної взаємодії між контент-мейкером та аудиторією – зокрема, унеможливлює реакцію на специфічні події (рейди, підписки, донати) та гнучке налаштування формату сповіщень.

1.2.1 Streamcord

Streamcord – один з найпопулярніших Discord-ботів для Twitch-сповіщень, якому довіряють понад мільйон серверів [6]. Сервіс пропонує веб-панель для налаштування, підтримує Twitch та YouTube, а також функцію Live Role – автоматичне присвоєння ролі учасникам сервера під час активної трансляції.

Перевагами системи є чиста веб-панель управління, швидка доставка сповіщень на базі офіційного Twitch API, фільтрація за назвою гри та трансляції (у Pro-версії), а також синхронізація розкладу стримерів з Discord-подіями.

Серед недоліків – жорсткий ліміт у 10 стримерів на безкоштовному тарифі (5 Twitch + 5 YouTube), відсутність публічного API для зовнішніх інтеграцій, неможливість самостійного хостингу та закритий вихідний код.

1.2.2 MEE6 Social Alerts

MEE6 – один з найпоширеніших мультифункціональних Discord-ботів з аудиторією понад 21 мільйон серверів [7]. Функція Social Alerts входить до ширшого набору модулів: модерація, система рівнів, вітальні повідомлення, внутрішня економіка тощо. Серед аналогів MEE6 підтримує найбільший перелік платформ – Twitch, YouTube, Kick, X (Twitter), TikTok, Reddit, Instagram, Bluesky, RSS.

Попри широке охоплення, сервіс має суттєві обмеження для стримерів та адміністраторів спільнот: сповіщення повністю відсутні у безкоштовному плані, а преміум-підписка вартістю $11.95 на місяць є найдорожчою серед розглянутих аналогів [7]. Оскільки Social Alerts є лише одним із десятків модулів, інтерфейс не оптимізовано під потреби стримерів. Відсутні специфічні для стримінгу функції – фільтрація за категорією, аналітика доставки та режим «тихих годин».

1.2.3 Streamlinx

Streamlinx – відносно новий спеціалізований сервіс, що позиціонує себе як мультиплатформений конкурент Streamcord. Підтримує Twitch, YouTube та Kick; безкоштовний рівень є найщедрішим серед аналогів – до 15 стримерів з увімкненою функцією Live Role без додаткової оплати [8].

Перевагами є повноцінна веб-панель, підтримка сповіщень про кліпи та нові відео (upload notifications), швидка доставка. Серед обмежень – відсутність публічного API, закритий вихідний код, а також відносна молодість платформи: сервіс не має верифікованої статистики надійності та задокументованого SLA.

1.2.4. Sx Live

Sx Live є компонентом мультифункціонального бота Sx Bot і підтримує Twitch, YouTube та Kick, позиціонуючись як спеціалізоване рішення для Kick-сповіщень. Безкоштовний рівень суттєво обмежений; повний функціонал – необмежені сповіщення, Live Roles, пріоритетна доставка – доступний за $3.99 на місяць або $29.99 на рік [9].

Мультифункціональність бота (модерація, Spotify Roles, реакція-ролі) може бути зручною для серверів із широким набором потреб, проте є недоліком для тих, хто шукає спеціалізоване рішення з мінімальними залежностями. Публічний API та можливість самостійного розгортання відсутні.

1.2.5 MonitoRSS та Pingcord

MonitoRSS – відкрита система на базі RSS, що дозволяє відстежувати практично будь-яку платформу за наявності відповідної RSS-стрічки. У режимі самостійного хостингу є повністю безкоштовною, що робить її привабливою для технічно підготовлених користувачів. Проте RSS є опосередкованим механізмом: затримка сповіщення може сягати кількох хвилин, а для

стримінгових подій потрібно самостійно знаходити або формувати відповідні стрічки, що не завжди можливо.

Pingcord підтримує широкий спектр платформ (TikTok, Instagram, YouTube, Twitch, Reddit, Facebook), однак має обмеження у безкоштовному плані та частково незавершений функціонал – зокрема, підтримка власних RSS-стрічок позначена як «coming soon». Обидві системи позбавлені механізму гнучкого управління лімітами на рівні окремих подій [10].

1.3 Порівняльний аналіз та обґрунтування актуальності розробки

У таблиці 1.1 наведено порівняльний аналіз розглянутих систем за ключовими критеріями.

Таблиця 1.1 – Порівняння систем-аналогів

Система	**Платформи**	**Безкоштовний ліміт**	**Ціна Pro**	**Відкритий код**
Streamcord	Twitch, YouTube	10 стримерів	$2.99/міс	Ні
MEE6 Social Alerts	Twitch, YouTube, Kick, X, TikTok, Reddit...	Немає (paid only)	$11.95/міс	Ні
Streamlinx	Twitch, YouTube, Kick	15 стримерів	$2.99/міс	Ні
Sx Live	Twitch, YouTube, Kick	Обмежено	$3.99/міс	Ні
MonitoRSS	RSS (будь-яка платформа)	Необмежено	Self-hosted	Так
розроблена система	Twitch (YouTube у планах)	Кредитна система	—	Так

Аналіз таблиці 1.1 дозволяє виявити такі ключові недоліки існуючих рішень:

– Жорсткі тарифні плани: усі комерційні системи обмежують кількість стримерів або функцій фіксованими пакетами без можливості гнучкого налаштування.

– Відсутність публічного API: жоден з аналогів не надає API для програмної інтеграції, що виключає автоматизацію управління підписками.

– Закритий вихідний код: неможливість аудиту безпеки, розширення функціоналу або самостійного хостингу.

Розроблювана система спрямована на усунення зазначених недоліків через: відкритий вихідний код з можливістю самостійного розгортання; гнучку кредитну модель замість фіксованих тарифних планів; публічний REST API та підтримку Discord slash-команд; мікросервісну архітектуру для незалежного масштабування компонентів; використання Twitch EventSub (push-модель, без polling) для мінімальної затримки доставки.

Окремою відмінністю від існуючих аналогів є підхід до інтеграції стримерів як повноправних користувачів системи. Наявні рішення орієнтовані переважно на адміністраторів серверів і не передбачають механізмів, що безпосередньо залучають контент-мейкерів до налаштування взаємодії з власною аудиторією. Надання стримерам інструментів для гнучкого управління сповіщеннями здатне суттєво розширити цільову аудиторію системи та підвищити її практичну цінність.

Ключовою особливістю бізнес-логіки системи є відмова від обов'язкових платних планів як умови створення сповіщень. Монетизація передбачається виключно на добровільній основі – через підтримку від користувачів або рекламні інтеграції як у самій системі, так і в шаблонах сповіщень. Такий підхід знижує поріг входу для нових користувачів і відповідає принципам відкритого програмного забезпечення.

З метою стимулювання органічного поширення системи розглядається механізм розширеного кредитного ліміту для адміністраторів великих спільнот, а також можливість адаптації функціоналу на основі прямого зворотного зв'язку від цільової аудиторії.

ВИСНОВОК ДО РОЗДІЛУ

У першому розділі проведено аналіз предметної області систем автоматизованих сповіщень з онлайн-медіаплатформ у месенджери. Розгляд п'яти систем-аналогів – Streamcord, MEE6 Social Alerts, Streamlinx, Sx Live та MonitoRSS – дозволив виявити спільні обмеження наявних рішень: жорсткі фіксовані тарифні плани, відсутність публічного API, закритий вихідний код та недостатню орієнтованість на потреби контент-мейкерів як окремої цільової групи. Зокрема, жоден з розглянутих аналогів не надає публічного API для зовнішніх інтеграцій, не підтримує самостійного розгортання та не пропонує гнучкого управління лімітами на рівні окремих підписок.

На основі отриманих результатів сформульовано функціональні та нефункціональні вимоги до розроблюваної системи, які охоплюють підтримку Twitch EventSub як механізму push-доставки, два режими відправки повідомлень у Discord (Bot та Webhook), а також кредитну модель управління підписками.

Кредитна модель передбачає диференційовану вартість підписок залежно від статусу стримера в системі: підписка на стримера, зареєстрованого в системі, коштує 0.5 кредиту, тоді як підписка на стримера поза системою – 1 кредит. Таким чином створюється стимул для контент-мейкерів реєструватися у системі самостійно: зареєстрований стример отримує доступ до розширеного набору персональних подій та додаткових опцій налаштування сповіщень, недоступних для зовнішніх підписок. Окремо передбачено механізм надання додаткових кредитів серверам Discord – адміністратори активних спільнот можуть отримати розширений ліміт підписок, що стимулює органічне поширення системи серед великих аудиторій і забезпечує додаткові ресурси для розвитку платформи.

Проведений аналіз підтверджує, що існуючі рішення не повною мірою задовольняють потреби власників спільнот та контент-мейкерів, що обумовлює доцільність розробки власної системи. Виявлені прогалини формують чітку конкурентну нішу, а запропонована кредитна модель та відкрита архітектура визначають напрями, за якими розроблювана система пропонує якісно відмінний підхід порівняно з наявними аналогами.

РОЗДІЛ 2. ВИМОГИ ДО СИСТЕМИ ТА АРХІТЕКТУРНІ РІШЕННЯ

2.1 Функціональні вимоги

На основі аналізу предметної області та виявлених обмежень існуючих систем-аналогів сформульовано функціональні вимоги до розроблюваної системи. Вимоги згруповано за функціональними підсистемами відповідно до їхнього призначення.

2.1.1 Автентифікації та авторизації

Підсистема автентифікації та авторизації є точкою входу до системи та визначає права доступу кожного учасника. Система має підтримувати три незалежних механізми реєстрації та входу: через електронну пошту з паролем, через Discord OAuth 2.0 та через Twitch OAuth 2.0, що дозволяє охопити як технічно підготовлених користувачів, користувачів які мають на меті бути партнерами або ж отримувати відгуки з платформи, так і аудиторію, для якої Discord є основним середовищем комунікації. Після успішної автентифікації система видає пару токенів – короткостроковий access-токен та довгостроковий refresh-токен у форматі JWT. Для забезпечення можливості примусового завершення сесій реалізується механізм інвалідації токенів через зберігання хешу рефреш токена, та надання йому стану, з майбутнім видаленням, коли час існування токена вже завершується. Розмежування доступу здійснюється на рівні ролей: роль користувача надає доступ до управління власними підписками та призначеннями, або ж перегляд власної статистки, як контент-мейкера, тоді як роль адміністратора розширює можливості до управління Twitch-додатками та перегляду стану всіх підписок у системі. Разом з цим, у системі має бути

закладена логіка для розширення дозволів користувачам, з підв’язкою до конкретних спільнот, з урахуванням створення певної ролі, що буде відповідати партнерським взаємовідносинам з платформою, з метою спрощення лімітів на використанням певного функціоналу та покращення якості, яку надає сервіс.

Вмикання та вимикання методів реєстрації здійснюється через конфігураційні прапорці (feature flags) без необхідності повторного розгортання системи. У той час як, рольова система базується на словнику, який може бути відредагованим користувачами, які мають найвищі права ролі для використання сервісу.

2.1.2 Управління Twitch-підписками

Підсистема управління Twitch-підписками реалізує основну бізнес-логіку взаємодії з платформою-джерелом подій. Система має забезпечувати реєстрацію та деактивацію EventSub-підписок через офіційний Twitch API, підтримуючи повний перелік релевантних типів подій: початок та завершення трансляції (stream.online, stream.offline), оновлення інформації про канал (channel.update), рейди (channel.raid), нові підписники (channel.subscribe) та донати (channel.cheer). Прийом вхідних webhook-запитів від Twitch здійснюється через окремий ізольований сервіс з обов'язковою верифікацією HMAC-SHA256 підпису кожного повідомлення, що унеможливлює обробку підроблених запитів.

З метою захисту від повторної обробки одного й того самого повідомлення – що є штатною поведінкою Twitch при мережевих збоях – реалізується дедуплікація подій через Redis з TTL-ключами тривалістю 10 хвилин, що відповідає гарантованому вікну повторних доставок платформи.

2.1.3 Управління призначеннями сповіщень

Підсистема управління призначеннями сповіщень забезпечує зв'язок між подіями платформи та каналами доставки. Користувач має можливість створювати, переглядати та видаляти Discord-призначення двома способами: через REST API з веб-інтерфейсу та через Discord slash-команди безпосередньо в месенджері.

Підтримується два механізми доставки повідомлень: через Discord Bot із зазначенням ідентифікатора каналу та сервера, а також через Discord Incoming Webhook із зазначенням webhook-URL, що дозволяє налаштовувати сповіщення без необхідності надавати боту права на сервер. Для кожного призначення окремо налаштовується вміст повідомлення – текст, embed-блок та колір, – що дозволяє адаптувати формат сповіщення під стиль конкретної спільноти. Разом з цим, додатково можуть бути налаштовані параметри фільтрації доставки повідомлень, та підхід взаємодії, який буде досягатися за рахунок того, що ми будемо мати параметри для Disocrd повідомлень, які будуть включати не тільки embed повідомлення, а й компоненти взаємодії (кнопки, випадаючи списки тощо). Ці компоненти взаємодії, дозволять створювати додаткові дії пов’язані з користувацькою аудиторію, та власниками спільнот, або ж власниками відповідних сповіщень.

2.1.4 Кредити

Кредитна підсистема реалізує гнучку модель управління підписками на сповіщення, що є принциповою відмінністю від фіксованих тарифних планів аналогів. Основною одиницею розрахунку є кредит – внутрішня валюта системи, яка не має грошового еквівалента та не може бути придбана за кошти.

Баланс кредитів може належати двом суб'єктам системи незалежно один від одного: окремому користувачу або Discord-серверу (спільноті). Кожне призначення сповіщення коштує 1 кредит, і при його створенні визначається, з

якого балансу буде списана вартість – з балансу користувача або з балансу сервера. Це дозволяє адміністраторам спільноти централізовано покривати витрати на сповіщення для всіх учасників або делегувати оплату безпосередньо користувачам. Вартість призначення фіксується на момент створення та не підлягає ретроактивному перерахунку.

Окремим типом є кредитна підписка, доступна у разі, якщо стример є партнером системи – тобто він самостійно зареєструвався та авторизувався через Twitch OAuth 2.0, надавши системі доступ до свого акаунту. У такому разі підписка на події цього стримера не списує кредити з жодного балансу. Натомість підписка підтримується виключно за рахунок токенів самого стримера та залишається активною доти, доки він перебуває в системі. Якщо стример відкликає доступ, виходить із системи або його токени більше не можуть бути оновлені – система автоматично деактивує всі пов'язані сповіщення, що використовують цей тип підписки, без жодних дій з боку адміністратора.

Перевірка достатності балансу виконується перед кожним додаванням нового призначення. Система підтримує механізм поповнення балансу серверів, що стимулює розвиток активних спільнот та органічне поширення платформи.

2.1.5 Веб-інтерфейс

Дана підсистема несе відповідальність за реалізацію користувацького інтерфейсу, в якому зосереджено всі основні точки взаємодії користувача з системою.

Веб-інтерфейс має задовольняти потреби, як звичайних користувачів, так і адміністраторів платформи. Тобто в рамках єдиного рішення має бути зосереджений різний функціонал, доступність якого визначається правами конкретного користувача у системі. Таким чином, інтерфейс одночасно

виступає точкою входу для адміністративного управління – додавання Twitch-додатків, перегляд статистики, лімітів, користувацького зворотного зв'язку тощо – та інструментом управління спільнотами й підписками з боку звичайного користувача. Для контент-мейкерів веб-панель також слугує зручним центром агрегації інформації про активність підписок та зворотний зв'язок від аудиторії щодо трансляцій.

Окремою ключовою потребою, яку закриває веб-інтерфейс, є коректна реалізація процесу авторизації. Інтеграція з платформами Discord та Twitch здійснюється через OAuth 2.0 Authorization Code Flow, що технічно вимагає наявності публічно доступного callback-ендпоінту для обробки коду авторизації. Крім того, з точки зору користувацького досвіду, важливо надати користувачу візуальний супровід процесу авторизації – відображення стану обробки, індикатор завантаження та підтвердження успішного входу. Альтернативний підхід, за якого авторизація відбувається виключно через REST API з поверненням JSON-відповіді, є технічно функціональним, проте позбавленим будь-якого UX-контексту і неприйнятним як основний сценарій взаємодії.

Окрім функціонального зосередження платформи у веб-інтерфейсі, важливим є візуальне представлення системи через landing-сторінку, що пояснює призначення та можливості продукту для нових відвідувачів. Крім цього, інтерфейс виступає централізованим місцем для розміщення офіційних документів, необхідних для публічного використання платформи: політики конфіденційності, контактної форми та інших матеріалів регуляторного характеру.

2.2 Нефункціональні вимоги

Нефункціональні вимоги визначають якісні характеристики системи та обмеження, за яких вона має функціонувати незалежно від конкретного сценарію використання.

2.2.1 Продуктивність

Час від моменту отримання webhook-події від Twitch до доставки повідомлення у Discord-канал не повинен перевищувати 15 секунд за нормальних умов навантаження. Ця вимога обумовлена характером використання системи: сповіщення про початок трансляції є чутливими до затримки, оскільки аудиторія очікує на оперативне інформування. Досягнення зазначеного показника забезпечується використанням push-механізму EventSub замість polling та асинхронної обробки подій через чергу повідомлень.

2.2.2 Надійність

Система має гарантувати доставку сповіщень навіть за умови тимчасової недоступності окремих компонентів. Зокрема, у разі перезапуску або збою Discord-бота повідомлення не втрачаються, а зберігаються в черзі RabbitMQ до відновлення сервісу. Аналогічно, тимчасова недоступність сервісу-приймача не впливає на отримання та збереження вхідних webhook-подій від Twitch.

2.2.3 Масштабованість

Архітектура системи має підтримувати горизонтальне масштабування кожного мікросервісу незалежно від інших. Усі сервіси є stateless – власний стан не зберігається в пам'яті процесу, а зовнішнє сховище (PostgreSQL та Redis) є єдиним джерелом істини. Це дозволяє запускати довільну кількість екземплярів кожного сервісу без змін у коді або конфігурації.

2.2.4 Безпека

Усі вхідні webhook-запити від Twitch верифікуються через HMAC-SHA256 підпис до будь-якої подальшої обробки. JWT access-токени мають обмежений термін дії; компрометовані токени інвалідуються через застосування відповідних флагів до сутностей у таблиці, разом з цим з використанням Redis-blacklist, як спосіб спрощеної перевірки з часом життя, у рамках використання Cache сервісу. Кожен мікросервіс взаємодіє з RabbitMQ через окремого користувача з мінімально необхідними правами відповідно до принципу найменших привілеїв.

2.2.5 Безпека клієнтського стану

Дані користувачів, що передаються системою у веб-інтерфейс або повертаються у відповідь на запити від Discord slash-команд, мають відповідати безпековій політиці системи з урахуванням ризиків витоку інформації на клієнтській стороні.

У контексті веб-інтерфейсу системою має надаватися виключно короткостроковий access token власної системи, та відповідний refresh token – без передачі токенів сторонніх платформ (Discord, Twitch), до яких користувач підключив свій акаунт в межах системи. Токени зовнішніх платформ є внутрішніми обліковими даними системи та не підлягають передачі на клієнт за жодних обставин.

Оскільки refresh token зберігається на клієнтській стороні (у localStorage), система має реалізовувати механізм його примусової інвалідації на сервері. Це забезпечує можливість захисту акаунту користувача у разі компрометації клієнтських даних: інвалідований refresh token унеможливлює отримання нового access token навіть за наявності вкраденого значення.

Взаємодія між клієнтом та сервером має здійснюватися виключно через захищений транспортний протокол HTTPS, що забезпечує шифрування даних у транзиті та унеможливлює їх перехоплення. Серверні відповіді не повинні містити чутливих даних користувача у незашифрованому або надмірно детальному вигляді – зокрема, паролів, повних токенів зовнішніх платформ або внутрішніх технічних ідентифікаторів, що можуть бути використані для несанкціонованого доступу.

2.2.6 Розширюваність

Архітектура системи має забезпечувати можливість підключення нових платформ-джерел подій (YouTube) та нових платформ-одержувачів (Telegram, Slack) без внесення змін до існуючих сервісів. Розширення досягається через додавання нових consumer-сервісів, які підписуються на відповідні черги повідомлень [30], та внесення балансування кредитної системи з метою урахування відповідних особливостей платформ. Разом з цим, створення окремих модулів модерації над конкретними платформами, з метою розподілу обов’язків та зміни відповідного використання різних систем.

2.2.7 Доступність

Веб-інтерфейс системи має відповідати вимогам стандарту WCAG 2.1 рівня AA [29]. Інтерактивні компоненти (модальні вікна, випадаючі меню, сповіщення) мають підтримувати повне управління з клавіатури та бути сумісними з допоміжними технологіями (screen readers). Ця вимога реалізується через використання Radix UI Primitives, які надають коректні ARIA-атрибути та управління фокусом відповідно до специфікацій WAI-ARIA [29].

2.2.8 Конфігурованість

Уся конфігурація системи задається через змінні середовища. Валідація конфігурації виконується при старті кожного сервісу за принципом fail-fast: за відсутності або некоректного значення будь-якої обов'язкової змінної сервіс не запускається, що унеможливлює непередбачену поведінку в runtime.

2.3 Архітектурні рішення

Архітектура системи є результатом свідомих інженерних рішень, кожне з яких продиктоване конкретними вимогами або обмеженнями. У цьому підрозділі обґрунтовано ключові з них.

2.3.1 Мікросервісна архітектура та монорепозиторій

Серверна частина організована як сукупність чьотирьох мікросервісів: api, webhook-receiver, notification-wroker та discord-bot. Кожен сервіс має єдину відповідальність і може бути розгорнутий, масштабований та перезапущений незалежно від інших. Попри розподіленість сервісів, кодова база організована як єдиний npm workspaces монорепозиторій без використання зовнішніх оркестраторів на кшталт NX чи Turborepo. Такий підхід забезпечує атомарне розповсюдження змін у спільних бібліотеках (сутності бази даних, черги, RPC-патерни) на всі сервіси одночасно, виключаючи версійну несумісність між пакетами. Локальна розробка не потребує публікації внутрішніх пакетів – залежності резолвуються безпосередньо з файлової системи монорепозиторію.

Невиключно, що у подальшій розробці, може бути прийняте рішення зміни на користь оркестратора, та потенційно розбиття вже наявних мікросервісів на ще дрібніші, що будуть відповідати більш коректному визначення власної зони відповідальності, проте на поточний час, такий підхід

має спростити, як розробку, так і потенційне використання з розгортанням системи, тим самим дозволивши облегшити питання масштабування сервісів.

2.3.2 Окремий веб застосунок

З огляду на потребу у створенні повноцінного веб-інтерфейсу було прийнято рішення про розгортання окремого репозиторію для веб-застосунку на базі Next.js 16. Такий підхід зумовлює певне архітектурне обмеження: відсутність єдиного оркестратора унеможливлює спільну кодову базу та єдиний монорепозиторій між серверною і клієнтською частинами. Проте на поточному етапі розробки це є цілком допустимим рішенням. Основним його недоліком є відсутність спільного внутрішнього пакету для зберігання типів та DTO, що потребувало б окремої публікації бібліотеки. Водночас ця проблема компенсується суттєвою перевагою у спрощенні процесу збірки та незалежного розгортання застосунків.

Для зберігання стану застосунку в оперативній пам'яті (in-memory state) обрано Redux Toolkit – він відповідає за базову інформацію, доступну у будь-якому контексті застосунку: стан авторизації, параметри фільтрації та пошуку, а також глобальні події застосунку. Для виконання HTTP-запитів використовується клієнт Axios з обгорткою на основі TanStack Query, що забезпечує автоматичне кешування запитів, декларативне управління станами завантаження та помилок, а також надає реактивний доступ до результатів запитів у довільному контексті застосунку.

2.3.3 Винесення webhook-receiver в окремий сервіс

Не зважаючи на те, що ми маємо не надто типічний варіант мікросервісної архітектури, було прийнято рішення реалізувати окремий сервіс, який буде виступати інтерфейсом для сповіщень.

Twitch EventSub вимагає публічно доступного HTTPS-ендпоінту для надсилання подій. Якби цей ендпоінт був частиною основного API-сервісу, то ми могли б зіткнутися з потенційними проблемами навантаження трафіку та збільшення обробки запитів одним сервісом. Маючи окремий сервіс, який витсупає інтерфейсом, і відповідальність якого обмежена верифікацією підпису, дедуплікацією та публікацією події у чергу, ми зменшуємо навантаження на нашу систему, та спрощуємо процес доставки сповіщень. Крім того, ці два сервіси мають принципово різні профілі навантаження: webhook-receiver отримує різкі сплески трафіку на початку популярних трансляцій, тоді як Notification Worker функціонує під рівномірним навантаженням. Це визначає і різні підходи до їх масштабування.

Webhook-receiver є точкою зовнішньої взаємодії та основним приймачем вхідних HTTP-запитів від Twitch, тож його горизонтальне масштабування доцільно організувати через reverse proxy на базі nginx як балансувальника навантаження між інстансами. При цьому маршрутизація може бути налаштована таким чином, щоб конкретний clientId Twitch-додатку завжди спрямовувався до відповідного кластера. Такий підхід відкриває можливість побудови регіональних інсталяцій системи, де обробка подій відбувається на найближчому до користувача вузлі.

Notification Worker взаємодіє з іншими компонентами системи виключно через брокер повідомлень і є ізольованою точкою обробки подій. Його масштабування спрощується завдяки природі чергової моделі: довільна кількість інстансів може одночасно конкурентно споживати повідомлення з однієї черги RabbitMQ, рівномірно розподіляючи навантаження між собою [30]. У разі, якщо один інстанс не зміг успішно обробити подію, повідомлення повертається до черги та стає доступним для повторної обробки іншим інстансом. Механізм максимальної кількості спроб (retry limit) запобігає нескінченному циклу обробки некоректних повідомлень та забезпечує їх переміщення до dead-letter черги після вичерпання ліміту.

2.3.4 Комунікація через RabbitMQ

Міжсервісна взаємодія у більшості сценаріїв реалізована через брокер повідомлень RabbitMQ за протоколом AMQP. Цей підхід має низку переваг перед прямими HTTP-викликами між сервісами. По-перше, publisher не залежить від доступності consumer: якщо discord-bot тимчасово недоступний, повідомлення накопичуються в черзі і будуть оброблені після відновлення сервісу. По-друге, topic exchange дозволяє легко додавати нових consumer-підписників (наприклад, telegram-bot) без будь-яких змін у publisher. По-третє, черга природно буферизує навантаження, запобігаючи перевантаженню downstream-сервісів під час піків. Система використовує чотири черги: twitch.subscriptions (від webhook-receiver до api), twitch.events (від webhook-receiver до notification-worker) та discord.notifications (від api до discord-bot). Кожен мікросервіс взаємодіє з RabbitMQ через виділеного користувача з мінімально необхідними правами [31].

2.3.5 Синхронна комунікація через TCP RPC

Для сценаріїв, де потрібна негайна відповідь у рамках одного запиту – зокрема, перевірка автентифікації та отримання даних під час обробки Discord slash-команд – використовується синхронний TCP RPC [31] між discord-bot та api. На відміну від HTTP, TCP-транспорт у локальній Docker-мережі забезпечує нижчу затримку та не потребує додаткового шару маршрутизації. NestJS надає вбудовану підтримку цього підходу через ClientProxy та декоратори MessagePattern, що мінімізує кількість інфраструктурного коду.

2.3.6 Стратегія зберігання даних

Реляційна СУБД PostgreSQL 16 обрана як основне сховище даних завдяки підтримці ACID-транзакцій, що є критичним для кредитної системи: операції перевірки балансу та додавання призначення мають виконуватися атомарно,

щоб унеможливити перевищення ліміту за умов конкурентних запитів. Redis 7 використовується як швидкий TTL-кеш для переліку різних цілей: дедуплікація вхідних webhook-подій, зберігання JWT-blacklist та зберігання декодованих користувацьких токенів з інших платформ, протягом певного часу. TypeORM забезпечує декларативне визначення схеми через анотації сутностей та управління міграціями; міграції зберігаються у спільній бібліотеці та застосовуються одноразовим Docker-сервісом migrate до старту основного api.

2.3.7 Конфігурація та розгортання

Уся конфігурація системи передається через змінні середовища та валідується при старті за допомогою Zod-схем. Підхід fail-fast гарантує, що некоректно налаштований сервіс не запуститься та не спричинить важкодіагностованих помилок у runtime.

Розгортання серверної частини реалізоване через Docker Compose з явним визначенням порядку запуску та healthcheck-залежностей між сервісами: postgres → migrate → api; redis та rabbitmq → webhook-receiver; rabbitmq → discord-bot та notification-worker.

Веб-застосунок розгортається як окремий незалежний процес поза межами серверного Docker Compose, що відповідає архітектурному рішенню про розділення кодових баз. Фронтенд збирається командою next build та запускається як самостійний Node.js-процес, або у власному контейнері, що не має прямих залежностей від серверної інфраструктури та взаємодіє з API виключно через HTTP.

Зовнішній доступ до обох складових системи організовано через nginx як зворотний проксі-сервер. nginx відповідає за маршрутизацію вхідного трафіку: запити до API та webhook-ендпоінтів перенаправляються до відповідних серверних сервісів, тоді як запити до веб-інтерфейсу проксіюються до Next.js-застосунку. Така конфігурація дозволяє обслуговувати обидві складові системи

під єдиним доменом з розмежуванням за шляхом або субдоменом, централізовано управляти SSL-сертифікатами через Let's Encrypt, а також використовувати nginx як балансувальник навантаження між інстансами webhook-receiver при горизонтальному масштабуванні.

2.4 Аналіз Twitch API

Нашою основною зовнішньою платформою для взаємодії, на поточний момент часу – виступає Twitch. Twitch надає розробникам публічний REST API – Twitch Helix API – для програмної взаємодії з платформою. У контексті даної системи використовуються два ключових підрозділи цього API: механізм авторизації на базі OAuth 2.0 [2] та підсистема подієвих підписок EventSub [1]. Розуміння принципів роботи цих механізмів є необхідним для коректного проектування відповідних компонентів системи.

2.4.1 Авторизація: Twitch OAuth 2.0

Twitch реалізує авторизацію відповідно до стандарту OAuth 2.0 і підтримує кілька типів потоків (grant types) залежно від контексту використання. Для даної системи релевантними є два з них.

Client Credentials Grant – потік для отримання токена доступу від імені програми (app access token) без участі кінцевого користувача. Система надсилає POST-запит на endpoint https://id.twitch.tv/oauth2/token з параметрами client_id, client_secret та grant_type=client_credentials. У відповідь Twitch повертає access_token та його термін дії (expires_in). Цей тип токена використовується для операцій, що не потребують дозволу конкретного користувача: реєстрація EventSub-підписок, отримання інформації про стримерів та управління підписками. App access token не прив'язаний до конкретного облікового запису Twitch і діє від імені зареєстрованого додатку.

Authorization Code Grant – потік для отримання токена від імені конкретного користувача Twitch. Користувач перенаправляється на сторінку авторизації Twitch, де надає дозвіл додатку на доступ до свого облікового запису. Після підтвердження Twitch повертає одноразовий код авторизації, який система обмінює на пару токенів: access_token та refresh_token [27]. Цей тип авторизації використовується у сценаріях, де стример самостійно реєструється в системі та надає доступ до свого каналу для отримання розширених можливостей – зокрема, підписки на події, що потребують прав власника каналу (channel.subscribe, channel.cheer).

Термін дії access_token у Twitch становить приблизно 60 днів, після чого токен стає недійсним і всі підписки, пов'язані з ним, отримують статус revoked. Для запобігання цьому система реалізує механізм проактивного оновлення токенів через refresh_token: cron-задача TwitchSubscriptionsCronService регулярно перевіряє стан підписок та оновлює прострочені токени через endpoint https://id.twitch.tv/oauth2/token з параметром grant_type=refresh_token. Оновлені значення access_token та refresh_token зберігаються як у PostgreSQL (для персистентності), так і в Redis (для швидкого доступу при обробці запитів).

Кожен зареєстрований Twitch-додаток (TwitchApp у термінах системи) має власну пару client_id та client_secret, що ідентифікує його у запитах до Twitch API. Система підтримує кілька додатків одночасно, що дозволяє розподіляти підписки між ними та обходити ліміти, пов'язані з вартістю підписок на рівні окремого додатку.

2.4.2 EventSub: реєстрація підписок

Twitch EventSub є уніфікованим механізмом підписки на події платформи, що прийшов на заміну застарілим Webhooks v1 та PubSub. EventSub підтримує два типи транспорту: webhook (HTTP POST на зазначений URL) та WebSocket (постійне з'єднання). У даній системі використовується webhook-

транспорт, оскільки він є більш надійним для серверних застосунків та не потребує підтримки постійного з'єднання.

Реєстрація нової підписки виконується через POST-запит на endpoint https://api.twitch.tv/helix/eventsub/subscriptions з обов'язковим заголовком авторизації (Bearer app access token) та тілом запиту у форматі JSON. Тіло запиту містить три ключових поля:

- type – тип події (наприклад, stream.online, channel.raid);
- condition – умова фільтрації, що визначає, для якого конкретного каналу або користувача реєструється підписка (наприклад, broadcaster_user_id для більшості типів подій);
- transport – об'єкт з описом механізму доставки: метод (webhook), URL ендпоінту webhook-receiver та webhook_secret для генерації HMAC-підпису.

Кожна підписка має вартість (cost), що залежить від типу події. Загальна вартість усіх активних підписок одного додатку не може перевищувати ліміт (max total cost), що на момент розробки становить 10 000 одиниць для більшості додатків [1]. Система відстежує поточну вартість підписок для кожного TwitchApp (поле currentCost) та блокує реєстрацію нових підписок при наближенні до ліміту. Саме це обмеження є однією з причин підтримки кількох TwitchApp в системі – при вичерпанні ліміту одного додатку нові підписки реєструються через інший.

Успішна реєстрація підписки повертає об'єкт з унікальним ідентифікатором підписки (id), що зберігається у полі subscriptionId сутності TwitchStreamerEvent та використовується для подальшого управління підпискою. Статус новоствореної підписки на цьому етапі є webhook_callback_verification_pending – підписка ще не активна до проходження процедури верифікації.

2.4.3 EventSub: верифікація webhook (Challenge-Response)

Після реєстрації підписки Twitch негайно надсилає POST-запит на зазначений webhook URL для верифікації того, що сервер дійсно контролює цей ендпоінт та готовий приймати події. Запит містить заголовок Twitch-Eventsub-Message-Type зі значенням webhook_callback_verification та тіло з полем challenge – унікальним рядком.

Сервер має відповісти на цей запит протягом 10 секунд, повернувши значення challenge як plain text з HTTP-статусом 200. Якщо відповідь не надійшла вчасно або містить некоректні дані – Twitch скасовує підписку. При успішній верифікації статус підписки змінюється на enabled, що відображається у полі eventStatus сутності TwitchStreamerEvent зі значенням VERIFIED.

Кожен запит від Twitch містить набір заголовків для верифікації автентичності:

- Twitch-Eventsub-Message-Id – унікальний ідентифікатор повідомлення для дедуплікації;
- Twitch-Eventsub-Message-Timestamp – часова мітка відправки;
- Twitch-Eventsub-Message-Signature – HMAC-SHA256 підпис у форматі sha256={hex}.

Підпис обчислюється як HMAC-SHA256 від конкатенації значень заголовків Message-Id, Message-Timestamp та тіла запиту з використанням webhook_secret як ключа [1], [28]. webhook-receiver відтворює цей розрахунок та порівнює результат із отриманим підписом. Запити з невалідним підписом відхиляються з HTTP-статусом 403 до будь-якої подальшої обробки, що унеможливлює ін'єкцію підроблених подій від сторонніх джерел.

2.4.4 EventSub: отримання та повторна доставка подій

Після верифікації підписки Twitch надсилає POST-запити на webhook URL при кожному настанні відповідної події. Тіло запиту містить деталі події

у полі event, а заголовок Twitch-Eventsub-Subscription-Type вказує тип події для коректної маршрутизації [1].

Twitch гарантує доставку кожного повідомлення протягом приблизно 10 хвилин [1]: у разі якщо сервер не відповів статусом 2xx, Twitch повторює доставку з експоненційними затримками між спробами. Саме цей інтервал визначає TTL для Redis-ключів дедуплікації: 10-хвилинне вікно гарантує, що повторно доставлена подія буде розпізнана як дублікат і відхилена без повторної обробки.

Якщо Twitch не отримав підтвердження доставки протягом встановленого вікна або отримав занадто велику кількість помилок підряд – підписка отримує статус revoked з вказівкою причини (наприклад, authorization_revoked при простроченні токена або user_removed при видаленні каналу). Система виявляє такі підписки через cron-моніторинг та оновлює їхній статус у базі даних до REVOKED.

2.4.5 EventSub: управління підписками

Для перегляду поточного стану підписок система використовує GET-запит на https://api.twitch.tv/helix/eventsub/subscriptions з опціональними параметрами фільтрації за статусом, типом події або ідентифікатором користувача [1]. Відповідь містить пагінований список підписок з деталями кожної: id, status, cost, умови та транспорт. Саме ці дані використовує cron-сервіс для синхронізації стану підписок у локальній базі даних зі станом на боці Twitch, виявляючи розбіжності, що могли виникнути через мережеві збої або зовнішні зміни.

Видалення підписки виконується через DELETE-запит на https://api.twitch.tv/helix/eventsub/subscriptions?id=\{subscriptionId\} [1]. Система виконує видалення при деактивації призначення користувачем або при виявленні підписок у статусі REVOKED, що більше не мають активних

призначень. Це дозволяє вивільнити cost-квоту TwitchApp для нових підписок та підтримувати консистентний стан між локальною базою даних та Twitch.

2.5 Обґрунтування технологічного стеку

Вибір технологій для побудови розподіленої системи є одним з найважливіших інженерних рішень, що безпосередньо впливає на продуктивність, надійність, зручність розробки та довгострокову підтримуваність продукту. У цьому підрозділі обґрунтовано вибір кожної ключової технології через порівняння з альтернативами та аналіз відповідності конкретним вимогам системи.

2.5.1 Мова програмування: TypeScript

TypeScript є надмножиною JavaScript [34], що додає до останнього статичну типізацію, інтерфейси, generics та розширені можливості для об'єктно-орієнтованого програмування. Компілятор TypeScript транспілює код у JavaScript, що виконується на рушії Node.js, зберігаючи повну сумісність з екосистемою npm.

Головною перевагою TypeScript у контексті розподіленої системи є можливість визначення спільних типів у бібліотеках монорепозиторію та їх використання в усіх мікросервісах без додаткових перетворень.

Наприклад, тип події повідомлення, визначений у бібліотеці queue, автоматично застосовується як у publisher (api), notification-worker (дестриб’ютор) так і у consumer (discord-bot), що унеможливлює розбіжність у структурі даних між сервісами на етапі компіляції, хоч і не в runtime. Статична типізація суттєво прискорює розробку завдяки автодоповненню в IDE та ранньому виявленню помилок: за результатами дослідження Gao et al. [11],

близько 15% помилок у JavaScript-проєктах могли би бути виявлені компілятором TypeScript.

У новіших версіях Nodejs, ми вже маємо вбудовані компілятори для Typescript, що облегшує суттєво роботу з самих базових підходів, проте це все ще має потнеційні проблеми після компіляції, та певні нюанси сумісництва, які виникають при використання нашого оснвоного фреймворку NestJS. Ці проблеми, якраз узгоджуються через інші білдери, тож на поточний момент часу, ми не віддаємо перевагу нативному білдеру з NodeJS.

Порівняно з чистим JavaScript, TypeScript вимагає додаткового кроку компіляції та певного порогу входження для розробників, незнайомих з системами типів. Проте у контексті монорепозиторію з кількома взаємопов'язаними сервісами ці витрати є незначними порівняно з вигодою від гарантій типобезпеки на межах між сервісами.

Альтернативою міг би стати .NET (C#) – зріла екосистема від Microsoft зі статичною типізацією, потужним DI-контейнером та розвиненою підтримкою мікросервісних архітектур через ASP.NET Core. C# як мова є виразнішою за TypeScript у певних аспектах: підтримка справжньої багатопоточності через Thread та Task Parallel Library, більш строга система типів з підтримкою value types, а також значно вища продуктивність завдяки компіляції у нативний код через AOT (Ahead-of-Time compilation) у .NET 8. Фреймворк ASP.NET Core надає вбудовані засоби для побудови REST API, gRPC-сервісів та фонових задач (BackgroundService), що є функціональним еквівалентом NestJS у .NET-екосистемі.

Проте вибір .NET для даної системи є недоцільним з кількох причин. По-перше, екосистема npm налічує сотні тисяч пакетів для інтеграції з Discord API, Twitch API та RabbitMQ, тоді як NuGet-аналоги є менш зрілими та активно підтримуваними у сфері інтеграцій з ігровими та стримінговими платформами. По-друге, крива навчання .NET є суттєво вищою для розробників з JavaScript/TypeScript-бекграундом, тоді як перехід від Node.js до NestJS є

значно природнішим. По-третє, операційні вимоги .NET-сервісів є вищими: базовий .NET-образ Docker є суттєво більшим за Node.js Alpine-образ [24], а споживання пам'яті в режимі простою є вищим, що є помітним фактором при розгортанні кількох мікросервісів на одному хості. Таким чином, попри безсумнівні технічні переваги .NET у сфері продуктивності, TypeScript у поєднанні з NestJS забезпечує оптимальний баланс між зрілістю екосистеми, швидкістю розробки та операційною простотою для даного проєкту.

2.5.2 Web-фреймворк: Next.js та React

Next.js позиціонується розробником – компанією Vercel – як «фреймворк для створення повностекових веб-застосунків» на базі React [12], що надає вбудовані рішення для маршрутизації, рендерингу та оптимізації збірки. У системі використовується Next.js 16 з App Router – сучасною системою маршрутизації на базі файлової структури, що прийшла на заміну Pages Router у версії 13.

Ключовою особливістю App Router є підтримка React Server Components (RSC) – компонентів, що рендеряться на сервері та не додають JavaScript до клієнтського бандлу [12]. Це суттєво знижує початковий розмір завантажуваного коду та покращує показники Core Web Vitals. Вкладені layouts дозволяють декларативно визначати спільну розмітку для груп маршрутів – наприклад, DashboardLayout, що містить AuthGuard та бічну панель навігації

React 19, використаний як UI-рушій, вносить покращення у механізм Concurrent Mode, що дозволяє браузеру переривати важкі рендеринги та обробляти більш пріоритетні взаємодії користувача [13]. Це безпосередньо впливає на сприйману швидкодію інтерфейсу при завантаженні великих списків або виконанні мутацій.

Порівняння з альтернативами дозволяє обґрунтувати вибір. Vite + React SPA є простішим рішенням без SSR, проте позбавляє можливості серверного

рендерингу та файлової маршрутизації, що ускладнює масштабування структури сторінок. Angular є повноцінним MVC-фреймворком зі строгими конвенціями, проте його архітектурна надмірність є невиправданою для задачі адміністративної веб-панелі. Таким чином, Next.js забезпечує оптимальний баланс між функціональністю, зрілістю екосистеми та швидкістю розробки.

2.5.3 Управління станом: Redux Toolkit та TanStack Query

Управління станом у клієнтській частині системи базується на принципі чіткого розмежування між двома категоріями даних, що мають принципово різні характеристики: синхронний персистентний стан та серверний асинхронний стан. Для кожної категорії обрано спеціалізований інструмент, що оптимально вирішує відповідну задачу.

Redux Toolkit (RTK) є офіційною бібліотекою для роботи з Redux, що суттєво знижує обсяг boilerplate-коду через слайси, immer-мутації та вбудований DevTools [14]. У системі RTK відповідає виключно за синхронний персистентний стан – статус авторизаційної сесії, токени доступу та бащзові стани фільтрації, чи пошукового контексту. Цей стан є синхронним, оскільки оновлюється лише через явні дії користувача (вхід, вихід, ініціалізація при завантаженні, зміну глобального пошуку), та персистентним, оскільки повинен зберігатися між навігаціями та бути доступним будь-якому компоненту дерева без передачі через props. Спроба реалізувати цей функціонал через Context API призвела б до некерованих ре-рендерів усіх підписаних компонентів при кожній зміні токена.

TanStack Query є бібліотекою для управління серверним асинхронним станом – даними, що отримуються від API та мають власний lifecycle: завантаження, кешування, інвалідація, фонове оновлення [15]. Бібліотека надає декларативні хуки (useQuery, useMutation), що автоматично управляють loading/error станами, дедуплікують однакові запити та інвалідують кеш при

мутаціях. Конфігурація staleTime: 60000 мс визначає, що дані вважаються актуальними протягом хвилини – після чого при наступному зверненні виконується фонове оновлення без блокування UI. Порівняно з SWR (аналогічна бібліотека від Vercel), TanStack Query надає більш розвинений API для складних сценаріїв інвалідації та оптимістичних оновлень.

Сумісне використання двох систем не є надмірністю, а є архітектурно обґрунтованим рішенням: RTK та React Query вирішують різні задачі і не дублюють функціональність. RTK Query (вбудований у Redux Toolkit аналог React Query) міг би замінити TanStack Query, проте він є менш гнучким при складних сценаріях кешування та має більш обмежену екосистему плагінів.

2.5.4 HTTP-клієнт: Axios

Axios – це Promise-based HTTP-клієнт для JavaScript з підтримкою перехоплювачів (interceptors), трансформації запитів та відповідей, та автоматичною серіалізацією JSON [16]. У системі Axios виконує роль єдиної точки виходу всіх HTTP-запитів з централізованою логікою автентифікації.

Архітектура клієнта для Web-застосунку побудована навколо двох interceptorів. Перший (request interceptor) автоматично додає Bearer-заголовок до кожного запиту, читаючи поточний access token з AuthService. Це усуває необхідність передавати токен вручну з кожного місця виклику API. Другий (response interceptor) реалізує механізм автоматичного оновлення токенів при отриманні відповіді зі статусом 401.

Разом з цим, для серверного фреймворку NestJS 11, так само під коптом використовується клієнт Axios, проте з обгорткою, що робить з нього вже більш як Http Client, який працює більш як на базі RxJS, який ми використовуємо по меншій мірі виключно для серверної частини, так як сам фреймворк має у багатьох процесах задіяним саме його.

Порівняно з нативним Fetch API, Axios надає переваги у вигляді автоматичного перетворення JSON, interceptorів, можливості скасування запитів через AbortController та більш зручного API для обробки помилок. Fetch API, попри вбудованість у браузер та відсутність залежностей, потребує значного обсягу boilerplate-коду для реалізації аналогічної функціональності.

2.5.5 UI-система: Tailwind CSS, SCSS та Radix UI

UI-система клієнтської частини побудована на трьох рівнях, що вирішують різні аспекти розробки інтерфейсу: утилітарний CSS-фреймворк для стилізації, SCSS препроцесор та headless бібліотека компонентів для поведінкової логіки та доступності.

Tailwind CSS v4 – це utility-first CSS-фреймворк, де стилізація здійснюється через застосування атомарних CSS-класів безпосередньо у розмітці [18]. Версія 4 принесла кардинальні зміни у підхід до конфігурації: замість JavaScript-конфігу (tailwind.config.js) дизайн-токени визначаються у CSS через CSS Custom Properties, що забезпечує органічну інтеграцію з браузерними механізмами кастомних властивостей. Утиліта cn() на основі clsx та tailwind-merge вирішує задачу умовного поєднання класів та дедуплікації конфліктуючих утиліт – наприклад, при розширенні базового компонента новими класами стилів.

Окремим шаром системи стилізації виступає SCSS у поєднанні з CSS Modules. SCSS (Sassy CSS) є препроцесором, що розширює можливості нативного CSS змінними, вкладеністю, міксинами, функціями та директивою @use для модульного імпорту. CSS Modules – це механізм на рівні збирача (у даному випадку Next.js + PostCSS), що автоматично генерує унікальні імена класів для кожного модуля, запобігаючи конфліктам стилів між компонентами у глобальному просторі імен.

Порівняно з опіненейтед UI-бібліотеками на кшталт Material UI або Ant Design, Tailwind надає повний контроль над візуальним результатом без необхідності перевизначати чужі стилі – що є суттєвою перевагою при розробці продукту з власною брендовою ідентичністю.

Radix UI Primitives – це бібліотека headless UI-компонентів, тобто компонентів з реалізованою поведінковою логікою та семантикою доступності, але без жодних вбудованих стилів [17]. Це дозволяє отримати такі переваги готових рішень, як ARIA-атрибути, управління фокусом, keyboard navigation та screen reader підтримку, зберігаючи повний контроль над візуальним оформленням. Зокрема, компонент Dialog (модальне вікно) автоматично управляє фокус-трепом (утримання фокусу всередині модалки), додає aria-modal та role="dialog" атрибути та обробляє закриття через Escape – поведінку, яку складно та трудомістко реалізувати вручну відповідно до вимог стандарту WCAG 2.1 [29].

Порівняно з використанням нативних HTML-елементів без бібліотеки, Radix UI суттєво знижує ризик порушень доступності, які є невидимими при ручному тестуванні, але критичними для користувачів з обмеженими можливостями та для проходження автоматизованих аудитів доступності.

2.5.6 Серверний фреймворк: NestJS 11

Згідно з офіційною документацією, NestJS є «прогресивним Node.js-фреймворком для побудови ефективних та масштабованих серверних застосунків» [19], натхненним архітектурними принципами Angular. Фреймворк побудований поверх Express (або Fastify як альтернативного HTTP-адаптера) і надає потужний шар абстракцій над ними.

Ключовою особливістю NestJS є вбудований IoC-контейнер (Inversion of Control) та система Dependency Injection, що дозволяє декларативно описувати залежності між компонентами через декоратори [19]. Це суттєво знижує

зв'язність між модулями та спрощує тестування: будь-яка залежність може бути замінена mock-об'єктом без змін у коді компонента. Модульна система NestJS дозволяє організовувати код у самодостатні блоки (модулі), кожен з яких інкапсулює свої контролери, сервіси та репозиторії, що природно відповідає принципу єдиної відповідальності.

Принципово важливою для даної системи є вбудована підтримка мікросервісних транспортів. NestJS «з коробки» підтримує RabbitMQ (через @nestjs/microservices з AMQP-адаптером) та TCP RPC як рівноправні транспортні механізми поряд з HTTP [19]. Це означає, що один і той самий сервіс може одночасно слухати HTTP-запити на порту 3000 та TCP RPC на порту 3010, використовуючи однакові декоратори (@MessagePattern, @EventPattern) для обох транспортів. Без NestJS реалізація такої гібридної архітектури потребувала б значного обсягу інфраструктурного коду.

2.5.7 База даних: PostgreSQL 16

PostgreSQL є однією з найбільш функціональних реляційних СУБД з відкритим кодом, що активно розвивається понад 35 років [20]. Версія 16 принесла покращення у паралельному виконанні запитів, логічній реплікації та продуктивності bulk-операцій.

Вибір реляційної моделі даних для даної системи є принциповим, а не випадковим. Операції кредитної системи – перевірка балансу та списання кредитів при додаванні призначення – мають виконуватися атомарно в рамках однієї ACID-транзакції, щоб виключити стан гонки за умов конкурентних запитів. Реляційна модель з підтримкою зовнішніх ключів також гарантує цілісність даних на рівні СУБД: неможливо створити TwitchStreamerEvent без відповідного TwitchStreamer, що усуває цілий клас помилок консистентності.

Альтернативним варіантом є MongoDB, що представляє собою документоорієнтованою СУБД, що зберігає дані у форматі BSON-документів

без фіксованої схеми. MongoDB є привабливим вибором для систем з нерегулярною структурою даних або для сценаріїв, де горизонтальне шардування є критичною вимогою з перших днів. Проте для даної системи відсутність фіксованої схеми є недоліком: структура сутностей чітко визначена і не змінюється динамічно. Крім того, MongoDB до версії 4.0 не підтримував multi-document транзакції, а їхня реалізація у пізніших версіях є суттєво складнішою за PostgreSQL-транзакції з точки зору продуктивності. Відсутність підтримки зовнішніх ключів на рівні СУБД перекладає відповідальність за цілісність даних на рівень застосунку, що є потенційним джерелом помилок.

2.5.8 ORM: TypeORM

TypeORM є найбільш зрілим ORM-рішенням для TypeScript-екосистеми [21], що підтримує декоратори для визначення сутностей, систему міграцій, Repository Pattern та ActiveRecord-підхід.

Ключовою перевагою TypeORM є органічна інтеграція з NestJS через офіційний пакет @nestjs/typeorm, що надає TypeOrmModule з підтримкою Dependency Injection для репозиторіїв. Визначення сутностей через TypeScript-декоратори (@Entity, @Column, @ManyToOne тощо) дозволяє зберігати схему бази даних безпосередньо в коді, що спрощує розуміння структури даних і знижує ризик розходження між кодом та схемою [21]. Система міграцій TypeORM генерує SQL-скрипти на основі різниці між поточною схемою та попередньою версією, що забезпечує відтворюване та версійоване управління схемою бази даних.

Найбільш відповідним аналогом, який можна було б розглядати – Prisma. Сучасне ORM-рішення з принципово іншим підходом: схема визначається у власному декларативному форматі (.prisma-файли), а Prisma Client генерується автоматично. Prisma забезпечує відмінний developer experience та типобезпеку на рівні запитів. Проте інтеграція Prisma з NestJS є менш органічною порівняно

з TypeORM, а підхід з кодогенерацією додає крок до процесу збірки. Крім того, Prisma має обмеження у роботі зі складними запитами та кастомними SQL-виразами, що може стати проблемою при розширенні системи.

2.5.9 Брокер повідомлень: RabbitMQ

RabbitMQ є одним з найбільш поширених брокерів повідомлень з відкритим кодом, що реалізує протокол AMQP (Advanced Message Queuing Protocol). Система активно розвивається з 2007 року та є зрілим промисловим рішенням з понад 15-річною історією розвитку, широко застосовуваним у комерційних системах різного масштабу [22].

Центральним поняттям RabbitMQ є exchange – компонент, що отримує повідомлення від publisher та маршрутизує їх до відповідних черг за визначеними правилами. Topic exchange, використаний у даній системі, дозволяє маршрутизувати повідомлення за шаблонами routing key, що надає гнучкість при додаванні нових типів подій без змін у конфігурації існуючих черг. Механізм підтвердження доставки (acknowledgement) гарантує, що повідомлення не буде видалено з черги до явного підтвердження його обробки consumer-ом, що забезпечує надійність за умов збоїв.

2.5.10 Кеш та сховище стану: Redis

Redis (Remote Dictionary Server) – резидентна in-memory база даних з підтримкою персистентності, що пропонує широкий набір структур даних: рядки, хеші, списки, множини, sorted sets та потоки [23]. У системі Redis виконує три принципово різні функції: дедуплікацію webhook-подій, Cache дані та роль оперативного сховища для чутливих до затримки даних автентифікації.

Для дедуплікації webhook-подій Redis використовується як TTL-кеш: ключ з ідентифікатором повідомлення зберігається з часом життя 10 хвилин – саме такий часовий інтервал гарантує Twitch для повторних доставок одного

повідомлення. Команда SET з параметром NX (set if not exists) є атомарною операцією, що одночасно перевіряє наявність ключа та встановлює його, виключаючи стан гонки при конкурентних запитах.

Для JWT-blacklist Redis зберігає інвалідовані токени до моменту закінчення їхнього природного терміну дії. Цей підхід є компромісом між безпекою та продуктивністю: перевірка кожного запиту через Redis додає один мережевий round-trip, але дозволяє примусово завершувати сесії без зміни secret-ключа, що інвалідувало б усі активні токени одночасно.

Третьою функцією Redis є роль оперативного сховища для даних, що потребують високошвидкісного доступу в процесі обробки запитів. Зокрема, Redis зберігає декоровані access-токени користувачів та Twitch-додатків, що дозволяє уникнути повторних звернень до PostgreSQL при кожній автентифікованій операції. Окремо зберігаються webhook-секрети для кожного Twitch-додатку – значення, що використовується webhook-receiver при верифікації HMAC-SHA256 підпису кожної вхідної події. Винесення цих даних у Redis замість читання з PostgreSQL при кожному вхідному webhook-запиті є принциповим рішенням з точки зору продуктивності: webhook-receiver обробляє сплески трафіку під час одночасного початку трансляцій у великої кількості стримерів, і додаткове звернення до реляційної СУБД при кожному запиті могло б стати вузьким місцем. Redis завдяки своїй in-memory природі забезпечує час відповіді на рівні мікросекунд, що є на кілька порядків швидшим за PostgreSQL-запит з мережевим round-trip.

2.5.11 Контейнеризація: Docker та Docker Compose

Docker забезпечує ізольоване та відтворюване середовище виконання через механізм контейнеризації на рівні операційної системи, що дозволяє пакувати застосунок разом з усіма його залежностями [24]. Кожен мікросервіс системи пакується у власний Docker-образ за допомогою multi-stage Dockerfile:

build-stage встановлює devDependencies та компілює TypeScript, production-stage копіює лише скомпільований dist та production node_modules, що суттєво зменшує розмір фінального образу.

Docker Compose використовується для оркестрації всіх сервісів системи як єдиного цілого у межах одного хоста. Compose-файл декларативно визначає порядок запуску сервісів через depends_on з умовами healthcheck: postgres, redis та rabbitmq мають пройти healthcheck перед стартом залежних від них сервісів. Окремий сервіс migrate виконує міграції бази даних одноразово та завершується, після чого стартує api. Така конфігурація гарантує коректний стан усіх залежностей незалежно від швидкості їхньої ініціалізації.

Кастомний RabbitMQ-образ містить init-скрипт, що при першому запуску створює необхідні vhost та користувачів з відповідними правами, що реалізує принцип найменших привілеїв та інфраструктуру-як-код – стан середовища повністю описаний у репозиторії та може бути відтворений з нуля за одну команду.

2.6 Архітектурні паттерни та принципи

Архітектура системи базується на низці усталених патернів проектування, кожен з яких вирішує конкретну задачу організації коду та взаємодії між компонентами. У цьому підрозділі описано патерни, застосовані у системі, із прив’язкою до конкретних компонентів реалізації.

2.6.1 Repository Pattern

Патерн застосовується для абстрагування логіки доступу до даних від бізнес-логіки. Кожна сутність має власний репозиторій, що інкапсулює всі операції з базою даних і надає сервісному шару типізований інтерфейс для роботи з даними. Наприклад, TwitchAppsRepository визначає методи

findActiveApp, findByClientId та updateCost, повністю приховуючи від сервісу деталі SQL-запитів та TypeORM-специфічний код. Такий підхід спрощує тестування: замість реальної бази даних у тестах підставляється mock-реалізація репозиторію з фіксованими відповідями.

2.6.2 Service Layer

Цей підхід забезпечує чітке розмежування між транспортним шаром та бізнес-логікою. Контролери та RPC-обробники є навмисно «тонкими»: їхня єдина відповідальність – прийняти запит, делегувати виконання відповідному сервісу та повернути результат. Вся бізнес-логіка зосереджена у сервісах: перевірка кредитного балансу, формування payload сповіщення, управління станом підписок. Це дозволяє повторно використовувати бізнес-логіку в різних контекстах – наприклад, одна й та сама логіка додавання призначення викликається як через HTTP REST API, так і через TCP RPC від Discord-бота.

2.6.3 DTO та Mapper Pattern

DTO та Mapper Pattern визначає чіткі межі між зовнішніми представленнями даних та внутрішніми сутностями. Вхідні дані від клієнта приймаються у вигляді DTO (Data Transfer Object) – класів з декораторами class-validator, що автоматично валідуються через NestJS ValidationPipe перед потраплянням до контролера. Це гарантує, що до сервісного шару потрапляють лише валідні дані правильної структури. Маппінг між DTO та Entity виконується явно у сервісному шарі, що дозволяє розвивати API та схему бази даних незалежно.

2.6.4 Guard та Strategy Pattern

Класичні патерни, для NestJS, які реалізують механізм автентифікації та авторизації. NestJS Guards – це клас, що реалізує інтерфейс CanActivate та може

бути застосований до окремих ендпоінтів або цілих контролерів через декоратори [19]. JwtAuthGuard перехоплює кожен захищений запит, витягує JWT-токен з Authorization-заголовка, верифікує підпис та термін дії, перевіряє наявність токена у Redis-blacklist та у разі успіху додає дані користувача до об'єкту запиту. Passport Strategies інкапсулюють логіку конкретного методу автентифікації (JWT, Discord OAuth), що дозволяє додавати нові методи без змін у захищених ендпоінтах [26].

2.6.5 EventPattern та MessagePattern

У контексті NestJS, EventPattern та MessagePattern є декораторами [19], що перетворюють звичайні методи класу на обробники повідомлень відповідного транспорту. EventPattern позначає one-way обробники подій (fire-and-forget) для асинхронних RabbitMQ-повідомлень, тоді як MessagePattern – обробники з відповіддю для синхронного TCP RPC. Завдяки цьому абстрактному шару код обробника залишається ідентичним незалежно від того, чи прийшло повідомлення через RabbitMQ, чи через TCP: транспортний механізм є деталлю конфігурації, а не бізнес-логіки.

2.6.6 Dependency Injection та IoC Containe

Dependency Injection та IoC Container є фундаментальним патерном NestJS [19, 30], що пронизує всю архітектуру системи. Замість того, щоб компоненти самостійно створювали свої залежності через new, вони декларують їх у конструкторі, а IoC-контейнер NestJS автоматично резолвує та ін'єктує необхідні екземпляри. Це забезпечує loose coupling між компонентами, спрощує тестування через ін'єкцію mock-залежностей та дозволяє NestJS управляти lifecycle об'єктів (singleton, request-scoped тощо).

2.6.7 Fail-Fast принцип

Основа цього принципу реалізована на рівні конфігурації через Zod-валідацію ENV-змінних при старті кожного сервісу [25]. Якщо будь-яка обов'язкова змінна відсутня або має некоректне значення – сервіс завершується з детальним звітом про помилку до того, як буде встановлено будь-яке підключення до зовнішніх систем. Це принципово відрізняється від підходу «дізнаємось про проблему в runtime», коли помилка конфігурації може проявитися лише за певних умов навантаження або при зверненні до рідко використовуваного функціоналу.

2.6.8 Cron-based моніторинг

Cron-based моніторинг стану підписок реалізований через TwitchSubscriptionsCronService, що за розкладом перевіряє актуальний стан EventSub-підписок через Twitch API [1] та синхронізує його зі станом у базі даних. Це є превентивним механізмом виявлення та відновлення після аномалій: підписки можуть бути відкликані Twitch у разі прострочення токена доступу або зміни налаштувань додатку. Без такого механізму система могла б не отримувати події протягом тривалого часу без жодного індикатора проблеми.

2.6.9 AuthGuard Pattern

AuthGuard є декларативним механізмом захисту клієнтських маршрутів, що реалізується на рівні layout-компонента Next.js. Компонент підписується на статус авторизаційної сесії з Redux store та при значенні guest негайно перенаправляє користувача на сторінку входу без рендерингу захищеного контенту. Стан initAuth (ініціалізація при завантаженні) обробляється поверненням null – порожній рендер до завершення перевірки токена запобігає

короткочасному мерехтінню захищеного контенту (flash of unauthorized content).

Перевага цього підходу порівняно з серверним захистом маршрутів полягає у відсутності мережевого round-trip: перевірка статусу відбувається синхронно з локального стану без звернення до API. Серверна перевірка залишається відповідальністю API через JWT-валідацію на кожному захищеному ендпоінті [32].

2.6.10 Provider Chain Pattern

Ієрархія провайдерів StoreProvider → QueryClientProvider → AuthProvider реалізує принцип односпрямованої залежності: зовнішні провайдери не залежать від внутрішніх, тоді як внутрішні мають доступ до контексту зовнішніх [32]. Зокрема, AuthProvider знаходиться найглибше в ієрархії та має доступ до Redux dispatch (через StoreProvider) та QueryClient (через QueryClientProvider), що дозволяє при ініціалізації сесії одночасно оновлювати глобальний стан та прогрівати кеш серверних даних.

Порядок провайдерів є принциповим архітектурним рішенням – зміна ієрархії призведе до помилок доступу до контексту на рівні компіляції TypeScript.

2.6.11 Interceptor Pattern

Централізація логіки оновлення токенів в HTTP-interceptori є застосуванням принципу єдиної відповідальності до рівня мережевого шару: жоден компонент або хук не повинен самостійно обробляти 401 помилки. Черга відкладених запитів (queue) реалізує патерн, відомий як «deferred promise» – запити не відхиляються при тимчасовій неможливості їх виконання, а утримуються до відновлення стану системи [32]. Це забезпечує прозорість для всіх шарів застосунку вище: компоненти та хуки React Query не «знають» про

механізм оновлення токенів і не потребують додаткової логіки обробки помилок автентифікації.

ВИСНОВОК ДО РОЗДІЛУ

У другому розділі сформульовано вимоги до розроблюваної системи та обґрунтовано ключові технічні й архітектурні рішення, покладені в основу її реалізації.

В результаті аналізу функціональних вимог визначено п'ять основних підсистем: автентифікації та авторизації, управління Twitch-підписками, управління призначеннями сповіщень, кредитну підсистему та підсистему веб-інтерфейсу. Кожна відповідає конкретній прогалині, виявленій в існуючих аналогах: гнучка кредитна модель з диференційованою вартістю підписок усуває обмеженість фіксованих тарифних планів; підтримка двох механізмів доставки (Bot та Webhook) розширює сценарії використання; інтеграція стримерів як повноправних користувачів надає можливості, відсутні у жодному з розглянутих рішень; веб-інтерфейс забезпечує повноцінний графічний канал взаємодії та коректну реалізацію OAuth 2.0 Authorization Code Flow з належним UX-супроводом.

Нефункціональні вимоги встановлюють цільові показники якості системи: максимальна затримка доставки повідомлень за нормальних умов навантаження, гарантована доставка за умов часткової недоступності компонентів, горизонтальна масштабованість кожного мікросервісу незалежно, відповідність веб-інтерфейсу стандарту WCAG 2.1 рівня AA, розширюваність на нові платформи без змін у існуючій кодовій базі, а також захист клієнтського стану через розмежування зберігання токенів власної системи та токенів зовнішніх платформ.

Проведений аналіз Twitch API визначив конкретні механізми взаємодії з платформою: двоетапну авторизацію через OAuth 2.0 (Client Credentials для серверних операцій та Authorization Code для користувацьких) [2], процедуру

Challenge-Response верифікації webhook-ендпоінту, HMAC-SHA256 верифікацію кожної вхідної події та стратегію управління cost-квотою підписок на рівні TwitchApp [1].

Обґрунтування технологічного стеку підтвердило доцільність кожного з обраних рішень через порівняння з альтернативами. На серверній стороні: NestJS забезпечує вбудовану мікросервісну інфраструктуру та DI-контейнер; TypeScript гарантує типобезпеку на межах між сервісами монорепозиторію; PostgreSQL забезпечує ACID-транзакції для кредитної системи; RabbitMQ – надійну асинхронну доставку з буферизацією; Redis – високошвидкісний доступ до чутливих до затримки даних, дедуплікацію подій та зберігання JWT-blacklist; Docker Compose та nginx – відтворюване розгортання з централізованим управлінням трафіком. На клієнтській стороні: Next.js із App Router забезпечує серверний рендеринг та оптимізовану збірку; Redux Toolkit відповідає за синхронний персистентний стан авторизації; TanStack Query – за серверний асинхронний стан з кешуванням; Axios – за централізовану HTTP-комунікацію з автоматичним оновленням токенів; Radix UI у поєднанні з Tailwind CSS та SCSS Modules – за доступний та гнучко стилізований інтерфейс.

Визначені архітектурні паттерни – Repository Pattern, Service Layer, DTO та Mapper, Guard та Strategy, EventPattern та MessagePattern, Dependency Injection, Fail-Fast конфігурація, а також клієнтські AuthGuard, Provider Chain та Interceptor Pattern – формують узгоджену систему принципів, що забезпечує низьку зв'язність між компонентами, високу тестованість та передбачувану поведінку системи в умовах відмов окремих вузлів.

Сукупність прийнятих рішень формує повну архітектурну основу для реалізації системи, що описується у наступному розділі.

РОЗДІЛ 3. ПРОЕКТУВАННЯ ТА РЕАЛІЗАЦІЯ СИСТЕМИ

У третьому розділі описано практичну реалізацію системи, що отримала технічну назву «Xelar», на основі архітектурних рішень та вимог, сформульованих у попередньому розділі. Розглянуто структуру кодової бази, схему бази даних, реалізацію кожного мікросервісу, механізми їх взаємодії та конфігурацію розгортання. Разом з цим, буде розглянуто впроваджений механізм CI CD.

3.1 Структура серверної частини у вигляді монорепозиторію

Кодова база серверної частини системи організована як npm workspaces монорепозиторій на базі NestJS [19], що об'єднує чотири виконувані мікросервіси та шість спільних бібліотек під єдиним кореневим package.json. Така організація забезпечує спільне використання типів, утиліт та конфігурацій між сервісами без необхідності публікації окремих пакетів до зовнішнього реєстру. Разом з цим, такий підхід вніс суттєву проблему в етапі розгортання сервісів, цей етап детально буде описано в подальшому розділі.

Файлова структура монорепозиторію побудована за принципом розмежування виконуваних застосунків та бібліотек спільного використання.

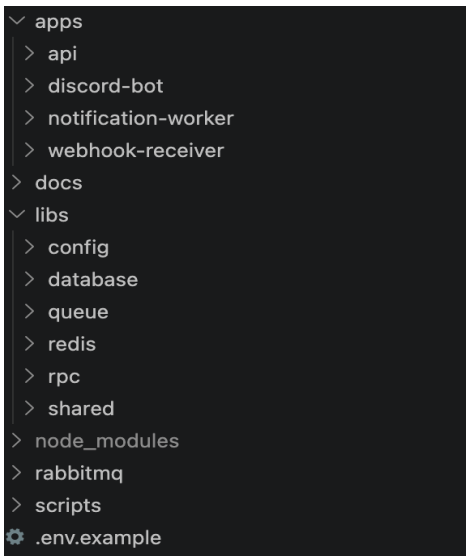


Рисунок 3.1 – Структура монорепозиторію системи

Виконувані сервіси ініціалізуються засобами NestJS CLI через команду nest generate app <name>, що автоматично створює необхідну структуру модулів, точку входу та конфігурацію збірки. Спільні бібліотеки створюються командою nest generate library <name> і резолвуються локально через path aliases у tsconfig.json без необхідності окремого деплою до пакетного реєстру.

Кожна бібліотека несе конкретну відповідальність. Бібліотека @xelar/config реалізує централізовану валідацію змінних середовища через Zod-схеми: кожен сервіс підключає лише релевантну для себе частину конфігурації, а за відсутності або некоректного значення обов'язкової змінної – завершується з детальним звітом про помилку до встановлення будь-яких зовнішніх підключень. Бібліотека @xelar/database містить TypeORM-сутності, міграції та data source конфігурацію – вона використовується як в api, так і в одноразовому сервісі migrate. Бібліотека @xelar/queue визначає типи подій та routing keys для всіх RabbitMQ-черг системи, гарантуючи типобезпеку на межі між publisher та consumer. Бібліотека @xelar/rpc аналогічно визначає патерни та типи повідомлень для TCP RPC між discord-bot та api. Бібліотека @xelar/redis надає сконфігурований ioredis-клієнт. Бібліотека @xelar/shared містить загальні утиліти, зокрема криптографічні функції для верифікації HMAC-підписів та

шифрування чутливих полів. Всі деталі реалізації будуть розлянуті у подальши розділах, разом з прикладами використання в основних сервісах системи.

3.2 Проектування бази даних

Схема бази даних реалізована засобами TypeORM [21] та розміщена у спільній бібліотеці @xelar/database. Всі міграції зберігаються у директорії libs/database/src/migrations/ та застосовуються командою migration:run одноразовим Docker-сервісом migrate до старту основного API. Модель даних складається з дев'яти сутностей, що відображають повну бізнес-логіку системи підписок, кредитної моделі та управління доступом. Відповідну ER-діаграму може буде побачити у додатку до дипломної роботи.

Сутність User є центральним об'єктом системи та містить облікові дані користувача: email, хешований пароль, закодовані токени до інших систем, Discord- та Twitch-ідентифікатори, а також основу кредитної системи у вигляді балансу користувача. Токени зовнішніх платформ (discordAccessToken, discordRefreshToken, twitchAccessToken, twitchRefreshToken) зберігаються у зашифрованому вигляді та використовуються виключно для серверних операцій без передачі клієнту.

Сутність TwitchStreamer представляє канал Twitch у системі та містить broadcasterId – унікальний ідентифікатор каналу на платформі, – а також twitchLogin та displayName. Стример може бути зареєстрований як повноправний користувач системи через зв'язок з User, або існувати виключно як зовнішній стример без облікового запису. Це визначає фактичний підхід, створення підписок іншими користувачами системи, які будуть зав’язні на цьому каналі. Якщо власник каналу є авторизованим користувачем системи – він надає інши користувачам можливість створювати підписки «у кредит», детальне пояснення цієї концепції буде надано пізніше, проте у загальних

рисах, ця конструкція впливає на собівартість створення підписок, що відповідно буде надавати користувачеві більші можливості у модерації своїх сповіщень.

Сутність TwitchApp представляє зареєстрований Twitch-додаток з полями clientId, clientSecret, webhookSecret та accessToken. Поля currentCost та maxCost відстежують сумарну вартість активних EventSub-підписок у межах одного додатку. Поле status типу enum визначає, чи приймає додаток нові підписки (значення ACTIVA, INACTIVE або LOCKED). Чутливі поля автоматично шифруються при збереженні та розшифровуються при читанні через CryptoService.

Сутність TwitchStreamerEvent є сполучною між стримером та Twitch-додатком і представляє конкретну EventSub-підписку. Містить тип події (event) та її поточний статус (eventStatus): PENDING – підписка зареєстрована, очікує верифікації; VERIFIED – Twitch підтвердив підписку через challenge-response; REVOKED – підписку скасовано платформою. Поле subscriptionId зберігає унікальний ідентифікатор від Twitch для подальшого управління підпискою.

Сутність DiscordNotification представляє призначення сповіщення через Discord Bot – канал, у який надсилається повідомлення при настанні відповідної події. Містить тип «оплати», ідентифікатори каналу та сервера, а також messagePayload у форматі JSONB для зберігання кастомного вмісту. Поле creditCost фіксує вартість призначення на момент його створення і не підлягає ретроактивному перерахунку.

Сутність WebhookNotification представляє призначення для доставки сповіщень через HTTP-вебхук. На відміну від бот-сповіщень, цей спосіб не вимагає присутності Discord-бота на цільовому сервері: система виконує HTTP POST-запит безпосередньо на зазначену URL-адресу. Поле webhookUrl оголошено з прапорцем select: false на рівні TypeORM-сутності, що унеможливлює його ненавмисне включення у відповіді REST API. Але разом з цим, вебхук може бути прив’язаним до Discord сервера, що може задати інший

типу відповідальності у відносинах оплати, та відповідно валідації відповідності сервера до самої URL-адреси.

Сутність DiscordGuild представляє Discord-сервер у системі та реалізує механізм колективного кредитного балансу на рівні спільноти. Поле guildId індексується засобами TypeORM-декоратора @Index() для прискорення пошуку. Поле balance з десятковою точністю дозволяє списувати вартість сповіщень з балансу спільноти замість персонального балансу користувача. Каскадне видалення забезпечує автоматичне очищення пов'язаних сповіщень при видаленні запису. Значення managerRoleId – надає можливість налаштовувати права доступу користувача до налаштувань спільноти та додавання сповіщень на сервер.

Сутність Role реалізує рольову модель управління доступом (RBAC). Кожна роль має унікальне ім'я, числовий пріоритет rolePriority та масив прав доступу типу Permission, що зберігається як PostgreSQL-масив enum. Визначені права: ADMIN (повний адміністративний доступ), READ_APPS та MANAGE_APPS (перегляд і управління Twitch-додатками), READ_ROLES та MANAGE_ROLES (перегляд і управління ролями). Зв'язок ManyToMany з UserEntity через проміжну таблицю user_roles дозволяє призначати користувачу кілька ролей одночасно.

Сутність UserSession реалізує управління сесіями для механізму ротації refresh-токенів. Кожен успішний вхід або оновлення токена створює новий запис сесії з полями refreshTokenHash (bcrypt-хеш refresh-токена) та expiresAt. При запиті на оновлення токена сервіс верифікує відповідність переданого токена збереженому хешу та негайно видаляє стару сесію, замінюючи її новою. Зв'язок ManyToOne із каскадним видаленням гарантує очищення всіх сесій при видаленні облікового запису користувача.

Сутність NotificationLogEntity реалізує журнал відправки, або ж помилки при відправці сповіщень по відповідному шляху, який визначається через поле type (Webhook або DiscordNotificatin). Запис у журналі може мати два різні

статуси, залежно від успішності виконання відправки (Sent або Failed). У випадки помилки, ми маємо записаний requestPayload та errorResponse. Для звичайних користувачів, ці полі не будуть надавати великої ролі, проте для адміністрування платформи, це допоможе визначати конкретні проблеми, які можуть виникнути в системі.

3.3 Реалізація API-сервісу

API-сервіс є центральним компонентом систем: приймає REST-запити від клієнтських застосунків, виконує автентифікацію користувачів, управляє підписками та сповіщеннями, а також обробляє вхідні RPC-виклики від Discord-бота. Для реалізації цього одночасно запускаються два транспорти: HTTP-сервер на порту 3000 та TCP-мікросервіс на порту 3010. Такий гібридний підхід, реалізований через app.connectMicroservice(), дозволяє одному процесу обслуговувати як зовнішніх REST-клієнтів, так і внутрішні синхронні виклики бота.

```
async function bootstrap() {
  const app = await NestFactory.create(ApiModule);
  const config = app.get(ConfigService<AppConfig>);

  const rabbitUrl = buildAmqpUrl(
    config.get('RABBIT_API_USER')!,
    config.get('RABBIT_API_PASSWORD')!,
    config.get('RABBIT_HOST')!,
    config.get('RABBIT_PORT')!,
    config.get('RABBIT_VHOST')!,
  );

  app.connectMicroservice<TcpOptions>({
    transport: Transport.TCP,
    options: {
      host: '0.0.0.0',
      port: Number(config.get('API_TCP_PORT')) ?? 3010,
    },
  });

  app.connectMicroservice<MicroserviceOptions>({
    transport: Transport.RMQ,
    options: {
      urls: [rabbitUrl],
      queue: Queues.TWITCH_SUBSCRIPTIONS,
      queueOptions: { durable: true },
      wildcards: true,
    },
  });
```

Рисунок 3.2 – Ініціалізація внутрішніх інтерфейсів

Сервіс побудований на основі класичної модульної архітектури для NestJS, з розподілом основних модулів таким чином, аби вони було спрямовані, як на Web підтримку комунікації, так і з прийманням запитів від Discord бота, який по суті працює, як проксі інтерфейс з цієї точки зору.

3.3.1 Модуль автентифікації

Модуль автентифікації реалізує JWT-автентифікацію з ротацією refresh-токенів та підтримує два методи входу: за email/паролем та через Discord OAuth 2.0. При успішному вході генерується пара токенів: короткостроковий access-токен та довгостроковий refresh-токен [26]. Access-токен містить поля sub (userId), sessionId та унікальний jti для точкового відкликання окремих сесій без інвалідації решти. Refresh-токен при кожному використанні замінюється новим – стара сесія видаляється з UserSession, нова записується. Реалізація стратегій спирається на Passport.js [19] з двома кастомними класами: JwtAccessStrategy та JwtRefreshStrategy.

Discord OAuth 2.0 реалізовано через пару ендпоінтів: POST /api/auth/discord (редирект до Discord) та GET /api/auth/discord/authorize (обробка OAuth callback). Обидва захищені FeatureGuard з перевіркою прапорця register-discord – це дозволяє вмикати та вимикати реєстрацію через Discord без змін у коді. Зв'язування наявного облікового запису з Discord виконується окремим ендпоінтом POST /api/auth/discord/link, який приймає OAuth code та оновлює існуючий User-запис зі збереженням зашифрованих токенів. Фактично цей ендпоінт можна видалити з систему, на користь існування прямої авторизації через Discord на пряму, але з точки зору користувацького використання – це надає можливість більш коректно оперувати очікуваннями користувача до відповідних дій.

```typescript
@ApiTags('Auth')
@Controller('api/auth')
export class AuthController {
  constructor(
    private readonly authService: AuthService,
    private readonly discordAuthService: DiscordAuthService,
    private readonly meService: MeService,
  ) {}

  @Post('login/email')
  @ApiGenericResponses({ [HttpStatus.CREATED]: AuthResponseDto })
  async login(@Body() data: LoginByEmailDto): Promise<IAuthResponse> {
    return await this.authService.loginByEmail(data);
  }

  @Post('register/email')
  @Feature('register-email')
  @UseGuards(FeatureGuard)
  @ApiGenericResponses({ [HttpStatus.CREATED]: AuthResponseDto })
  async registerByEmail(@Body() data: RegistrationByEmailDto): Promise<IAuthResponse> {
    return await this.authService.registerByEmail(data);
  }

  @Post('discord')
  @Feature('register-discord')
  @UseGuards(FeatureGuard)
  @ApiGenericResponses({ [HttpStatus.MOVED_PERMANENTLY]: String })
  async redirectForDiscordAuth(@Res() res) {
    const redirectUri = this.discordAuthService.getDiscordAuthRedirectUri();
    return res.redirect(redirectUri);
  }

  @Get('discord/authorize')
  @Feature('register-discord')
  @UseGuards(FeatureGuard)
```

Рисунок 3.3 – Роутінг авторизаційних процесів

У відповідність до того, чи користувач вже є в системі, чи його акаунт підлягає створенню, ми визначаємо, чи потрібно нам надавати користувачеві додаткові ролі. Разом з цим, якщо користувач ще відсутній в системі, через його поточний спосіб авторизації, але він вже має акаунт, який зареєстрований на вказану / прив’язану електрону адресу – ми авторизуємо користувача у вже існуючий акаунт, оновлючи його дані. Таким чином, якщо користувач зареєструвався в системі через пошту, з використанням паролю – то при спробі увійти в систему через Discord, акаунт якого прив’язаний до тієї ж електронної

адреси, ми оновимо дані, додаючи потрібні відповідно поля з Discord акаунту, і авторизуємо відповідно у вже існуючий акаунт.

```
sync authByDiscordCode(code: string): Promise<IAuthResponse> {
 const { accessToken, expiresIn, refreshToken, } = await this.discordAuthService.exch
 const discordMe = await this.discordApiService.fetchMeUserByToken(accessToken);

 let userData = await this.usersRepository.findOne({
   where: [
     { email: discordMe.email },
     { discordId: discordMe.id },
   ],
   relations: { roles: true },
 });

 if (userData) {
   if (userData.status === AccountStatus.BLOCKED) {
     throw new ForbiddenException('Account is blocked');
   }
   if (userData.discordId !== discordMe.id) {
     userData.discordId = discordMe.id;
   }
   userData.discordAccessToken = accessToken;
   userData.discordRefreshToken = refreshToken;
   userData.discordTokenExpiresAt = new Date(Date.now() + expiresIn * 1000);
   await this.usersRepository.save(userData);
 } else {
   const newUser = this.usersRepository.create({
     email: discordMe.email,
     displayName: discordMe.globalName,
     discordId: discordMe.id,
     discordAccessToken: accessToken,
     balance: BASE_BALANCE,
     discordRefreshToken: refreshToken,
     discordTokenExpiresAt: new Date(Date.now() + expiresIn * 1000),
     roles: [],
   });
   userData = await this.usersRepository.save(newUser);
 }
```

Рисунок 3.4 – Використання одного акаунту, з різними flow авторизації

3.3.2 Twitch модуль

Модуль Twitch групує сервіси, які відповідають за комунікацію з платформою Twitch, та відповідно реалізують управління різними аспектами внутрішньої інфраструктурою, яка пов'язана з комунікацію між системою та Twitch сервісом.

Відповідно до того, що ми маємо розширювальний підхід платформи, що дозволяє нам підключати багато Twitch застосунків, на які будуть виконуватися підписки Webhook -ів, ми маємо за потребу зберігати велику кількість Twitch Secret -ів, токенів доступу тощо. Зважаючи на це, та міркування безпеки, нами було реалізовано логіку шифрування та зберігання у базі даних вже шифрованого токену, та секретів, які ми використовуємо для комунікації з зовнішніми системами. Таким чином, аби створити універсальний підхід, де ми не будемо мати потреби вручну виконувати шифрування та дешифрування, було розроблено окрему надбудову над TypeORM Repository, яка інджектить у себе Crypto Service, що походить з локальної бібліотеки, і робить це автоматично, за відповідними опціями у запиті.

```
async findLeastLoaded(
  shouldDecrypt = false,
  decryptOptions: ITwitchAppEncryptOptionsModel = defaultEncryptOptions,
): Promise<TwitchAppEntity | null> {
  const app = await this
    .createQueryBuilder('app')
    .where('app.currentCost < app.maxCost')
    .orderBy('app.currentCost', 'ASC')
    .getOne();
  return (app && shouldDecrypt)
    ? this.decryptFields(app, decryptOptions)
    : app;
}
```

Рисунок 3.5 – Приклад методу з репозиторія TwitchApp

Зважаючи на те, що ми використовуємо декілька Twitch застосунків одночасно, ми виконуємо запити для пошукової інформації у відповідності до кожного окремого застосунку, що дозволяє нам бути певними у тому, що ми розподіляємо обов'язки між ними у відповідності до навантаження, яке ми зважуємо за рахунок витрачених кредитів системи Twitch. Тож відповідно до поставленої задачі, ми маємо структуровані сервіси у модулі, що виконують запити, відповідно до Twitch застосунку.

```
async getSubscriptions(
  clientId: string,
  options: IGetTwitchSubscriptionParamsModel
) {
  const app = await this.twitchAppsRepository.findOne(
    { where: { clientId, }, }, true, { accessToken: true,}
  );

  if (!app) {
    throw new InternalServerErrorException('Cannot find Twitch app for provided clientId');
  }

  try {
    const { data, } = await firstValueFrom(
      this.httpService.get<ITwitchGetSubscriptionsApiResponseModel>(
        `helix/eventsub/subscriptions`,
        <ITwitchHttpConfigModel> {
          params: {
            subscription_id: options.subscriptionIds,
            status: options.statuses,
            after: options.after,
            user_id: options.userIds,
          },
          twitchClientId: clientId,
          accessToken: app.accessToken,
        },
      ),
    );
    return data;
  } catch(e: unknown) {
    const { message, status } = <AxiosError> e;
    console.error(message);
    throw new InternalServerErrorException(
      `Twitch API subscription error. Please contact administrator for more information [${status}]`
    );
  }
}
```

Рисунок 3.6 – Метод виклику Twitch API із конкретним clientId

З одночасним використанням різних застосунків, ми маємо потенційну проблему, що токен, який був отриманий від Twitch, може закінчити своє існування. З цієї причини, HttpService, який є частиною модуля Twitch, має перехоплювач запитів, який відловлю кожен запит, та відповідь на нього, і приймає рішення, зважаючи на тіло відповіді та опції, які задаються при виклику з нашого сервісу. HttpService – працює як обгортка довкола клієнту Axios, тож відповідний перехоплювач встановлюється саме на клієнт Axios.

```
})
export class TwitchModule implements OnModuleInit {
  constructor(
    private readonly httpService: HttpService,
    private readonly interceptor: TwitchAuthInterceptor,
  ) {}

  onModuleInit() {
    this.interceptor.createAxiosInterceptors(this.httpService);
  }
}
```

Рисунок 3.7 – Реєстрація перехоплювача запитів у Twitch модулі

Перехоплювач запитів контролює два процеси: відправка запитів по HTTP та отримання відповіді.

У рамках відправки запитів, ми можемо передати новий параметр, який буде перетворений у коректний хедер. Тобто ми передаємо у запит ClientId, а перехоплювач відловлює це, шукає відповідний застосунок по ClientId, та дістає з нього токен доступу Twitch, який він потім використовує у хедері запиту, через авторизаційну стратегію Bearer.

Процес перехоплення відповіді, в основі своїй включає перевірку статусу відповіді. Якщо статус при відповіді з помилкою є 401 – то ми проводимо процедуру отримання нового Twitch токену, за рахунок обміну секрету застосунку. Цей процес у нас проводиться відповідно через окремий сервіс у

контексті Twitch модуля, який відповідає виключно за авторизації Twitch застосунків.

```
createAxiosInterceptors(httpService: HttpService) {
  const axios = httpService.axiosRef;

  axios.interceptors.request.use(async (config: any) => {
    const clientId = (<ITwitchHttpConfigModel> config).twitchClientId;
    if (!clientId) {
      return config;
    }

    const token = await this.twitchAuthService.getTokenForApp(clientId);
    config.headers['Authorization'] = `Bearer ${token}`;
    config.headers['Client-Id'] = clientId;
    return config;
  });

  axios.interceptors.response.use(
    (response) => response,
    async (error: AxiosError) => {
      const originalRequest = error.config as any;

      if (error.response?.status === 401 && !originalRequest._retry) {
        originalRequest._retry = true;

        const clientId = originalRequest.twitchClientId;
        const token = await this.twitchAuthService.getTokenForApp(clientId, true);
        originalRequest.headers['Authorization'] = `Bearer ${token}`;

        return axios(originalRequest);
      }

      return Promise.reject(error);
    },
  );
}
```

Рисунок 3.8 – Перехоплювач запитів Twitch модуля

Для створення запитів, як вже раніше було зазначено, ми використовуємо токени, які ми зберігаємо у зашифрованому стані. Процес дешифрування сам по собі у концепті займає певний час та ресурси. Беручи до уваги можливість того, що ми можемо робити одночасно по декілька паралельних, або послідовних, запитів з потребою використання одного й того ж токену доступу

– ми маємо потенційну проблему у затримці виконанні запитів. Для цього, у нас є окремий менеджер токенів, який виконує роль централізованого управління наявними токенами, та зберігає у пам’яті нещодавно дешифровані, що врази облегшує нам операцію їх отримання.

3.3.3 Модуль підписок

У системі присутнє розділене представлення «підписок» та «сповіщень». Підписку відповідають за створення записів у базі даних, у відповідності до того, як ми створюємо підписку на Webhook Event зі сторони Twitch. Якщо це розглядати за рівнем ієрархії дій, то ми отримуємо, що верхній рівень є сповіщення, які є фактичною вказівкою куди направляти повідомлення про подію, після йде підписки, які є структурною одиницею запису у системі, що поєднує нас з останнім рівнем, Twitch -ем, який контролює якраз фактичну наявність цих підписок, та за потребою відкликає їх, відправляючи нам відповідний запит, на інший наш сервіс (Webhook Reciever).

Відповідно на поточний момент часу, у системі присутні виключно підписки для Twitch подій. Модуль представляє з себе два сервіси: менеджмент підписок і крон сервіс, для синхронізації потенційно не коректних підписок.

Сервіс управління підписками, надає користувачеві можливість переглядати підписки, які записані у нашій базі даних, додавати нові, видаляти та редагувати існуючі. Разом з цим, ми надаємо можливість переглядати підписки, які існують виключно на стороні Twitch застосунку. З технічного розрахунку, ці підписки мають бути у відповідному синхронному зв’язку, тож фактично нам має вистачати перевірки локальних підписок, проте для перевірки фактичного стану на Twitch стороні, та з метою тестування у середовищі розробки – це цілком стає доцільним доповненням.

Створення нових підписок відбувається у декілька етапів. Першочергово, ми викликаємо наш метод з сервісу, який відповідає за реєстрацію підписок. Він

у свою чергу, створює нову сутність підписки, наповнюючи її ключовими значеннями, які походять з Twitch App переважно. Після створення сутності, ми робимо запит через Twitch модуль, до самої Twitch платформи, і отримуємо ідентифікатор підписки у системі Twitch, який ми додаєм до нашого запису. Запит до Twitch у свою чергу викликає Challenge запит до нашого іншого мікросервісу (Webhook Receiver), який має підтвердити активацію підписки, та додати цю дію у чергу на оновлення стану підписки.

```
const createdEvent = await this.twitchStreamerEventsRepository.save(newEvent);

const webhookBaseUrl = this.configService.get<string>('TWITCH_WEBHOOK_URL');

const twtichResData = await this.twitchApiService.registerEvent(
  app.clientId,
  `${webhookBaseUrl}/${app.clientId}/${createdEvent.id}`,
  app.webhookSecret,
  broadcasterId
);

const subscriptionId = twtichResData.data[0].id;

// Note: do not use "save" method in case we can overwrite status change etc
await this.twitchStreamerEventsRepository.update(
  { id: createdEvent.id, },
  { subscriptionId, }
);
createdEvent.subscriptionId = subscriptionId;

return createdEvent;
```

Рисунок 3.9 – Основний алгоритм створення підписок

Аби створену підписку можна було знайти, при запиті від Twitch, для підтвердження статусу – ми використовуємо шлях, який містить у собі конкретні властивості, для розпізнавання нашої сутності у майбутньому, з причини того, що ми маємо за потребу підтвердити підпис запиту, через спеціальний Webhook Secret, який є унікальним для кожного Twtich застосунку.

Під час створення нашої підписки, ми виставляємо за замовченням статус «Pendding», який потім має мінятися на «Verified», або ж «Revoked», залежно від успішності підтвердження реєстрації підписки. Проте існує дрібна ймовірність, що до прикладу наш сервіс (Webhook Receiver) – буде не доступним, або ж повідомлення у черзі з якоїсь причини не дійде до нашого головного API сервісу. З розрахунку такої потенційної ситуації, ми маємо реалізований Cron сервіс, який має на меті перевірку статусів підписок, які все ще знаходяться у стані «Pendding».

Цей сервіс відпрацьовує кожну годину часу, і перевіряє виключно ті підписки, статус яких все ще не змінився з початкового. Для таких випадків, наш алгоритм дій полягає у тому, що ми групуємо наші пдіписки за clientId, який відповідає Twitch застосунку, на якому вони зареєстровані, та робимо один груповий запит з усіма відповідними ідентифікаторами зі системі Twitch. Як результат, ми або видаляємо підписку, а разом з цим каскадною операцією і сповіщення, або ж перекосимо підписку у статус активної, чи вимкненої.

3.3.4 Модуль сповіщень

Модуль сповіщень реалізує CRUD-операції, для управління сповіщеннями. Як вже було раніше сказано, сповіщення виступають найвищою ланкою у ланцюгу створення повідомлень на події. Основним елементом цього модуля є функціонал створення сповіщень, та їх валідація. На поточний момент часу, у системі доступні виключно підписки для подій від Twitch платформи. Ці сповіщення поділяються на фактично два типи: Discord та Webhook, які в свою чергу теж можеть бути частиною Discord сповіщень.

Основним порядком дій при створенні сповіщень – йде валідація на доступність технічного характеру, для створення сповіщення, валідація на доступність з точки зору балансу користувача, або ж спільноти, залежно від

типу перекладання відповідальності за оплату, і після цього йде ряд викликів інших модулів, з якими ми вже ознайомлені раніше (Twitch та Підписки).

Коли ми кажемо, про створення сповіщень через Webhook – то ми маємо два варіанти, або створити її незалежно від будь-яких прив'язок, або ж створити з підключенням до конкретного Discord сервера. Якщо ми використовуємо підхід зі створенням для Discord сервера – ми виконуємо ще один запит на вимогу перевірки, чи дійсно Webhook відноситься до Discord сервера, про якого йдеться мова у сповіщенні.

Рисунок 3.10 – Приклад відповіді Discord Webhook -а

На поточний момент часу, у системі присутні дві варіації створення підписок, які відрізняються у підходах в «оплаті». Мова йде не про реальні гроші, а про баланси умовних кредитів. Кожен користувач має певний Баланс з самого початку після реєстрації у системі. Кожна підписка у системі – коштує 1 кредит, який відповідно може бути списаний з балансу користувача, або ж за наявності, з балансу Discord спільноти. Таким чином, ми зараз кажемо за те, що ми можемо розділяти ці баланси, з метою підтримки окремих спільнот, і обмеження без розсудливих сповіщень.

3.3.5 Discord модуль

Модуль відповідальний за підключення та комунікацію між Discord-ом, з метою отримання інформації від користувачів, та від нашого Discord бота.

У якості реалізованого сервісу, ми у першу чергу маємо зазначити сервіс, який відповідає за підтримку авторизації, та обробку запитів з токенами користувачів. Основна проблема полягає у тому, що від користувача ми отримуємо Access та Refresh токени, які мають тривалість життя врази меншу, на відміну від відповідно токену, який ми отримуємо по нашому Twitch застосунку. На результат використання у межах нашої платформи, ми не маємо за потребу тримати це підключення на абсолютно постійній основі активним, тож ми це виконуємо на вимогу. Тобто якщо користувач буде виконувати запити, які потребують від системи використання Discord токенів, а вони у свою чергу будуть застарілими, то ми віддаємо спеціальні Exception -и, які інформують користувача, що він має за потребу повторну авторизуватися через Discord. Або ж, якщо у нас застарів виключно Access токен – то ми самостійно його оновимо за допомогою Refresh токену, та збережемо нову пару.

Зважаючи на те, що ми не можемо вирахувати потенційну кількість користувачів, які можуть бути зацікавленими у використанні системи – ми не можемо прогнозувати навантаження, у разі виконання оновлення токенів через Discord API за рахунок Cron інтервалів. Тож на поточний момент часу, рішення виконувати оновлення на вимогу – виступає оптимальним рішенням, хоч і більш тимчасовим, аніж постійним.

Іншим ключовим функціоналом модуля є можливість отримувати від самого Discord інформацію, про наявні канали, ролі та права користувачів на серверах, як з боку користувача, так і зі сторони нашого бота. З певної точки зору, ми хочемо бути певні у тому, що наш бот все ще присутній на сервері, аби мати розуміння, що ми завжди будемо мати контроль над тим, що ми зможемо отримати інформацію.

Важливою інформацією, яку ми отримуємо є ролі користувача на сервері, та права користувача на відповідних серверах. За замовченням, наш користувач має можливість додавати, чи редагувати сповіщення, за умови права Адміністратора спільноти, що є найвищим правом на сервері Discord. Аби облегшити модерацію у цьому, для кожного сервера, можна визначити додатково ще роль, яка буде виступати ключем для перевірки права користувача, на управління сповіщеннями для сервера. На поточний момент часу, у нас нема реалізації для розподілу обов'язків у відповідності до ролі (на Discord сервері), та конкретних прав, які можуть виконувати користувачі на платформі відносно сервера, проте це могло б стати цілком доцільним доповненням у майбутньому.

Як вже було раніше зазначено, ми маємо структуру даних у системі, що відповідає Discord серверам, які ми маємо за записи у нашій базі дних. Ці записи надують нам можливість керувати якраз ролями для користувача, сповіщеннями тощо. Значною мірою, ця структура даних є просто контактом, між сповіщеннями, та призначенням для Discord.

3.3.6 Discord Handler модуль

У зв'язку зі раніше визначеною задачею виконання ролі інтерфейсу з точки зору Discord Бота – було прийнято рішення реалізувати це саме у форматі посередника, аналогічно до того, як ми використовуємо наш Web застосунок з UI складовою, аби взаємодіяти з нашою системою. У рамках абстрагування Бота від доступу до основної частини системи, розроблено формат обміну повідомленнями через RPC, де API сервіс виступає серверною частиною у цих відносинах, що у результаті дає йому змогу приймати запити від Discord Бота, та виконувати відповідні операції. Запити з Discord передають нам відразу вже всі потрібні нам авторизаційні параметри, у вигляді discordId, тож ми можемо оперувати вже з цим, але наша система за основу визначення персоналії

користувача потребую мати акаунт з електронною адресою, тож якщо не буде знайдено користувача зі зазначеним discordId – буде надіслано відповідь, з потребою зареєструватися у системі через Discord Flow, яким ми вже користуємося у системі, якщо працювати просто через Web.

Основним функціоналом цього модуля є надання можливості керувати сповіщеннями, балансом та серверами. Це надає більшу можливість і простіший формат комунікації, за рахунок того, що ми вже маємо Discord пряму підключення, яке не потребує наших додаткових запитів, а буде працювати у форматі «під капотом», і надавати вже пряму інформацію серверу.

Відповідно до кожного запиту, ми маємо відповідний патерн, за яким ми будемо відловлювати запити по RPC. Ці патерни будуть працювати так само як і звичайні шляхи, коли ми робимо HTTP запит, проте у цьому контексті, запити будуть йти через TCP.

```
@MessagePattern(RpcPatterns.discord.addDestination)
async handle(@RpcPayload() data: AddDestinationDto): Promise<IAddDestinationResult> {
  const user = await this.usersRepository.findOne({
    where: { discordId: data.discordId },
  });

  if (!user) {
    throw new RpcBusinessException(
      RpcError.NOT_REGISTERED,
      'User is not registered in the system',
      { authUrl: this.discordAuthService.getDiscordAuthRedirectUri() },
    );
  }
```

Рисунок 3.11 – Handler запитів від Discord по RPC

Відповідно до зазначеної раніше дії в отриманні RPC запитів, ми маємо окремо підключений інтерфейс, який буде слухати внутрішню систему. У майбутньому, це буде використано на нашу користь, з точки зору розгортання сервісу на одному хостінгу, та в одному Docker Compose серидовищі.

```
app.connectMicroservice<TcpOptions>({
  transport: Transport.TCP,
  options: {
    host: '0.0.0.0',
    port: Number(config.get('API_TCP_PORT')) ?? 3010,
  },
});
```

Рисунок 3.12 – Підключення інтерфейсу для RPC повідомлень

3.3.7 Модуль журналу сповіщень

Цілком очікуваним інструментом у платформі, де ми маємо події, пов'язані з зовнішніми сервісами мати за потребу інструменти моніторингу. Основним інструментом, який має бути у будь-якій системі – має бути логування, або ж у конкретиці нашого варіанту «Журнал сповіщень». Основною функціональною ціллю модуля є отримання переліку сповіщень, їх статус (успішний або ж помилка) і разом з цим вивід додаткової інформації, яка потрібна виключно для адміністраторів системи. Створення нових записів у журналі відбувається в іншому сервісі системи, атож у контексті цього модуля розглядаються переважно READ операції, над вже існуючими записами.

3.4 Реалізація сервісу прийому вебхуків

З метою розподілу обов'язків, та потенційного масштабування у більш коректний масштабах для майбутнього – було створено мікросервіс, основна задача якого полягає у прийомі запитів від Twitch.

Першочерговим форматом комунікації є підтвердження реєстрації підписок на події, що у подальшому буде призводити до Webhook запитів. Реєстрація події відбувається спочатку у сервісі API, який виконує роль ініціатора, виконуючи запит до Twitch платформи, і вказуючи, який саме шлях

потрібно використати Twitch, для підтвердження реєстрації платформи. Цей шлях, на далі буде використовуватися також для отримання сповіщень про нові події, та про потенційне відкликання підписок.

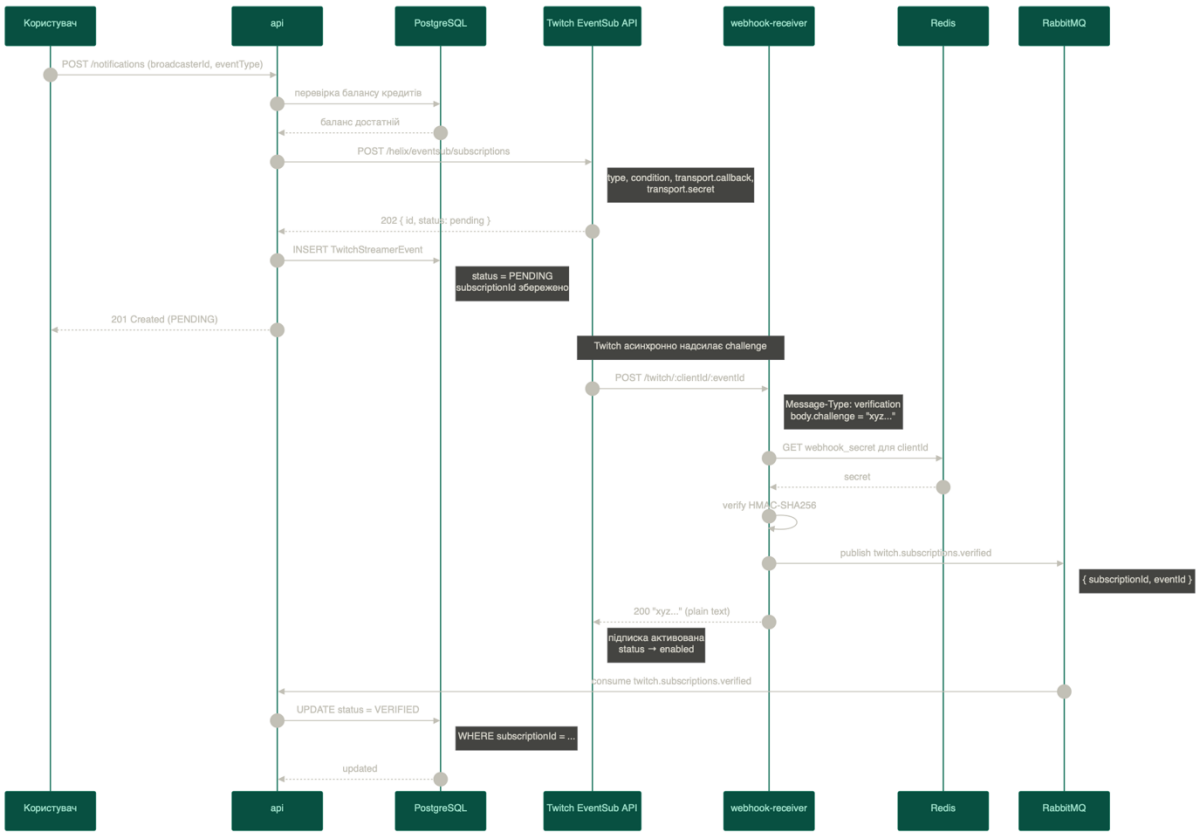


Рисунок 3.13 – Послідовність верифікації EventSub-підписки через Challenge-Response

Відповідно другим важливим функціонал, який має при собі Webhook Receiver – це отримання та делегація сповіщень у відповідні черги RabbitMQ, які пізніше зчитаються воркером, і як результат виконуються. Себто наш Webhook Receiver виступає певним проксі, який першим ловить сповіщення про події, та валідує їх, з метою уникнення дублікатів, не справжніх сповіщень тощо. Основний алгоритм дій розгортається довкола валідації повідомлення,

що ми отримуємо від Twitch. Ми маємо окремий секрет, для кожного Twitch застосунку, який ми зберігаємо у базі даних у зашифрованому вигляді, і використовуємо його, для підтвердження, що підпис у заголовку дійсно відповідає тому, що ми отримуємо з нашим секретом. Оскільки Twitch чутливий до очікування відповіді на запити, а запитів якщо за короткий проміжок часу приходить занадто багато – ми маємо за потребу швидше діставати секрет, аніж постійно виконувати дешифрування. У відповідності до цього, ми зберігаємо у Redis дешифрований секрет, який має TTL в 1 годину, після чого нам потрібно буде виконувати по новому дешифрування, та класти його у Redis. Отримання секрету з Redis врази пришвидшує відповідь, та спрощує нам потік обробки повідомлення, бо ми не будемо мати на потребі використовувати одночасно більше ніж нам потрібно ресурсів.

Після підтвердження валідності запиту, ми перевіряємо Тип запиту, і направляємо його у коректну обробку. Як ми вже казали до цього, у нас присутній один шлях, і для підтвердження підписок, і для деактивації підписок зі сторони Twitch, і відповідно для отримання події.

При отриманні події, ми як вже було сказано раніше, валідуємо запит, далі, ми перевіряємо за ідентифікатором повідомлення (MessageId надається Twitch), чи не отримували ми це повідомлення протягом останніх 20 хвилин. Якщо ж отримували – то це дублікат повідомлення. Це може траплятися, коли Twitch не певен, чи дійсно ми отримали повідомлення. У таких випадках, ми просто відповідаємо статусом 200, після чого закінічуємо обробку. Якщо ж події ми не отримували ще – то ми додаємо її до списку нещодавно отриманих, та передаємо її в RabbitMq чергу, яка далі буде зчитуватися іншим нашим сервісом, наповнюватися та делегуватися у наступні потоки.

Черги у контексті публікації подій – розділені на стадії, відповдіно до того, який сервіс має їх отримати, чи який сервіс їх віддає. На етапі публікації повідомлення у чергу зі сторони Webhook Receiver -а, ми публікуємо

повідомлення у чергу «stream.events.online» (online – це конкретний приклад з публікацією на подію про початок трансляції).

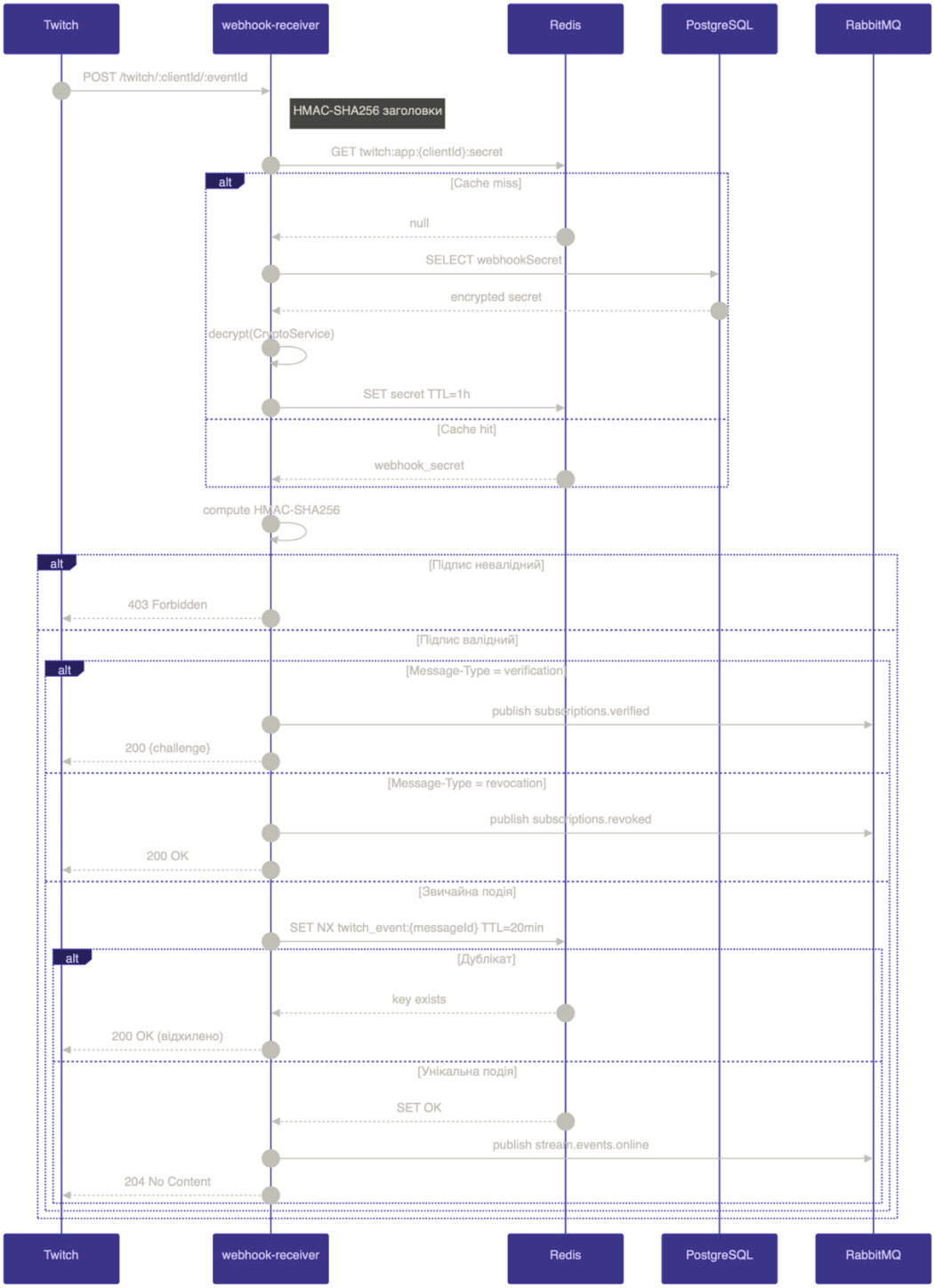


Рисунок 3.14 – Послідовність обробки вхідного webhook-запиту від Twitch EventSub

3.5 Реалізація сервісу обробки сповіщень

Сервіс notification-worker є RabbitMQ-консюмером, що виконує бізнес-логіку диспетчеризації подій до кінцевих призначень. Основною задачою сервісу на поточний час, є прослухвання черги «stream.events.*». Звідти сервіс отримує повідомлення, що містить базову інформацію про подію, та про виконавця події. Сервіс має доступ до бази даних, що дає йому можливість переглядати існуючі підписки та сповіщення, і відповідно направляти їх у правильний контекст.

Як тільки сервіс отримує розуміння, що він має зробити з підписками – він виконує «Acknowledgement» у відповідь, що позначає повідомлення виконаним, і не залишить його у черзі на повторні спроби, як могло б бути за умови виникнення помилок зі сторони сервісу, або ж його недоступності до бази даних тощо. Якщо ж сервіс отримав помилку під час пошуку сповіщень, чи просто з не відомих нам причин – він відповідає «Negative Acknowledgement».

Після отримання списку сповіщень, які має надіслати сервіс – він будує об’єкт, який буде використовуватися для заміни інформації у сповіщенні (вставка назви трансляції, користувача чи час початку). Після цього формування, сервіс виконує груповані виклики для Webhook, або ж додає у чергу для Discord бота, вже наповнені контентом сповіщення, які йому залишається виключно відправити у якості повідомлення.

3.6 Реалізація сервісу Discord-бота

Окремий сервіс у форматі Discord бота, який побудований за рахунок архітектури Nestjs та бібліотеки discordjs. Загальною мірою, він відповідає за створення ще одного комунікатора між користувачем та системою, разом з цим

виконуючи не менш важливу роль у якості відправки сповіщень, саме через клієнт бота, за рахунок присутності на сервері.

Присутність бота на сервері є обов’язковою умовою для налаштування сповіщень для цього конкретного сервера. Разом з цим, бот може замінити Web панель, за винятком потреби виконати початкову авторизацію у систему, з метою створення акаунту та запису на конкертний email. У потенційному форматі, цей крок можна було б спростити, проте це могло б створити менш надійний зв’язок, та надати користувачу більше можливості обійти основний захист системи, створюючі більше акаунтів, та маючи більший базовий баланс, аніж він мав би мати у відповідності до своїх налаштувань.

Окрім комунікатора, бот також відіграє важливу роль з точки зору отримання інформації про сервер, що допомагає збирати інформацію, навіть за відсутності користувача-комунікатора активно авторизованим у нашій системі.

Відповідно до обов’язків потенційного комунікатора, бота дає можливість налаштовувати підписки, створювати нові, та може створювати за потребою самостійні Webhook, з метоб облегшення роботи користувача.

3.7 Реалізація веб-застосунку

Клієнтська частина системи реалізована як окремий репозиторій на базі Next.js з App Router та розгортається незалежно від серверної частини на порту 4000. Зв'язок з API здійснюється виключно через змінну середовища NEXT_PUBLIC_API_URL, що дозволяє розгортати фронтенд та бекенд на різних хостах без змін у коді.

3.7.1 Структура застосунку та маршрутизація

Файлова структура застосунку організована відповідно до конвенцій Next.js App Router з чітким розмежуванням між публічною та захищеною зонами.

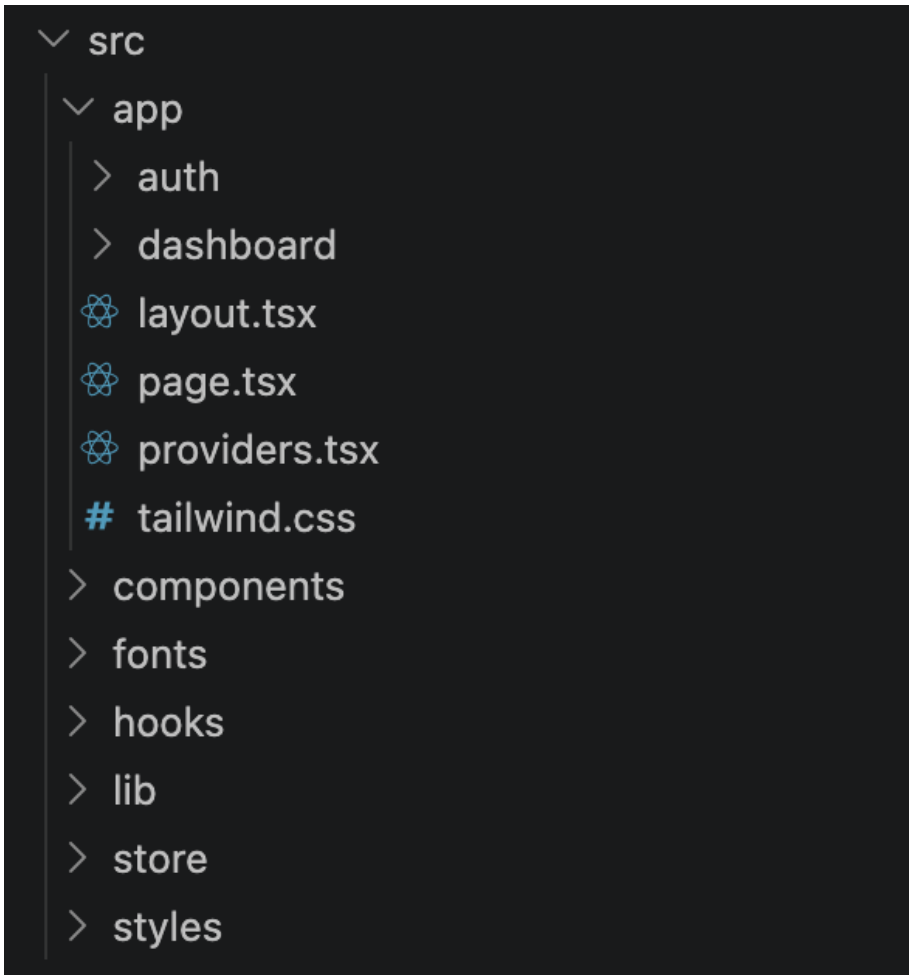


Рисунок 3.15 – Структура веб-застосунку

Маршрутизація поділена на дві зони доступу. Публічна зона (/auth) містить сторінки входу та реєстрації з вкладками Sign In / Sign Up, сторінку ініціалізації Discord OAuth-flow та callback-сторінку обробки отриманого авторизаційного коду. Захищена зона (/dashboard) включає головний огляд зі статистичними картками, сторінки управління Discord-серверами, сповіщеннями, ролями, Twitch-додатками та підписками.

Таблиця 3.1 – Перелік маршрутів веб-застосунку

Маршрут	Призначення
Публічна зона – /auth	
/auth	Сторінка входу та реєстрації (вкладки Sign In / Sign Up)
/auth/discord	Ініціювання Discord OAuth-flow, перенаправлення до Discord
/auth/discord/callback	Обробка ?code= від Discord, обмін на токени системи
Захищена зона – /dashboard	
/dashboard	Головна сторінка: статистичні картки та огляд системи
/dashboard/discord	Головна сторінка: статистичні картки та огляд системи
/dashboard/discord/[guildId]	Управління сервером: вкладки Notifications та Management
/dashboard/notifications	Усі сповіщення (Bot та Webhook) без фільтрації по сервері
/dashboard/roles	CRUD ролей з управлінням правами доступу (RBAC)
/dashboard/twitch-apps	CRUD Twitch-додатків (clientId, clientSecret, webhookSecret)
/dashboard/twitch-subscriptions	Локальні та raw підписки Twitch EventSub (адмін-панель)

Захист маршрутів реалізований через компонент AuthGuard, що обгортає відповідні layout-и. Для публічних маршрутів встановлюється прапорець guestOnly: true – автентифікований користувач автоматично перенаправляється

до /dashboard. Для захищених маршрутів перенаправлення відбувається у зворотному напрямку. Під час початкової ініціалізації, коли статус автентифікації ще невизначений ('init'), компонент відображає індикатор завантаження без рендерингу захищеного контенту.

3.7.2 Система автентифікації

Стан автентифікації управляється через Redux-slice з трьома можливими значеннями: 'init' (початковий стан до читання localStorage), 'guest' (токен відсутній або очищений) та 'authenticated' (валідний access token присутній).

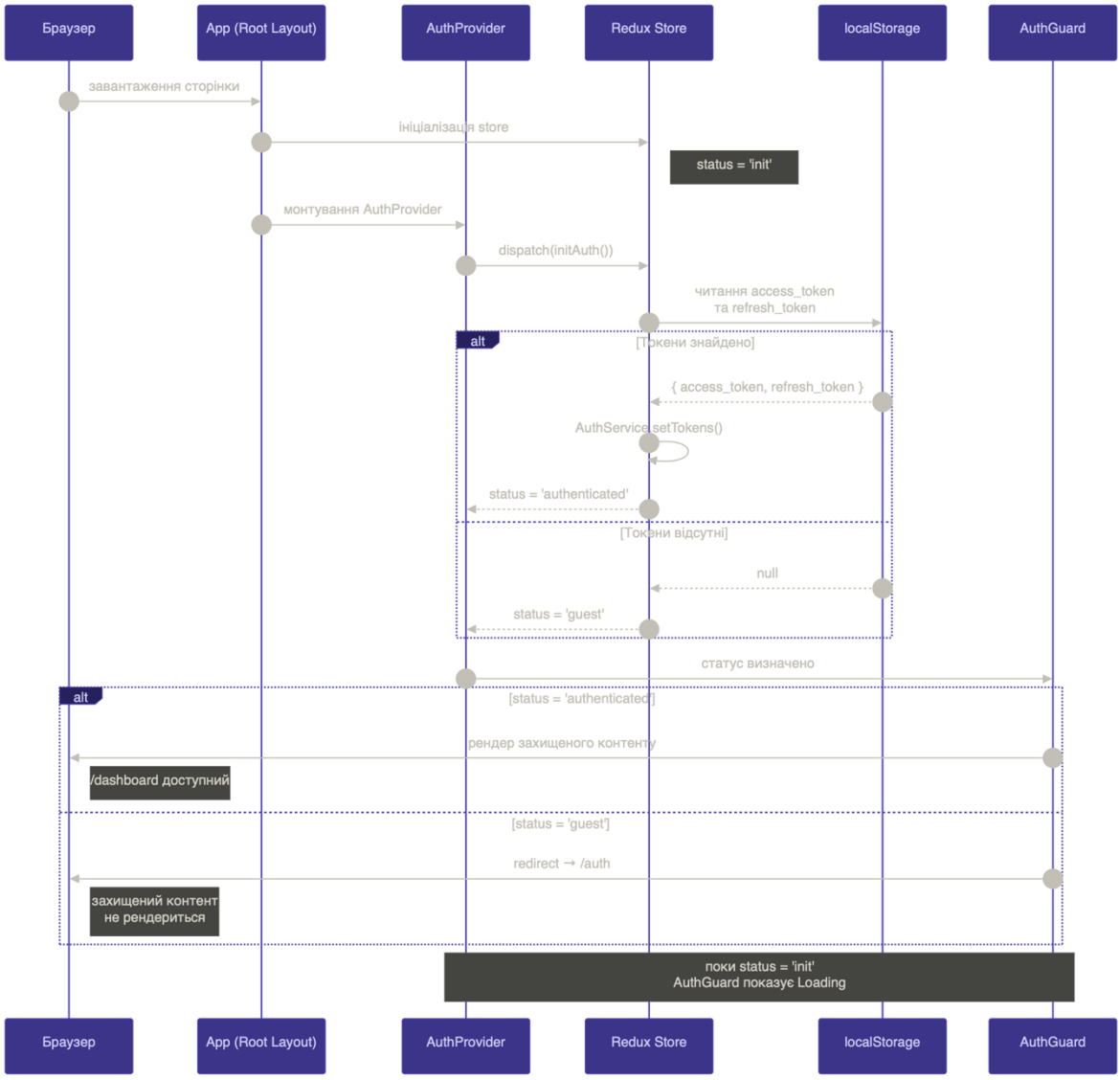


Рисунок 3.16 – Послідовність ініціалізації автентифікаційного стану

При першому завантаженні застосунку компонент AuthProvider викликає дію initAuth(), що зчитує токени з localStorage та встановлює відповідний статус у Redux store. Access token та refresh token зберігаються у localStorage і доступні через статичні методи класу AuthService. Такий підхід забезпечує збереження сесії між перезавантаженнями сторінки.

Discord OAuth 2.0 реалізований через Authorization Code Flow [27]: сторінка /auth/discord ініціює перенаправлення до Discord, після чого платформа повертає авторизаційний code на callback-сторінку /auth/discord/callback. Callback-сторінка загорнута у React <Suspense>, а вкладений компонент зчитує code через useSearchParams() та виконує мутацію для обміну коду на токени системи. Після успішного отримання токенів виконується dispatch(login(...)) та відбувається перенаправлення до /dashboard.

3.7.3 Система автентифікації

Усі HTTP-запити виконуються через централізований Axios-клієнт [16], визначений у src/lib/api-client.ts. Клієнт налаштовано з трьома інтерцепторами: request interceptor додає заголовок Authorization: Bearer <token> до кожного запиту; перший response interceptor розгортає вкладене поле response.data.data ?? response.data, завдяки чому всі сервіси повертають вже розгорнуті дані без додаткового розпакування у компонентах; другий response interceptor обробляє відповіді зі статусом 401.

Механізм автоматичного оновлення токенів реалізує захист від race condition при паралельних запитах [32]: перший запит із відповіддю 401 встановлює прапорець isRefreshing = true та виконує POST /api/auth/refresh з refresh token у заголовку; усі наступні 401-запити, що надходять під час оновлення, додаються до черги queue[] і призупиняються. Після успішного оновлення токенів черга розбирається та всі відкладені запити виконуються

повторно з новим access token. У разі помилки оновлення токени очищуються та користувач перенаправляється на сторінку входу.

Сервісний шар (src/lib/services/) повністю ізольований від компонентів і містить окремі класи для кожного домену: AuthService, DiscordService, RolesService, TwitchAppService, TwitchSubscriptionService та NotificationLogsService. Кожен сервіс визначає enum QueryKey для ідентифікації кешованих запитів у React Query.

3.7.4 Управління станом

Управління станом у застосунку базується на розмежуванні між двома категоріями даних з принципово різними характеристиками [14] [15].

Redux Toolkit відповідає виключно за синхронний стан автентифікації через єдиний authSlice. Slice містить поле status та три дії: initAuth() для ініціалізації при завантаженні, login() для встановлення автентифікованого стану та logout() для очищення. Мінімальний обсяг Redux-стану є свідомим архітектурним рішенням: уся решта даних керується TanStack Query, що усуває надмірність та запобігає дублюванню серверного стану у клієнтському сховищі.

TanStack Query v5 [15] відповідає за весь серверний асинхронний стан. Хуки запитів (queries) та мутацій організовані у відповідних директоріях і покривають усі домени застосунку. Конфігурація QueryClient визначає глобальні параметри кешування: staleTime встановлює інтервал, протягом якого дані вважаються актуальними, після чого при наступному зверненні виконується фонове оновлення без блокування UI. Після виконання мутацій відповідні QueryKey інвалідуються, що забезпечує автоматичне оновлення залежних компонентів.

3.7.5 Ключові компоненти

Компонент DashboardShell є головною оболонкою панелі управління та містить три дочірніх компоненти: DashboardSidebar з навігаційними посиланнями до всіх розділів, DashboardTopBar з UserCard поточного користувача та DashboardMobileNav для мобільних пристроїв. UserCard відкриває ProfileModal з трьома вкладками: оновлення профілю, управління підключеними платформами (прив'язка/від'язка Discord) та перегляд активних сесій з можливістю їх відкликання.

```
const DashboardShell = ({ children }: PropsWithChildren) => {
  return (
    <div className={styles.shell}>
      <DashboardBgEffects  />

      <DashboardSidebar />
      <div className={styles.content}>
        <DashboardTopBar />
        <main className={styles.main}>{children}</main>
      </div>

      <DashboardMobileNav />
    </div>
  );
}
```

Рисунок 3.17 – Структура компонентів DashboardShell

Компонент AddNotificationModal реалізує форму створення сповіщення та складається з чотирьох підкомпонентів: StreamerSearchField виконує пошук Twitch-каналів через GET /api/twitch/channels?search= з debounce-затримкою; ChannelSearchField завантажує список текстових каналів обраного Discord-сервера; RoleMentionPicker дозволяє обрати Discord-роль для mention у повідомленні; PayloadSection надає поля для кастомізації вмісту сповіщення.

Валідація форми реалізована через React Hook Form у поєднанні з Zod-схемами [25].

Компонент DiscordReconnectBlocker відображає overlay при виявленні помилок DISCORD_NOT_CONNECTED або DISCORD_TOKEN_REVOKED у відповідях API, блокуючи взаємодію з захищеним контентом до повторного підключення Discord-акаунту.

Компонент RawTwitchSubscriptionsTable надає адміністративний перегляд підписок безпосередньо від Twitch API з cursor-based пагінацією. Поле isOrphaned позначає підписки, які існують у Twitch, але не мають відповідного запису у локальній базі даних системи – такі підписки можуть бути видалені для звільнення cost-квоти додатку.

3.7.6 Система стилізації

Стилізація застосунку побудована на трьох рівнях. Tailwind CSS [18] є основним інструментом для utility-класів та типових патернів позиціонування і відступів. SCSS Modules застосовуються для компонентної логіки зі станами (:hover, :focus-visible, [data-state="open"]), анімацій через @keyframes та псевдоелементів – там, де декларативний CSS є виразнішим за набір атомарних класів. Глобальні стилі (src/styles/global.scss) та SCSS-абстракції (src/styles/abstracts/) визначають базові змінні та міксини, доступні у всіх модулях.

Утиліта cn() на основі clsx та tailwind-merge використовується для умовного поєднання класів та усунення конфліктів між Tailwind-утилітами. Shadcn/ui компоненти (Badge, Dialog, Table) застосовуються у деяких модулях як headless-основа [17]. Анімації реалізовані через Framer Motion для auth-секції та фонових ефектів. Типографіка базується на п'яти шрифтах з різним призначенням: DM Sans як основний, Manrope та Space Grotesk для акцентів,

JetBrains Mono для відображення коду та технічних ідентифікаторів, Syne для декоративних елементів.

3.8 Розгортання застосунків

3.8.1 Процес розгортання серверної частини

Розгортання серверної частини системи автоматизовано засобами GitHub Actions та реалізується у два послідовних етапи: збірка Docker-образів та їх розгортання на production-сервері. Хостинг системи здійснюється на платформі Hostinger [24], що надає виділений VPS із підтримкою Docker.

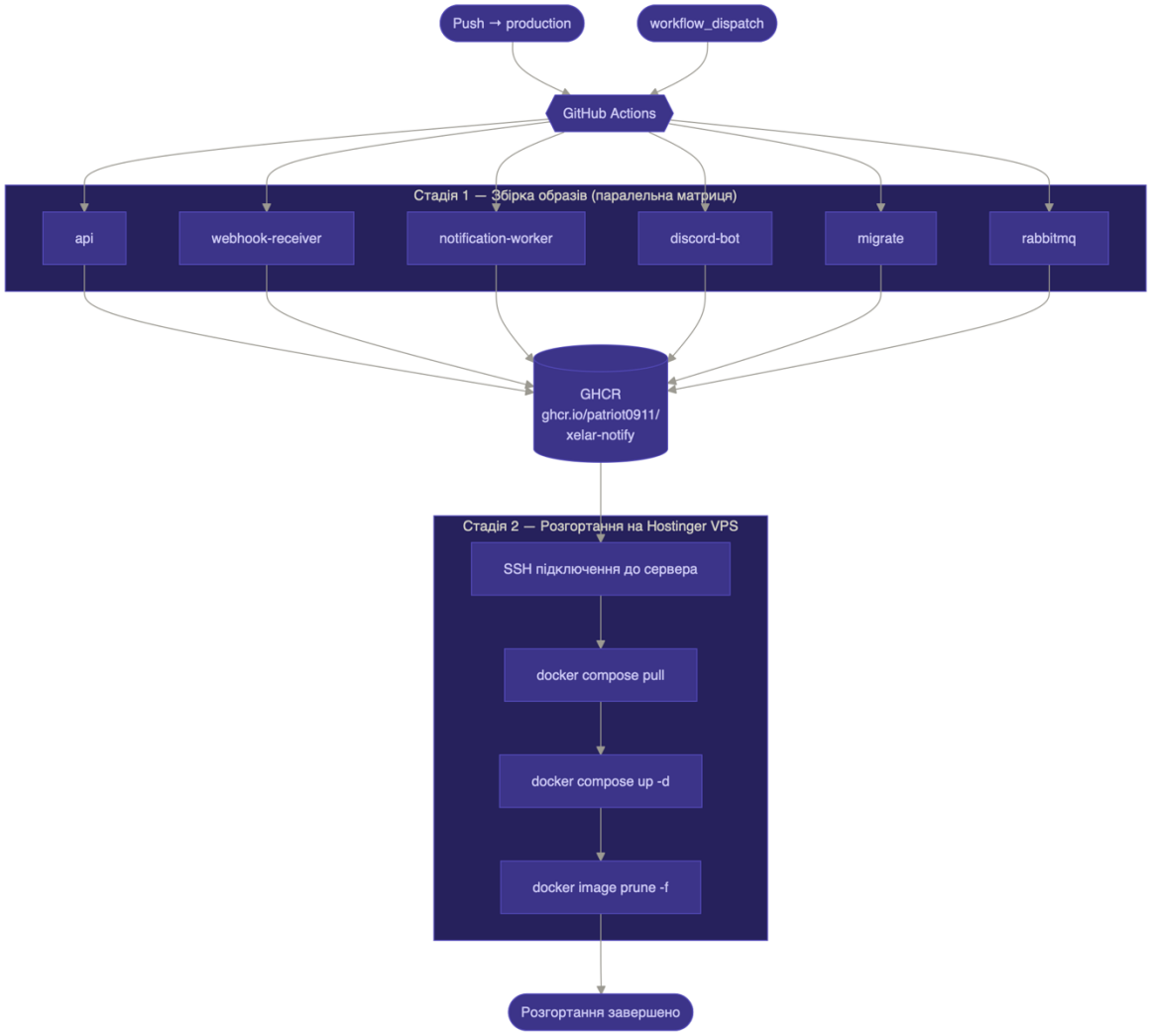


Рисунок 3.18 – Схема CI/CD пайплайну серверної частини

Конфігурація CI/CD. Пайплайн визначений у файлі .github/workflows/deploy.yml та запускається автоматично при push до гілки production або вручну через workflow_dispatch. Процес складається з двох незалежних стадій.

На першій стадії виконується паралельна збірка шести Docker-образів через матрицю (strategy.matrix): чотири образи мікросервісів (api, webhook-receiver, notification-worker, discord-bot), образ для виконання міграцій (migrate) та образ RabbitMQ з init-скриптом. Кожен образ публікується до GitHub Container Registry за адресою ghcr.io/patriot0911/xelar-notify/{app}:latest. Для прискорення повторних збірок використовується GitHub Actions Cache з окремим scope для кожного образу.

На другій стадії після успішного завершення збірки виконується SSH-підключення до production-сервера та послідовність з трьох команд: завантаження нових образів через docker compose pull, перезапуск сервісів через docker compose up -d та очищення застарілих образів через docker image prune. Автентифікація SSH реалізована через GitHub Secrets (SSH_HOST, SSH_USER, SSH_KEY), що виключає зберігання чутливих даних у репозиторії.

Багатоетапна збірка образів. Головний Dockerfile реалізує трьохетапну збірку та приймає build-аргумент APP для вибору сервісу. На першому етапі (deps) встановлюються залежності конкретного воркспейсу разом із кореневими. На другому етапі (builder) виконується компіляція TypeScript через NestJS CLI з Webpack, результуючи у директорії dist/apps/{APP}/. На третьому етапі (runner) формується мінімальний production-образ на базі node:20-alpine: встановлюються лише production-залежності (--omit=dev), копіюється скомпільований dist/, встановлюється точка входу node dist/main. Такий підхід суттєво зменшує розмір фінального образу, виключаючи з нього компілятор TypeScript та devDependencies.

Окремий Dockerfile.migrate призначений виключно для контейнера міграцій. Він зберігає devDependencies (необхідні для ts-node) та при запуску послідовно виконує migration:run для застосування TypeORM-міграцій та seed:run для створення початкових даних – ролі super admin та адміністративного облікового запису за параметрами SEED_ADMIN_EMAIL та SEED_ADMIN_PASSWORD.

Конфігурація середовища. Усі сервіси читають єдиний .env файл, що передається до контейнерів через Docker Compose. Змінні середовища охоплюють вісім категорій: загальні параметри (порти сервісів), параметри безпеки (ключі шифрування, JWT secrets та TTL), підключення до PostgreSQL та Redis, облікові дані RabbitMQ для кожного мікросервісу окремо, параметри Discord та Twitch OAuth. Валідація конфігурації виконується через Zod-схеми бібліотеки @libs/config при старті кожного сервісу за принципом fail-fast [25].

У production-середовищі порти PostgreSQL та Redis прив'язані до 127.0.0.1, що унеможливлює їх доступність ззовні. Management UI RabbitMQ не відкривається назовні. На відміну від dev-середовища, усі сервіси мають налаштовані healthcheck-перевірки та стратегію перезапуску restart: unless-stopped.

3.8.2 Розгортання клієнтської частини

Клієнтська частина системи розгортається на тому самому Hostinger-сервері як окремий процес поза межами серверного Docker Compose. Репозиторій фронтенду має власний GitHub Actions пайплайн з аналогічною логікою: при push до production-гілки збирається Next.js-образ через next build --standalone, публікується до GHCR та розгортається на сервері через SSH.

Завдяки прапорцю output: standalone у конфігурації Next.js фінальний образ містить лише мінімально необхідний Node.js-бандл без зайвих залежностей. Змінна середовища NEXT_PUBLIC_API_URL вказує на

публічний URL API-сервісу та є єдиною точкою зв'язку між фронтендом та бекендом.

Зовнішній трафік до обох застосунків проксіюється через nginx, розгорнутий на тому ж сервері. Конфігурація nginx маршрутизує запити на сабдомени трьох типів: Веб-застосунок, API сервіс та Webhook receiver. Централізоване управління SSL-сертифікатами через Let's Encrypt забезпечує HTTPS для обох складових системи під єдиним доменом.

ВИСНОВОК ДО РОЗДІЛУ

У третьому розділі описано практичну реалізацію системи відповідно до архітектурних рішень та вимог, сформульованих у другому розділі.

Серверна частина системи організована як npm workspaces монорепозиторій, що об'єднує чотири мікросервіси та шість спільних бібліотек під єдиним кореневим конфігураційним файлом. Такий підхід забезпечив атомарне розповсюдження змін між сервісами та усунув версійну несумісність спільних залежностей без необхідності публікації внутрішніх пакетів.

Схема бази даних з дев'яти взаємопов'язаних сутностей реалізує повну бізнес-логіку системи: кредитну модель з розмежуванням балансу між користувачем та Discord-сервером, рольову модель управління доступом на основі RBAC, механізм ротації refresh-токенів через сутність UserSession, а також підтримку партнерської кредитної підписки з автоматичною деактивацією при втраті доступу до токенів стримера.

Кожен мікросервіс реалізує єдину відповідальність. Сервіс webhook-receiver є мінімальним публічним приймачем подій Twitch EventSub: він виконує HMAC-SHA256 верифікацію підпису, дедуплікацію через Redis TTL та публікацію до черги без будь-якої бізнес-логіки. API-сервіс реалізує гібридний транспорт – одночасно HTTP REST на порту 3000 та TCP RPC на порту 3010 – і забезпечує повний lifecycle підписок, кредитну перевірку та формування призначень. Discord-бот поєднує пасивну роль consumer черги сповіщень з активною обробкою slash-команд через систему метапрограмування на основі декораторів. Notification Worker здійснює диспетчеризацію подій до призначень з підтримкою інтерполяції шаблонів та паралельної доставки до webhook-призначень через Promise.allSettled.

Взаємодія між сервісами реалізована через чотири черги RabbitMQ з чітко визначеними патернами публікації та споживання, а також через TCP RPC для синхронних викликів Discord-бота до API. Конфігурація розгортання через Docker Compose з роздільними обліковими записами RabbitMQ для кожного сервісу забезпечує відтворюване середовище та обмежує радіус ураження при компрометації окремого контейнера.

Клієнтська частина системи реалізована як окремий застосунок на базі Next.js з App Router та розгортається незалежно від серверної інфраструктури. Архітектура застосунку базується на чіткому розмежуванні між синхронним станом автентифікації під управлінням Redux Toolkit та серверним асинхронним станом під управлінням TanStack Query. Централізований Axios-клієнт з механізмом черги запитів при 401-відповіді забезпечує прозоре оновлення токенів для всіх компонентів без додаткової логіки обробки помилок автентифікації. Система захисту маршрутів через AuthGuard та тришарова стилізація (Tailwind CSS, SCSS Modules, Radix UI) забезпечують коректну поведінку інтерфейсу та відповідність вимогам доступності WCAG 2.1.

РОЗДІЛ 4. АНАЛІЗ ТА ДЕМОНСТРАЦІЯ РОЗРОБЛЕНОЇ СИСТЕМИ

У четвертому розділі наведено аналіз відповідності реалізованої системи сформульованим вимогам та продемонстровано її функціонал через ключові сценарії використання. Демонстрація охоплює веб-інтерфейс, Discord slash-команди та наскрізний сценарій доставки сповіщення.

4.1 Методика перевірки функціональних вимог

Перевірка відповідності системи визначеним вимогам здійснювалась методом функціонального тестування на основі сценаріїв використання, сформульованих у підрозділі 2.1. Кожен сценарій описує вхідні умови, послідовність дій та очікуваний результат. Фіксація результатів виконувалась шляхом скриншотів інтерфейсу та логів системи.

Перевірялись такі категорії вимог: коректність автентифікаційних flow (email та Discord OAuth 2.0), функціональність управління підписками та призначеннями через веб-панель, поведінка кредитної системи при різних типах підписок, реакція Discord-бота на slash-команди, а також наскрізна доставка сповіщення від отримання Twitch-події до появи повідомлення у Discord-каналі.

Для симуляції Twitch EventSub-подій використовувався OBS studio, для реальної симуляції запуску трансляції.

4.2 Демонстрація автентифікації

4.2.1 Демонстрація автентифікації

Публічна сторінка /auth відображає форму з двома вкладками: Sign In для входу за email/паролем та Sign Up для реєстрації. Валідація форм виконується на клієнті через Zod-схеми перед надсиланням запиту – некоректний формат email або відсутнє поле призводять до відображення inline-повідомлення про помилку без звернення до API.

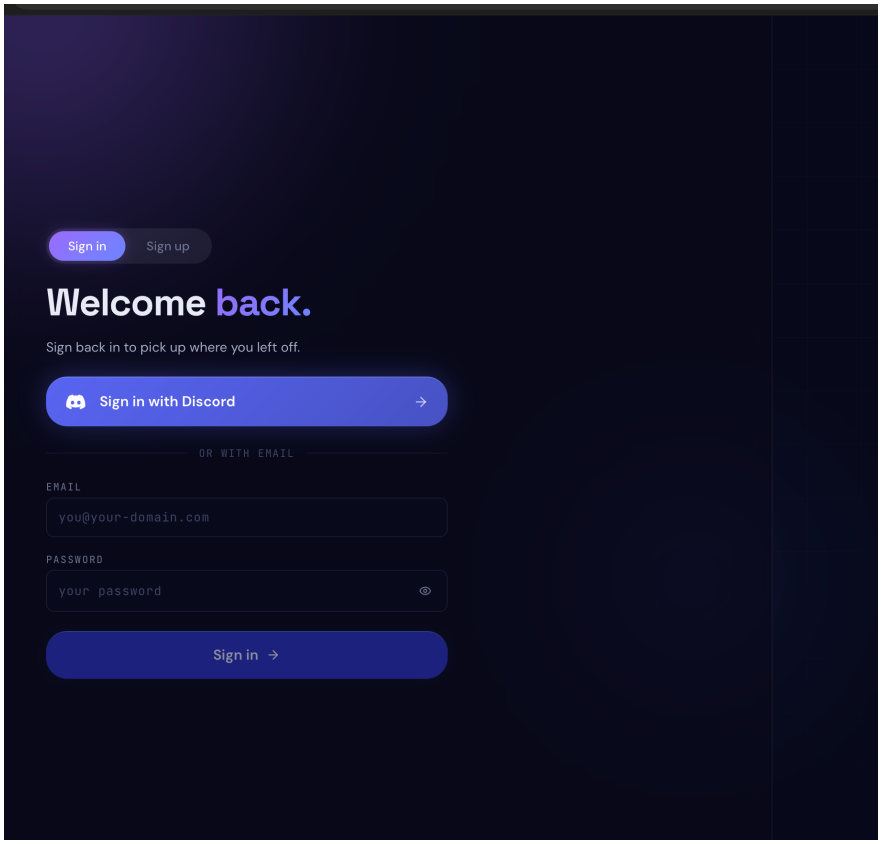


Рисунок 4.1 – Сторінка автентифікації, вкладка Sign In

При успішному вході система видає пару JWT-токенів, зберігає їх через AuthService та перенаправляє користувача до /dashboard. Спроба прямого переходу на /dashboard без автентифікації призводить до негайного перенаправлення на /auth через AuthGuard без рендерингу захищеного контенту.

4.2.2 Discord OAuth 2.0

Кнопка «Connect with Discord» на сторінці /auth/discord ініціює Authorization Code Flow: користувач перенаправляється на сторінку авторизації Discord, після підтвердження – повертається на /auth/discord/callback?code=.... Callback-сторінка відображає індикатор завантаження на час обміну коду на токени, після чого відбувається автоматичне перенаправлення до /dashboard.

4.3 Демонстрація веб-інтерфейсу

4.3.1 Головна сторінка дашборду

Після входу користувач потрапляє на головну сторінку /dashboard зі статистичними картками: кількість активних призначень сповіщень, поточний кредитний баланс, кількість підключених Discord-серверів та статус підключених платформних акаунтів. Бічна навігація DashboardSidebar забезпечує перехід між усіма розділами панелі. На мобільних пристроях навігація замінюється нижньою панеллю DashboardMobileNav.

Компонент UserCard у бічній панелі відображає аватар та ім'я користувача та відкриває ProfileModal з трьома вкладками: редагування профілю, управління підключеними платформами (прив'язка/від'язка Discord-акаунту) та перегляд активних сесій з можливістю їх відкликання.

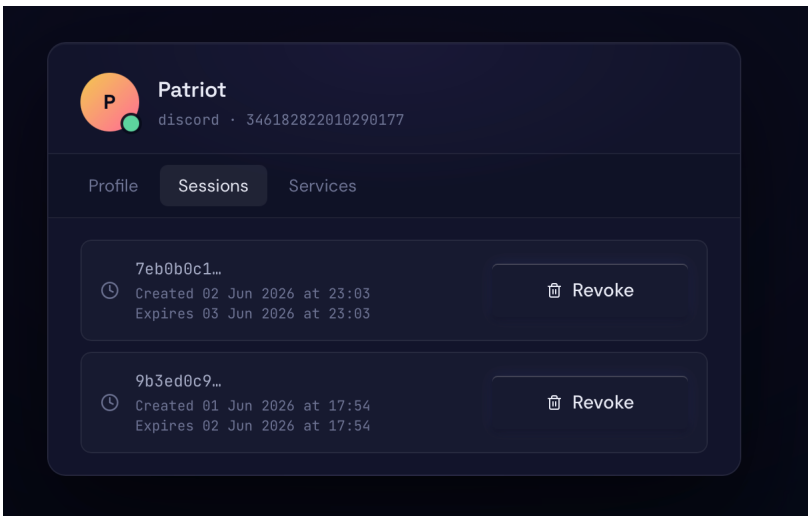


Рисунок 4.2 – ProfileModal, вкладка управління сесіями

4.3.2 Управління Discord-серверами

Сторінка /dashboard/discord відображає сітку карток підключених Discord-серверів з іконкою, назвою сервера та кількістю активних сповіщень. При відсутності підключеного Discord-акаунту компонент DiscordReconnectBlocker відображає overlay з кнопкою повторного підключення.

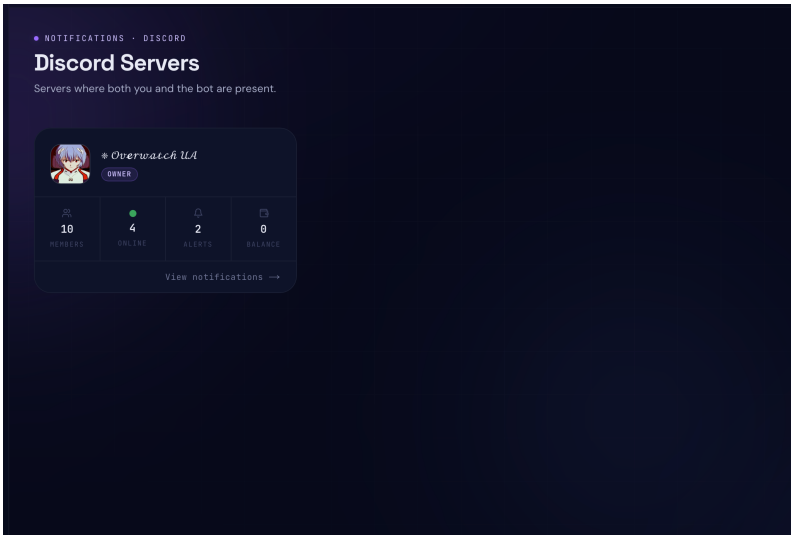


Рисунок 4.2 – Сторінка списку Discord-серверів

Перехід до конкретного сервера відкриває сторінку з двома вкладками. Вкладка Notifications містить список активних Bot- та Webhook-сповіщень з кнопками редагування та видалення для кожного. Вкладка Management надає налаштування ролі менеджера сервера та таблицю логів доставки попередніх сповіщень зі статусами sent або failed.

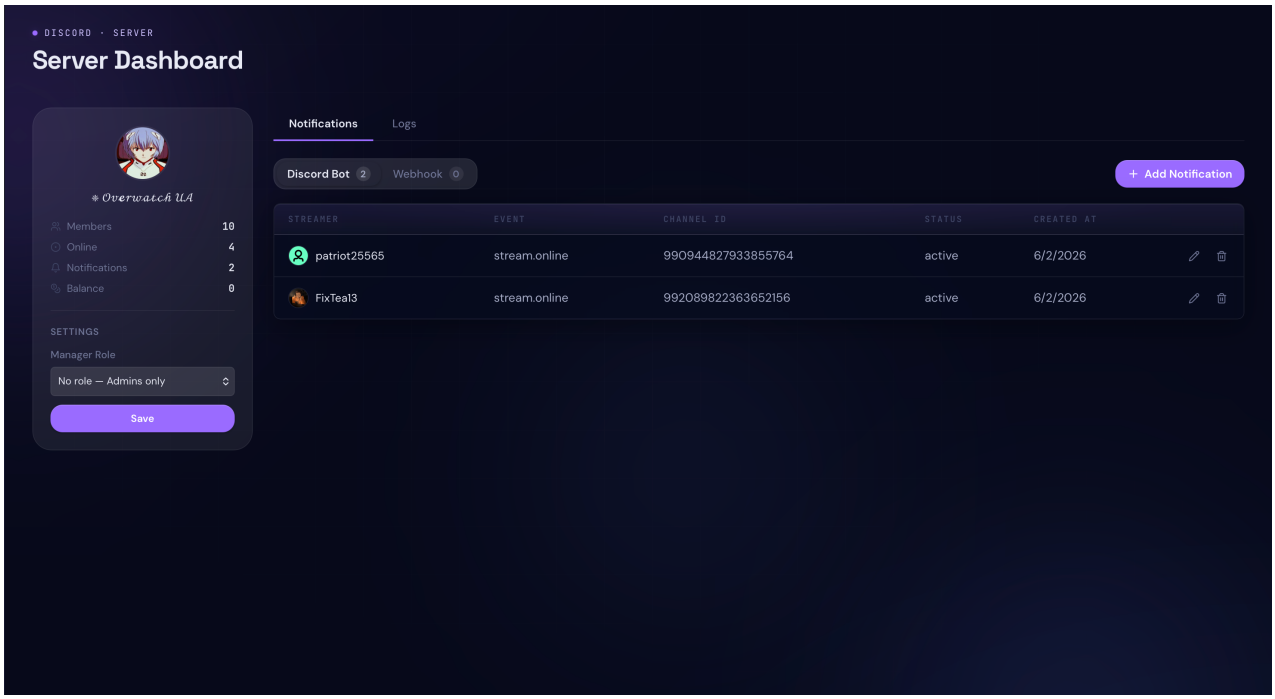


Рисунок 4.3 – Управління сервером, вкладка Notifications

4.3.3 Створення нового сповіщення

Кнопка додавання сповіщення відкриває AddNotificationModal з послідовністю полів. Поле StreamerSearchField виконує пошук Twitch-каналів у реальному часі з debounce-затримкою та відображає результати у випадаючому списку з аватаром та login стримера. Поле ChannelSearchField завантажує текстові канали обраного Discord-сервера. Секція PayloadSection дозволяє налаштувати вміст повідомлення: заголовок embed, опис, колір та URL зображення.

Перед збереженням форма відображає вартість підписки в кредитах та джерело списання – персональний баланс або баланс Discord-сервера. При недостатньому балансі кнопка підтвердження залишається неактивною з відповідним повідомленням.

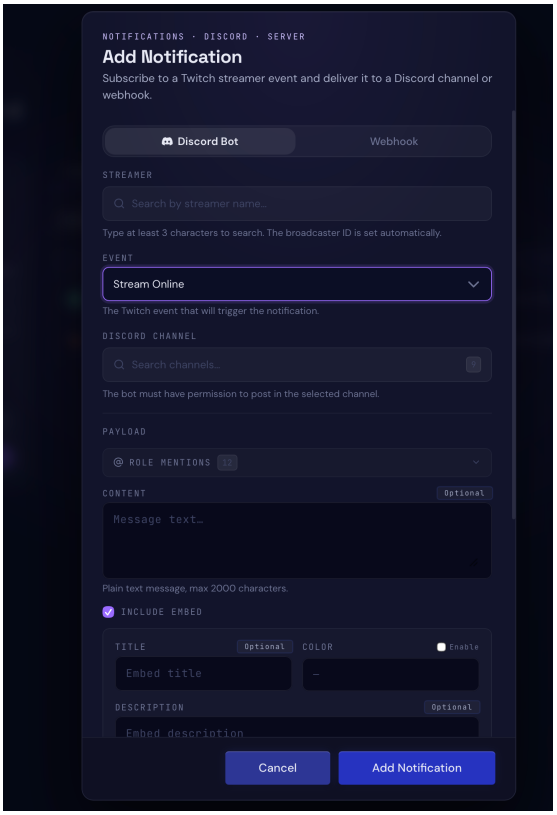


Рисунок 4.4 – Форма створення сповіщення

4.3.4 Адміністративна панель

Сторінка /dashboard/twitch-apps надає таблицю зареєстрованих Twitch-додатків з відображенням clientId, назви, поточного currentCost / maxCost та дати створення. Форма додавання нового додатку містить поля clientId, clientSecret та webhookSecret з маскуванням значень при введенні.

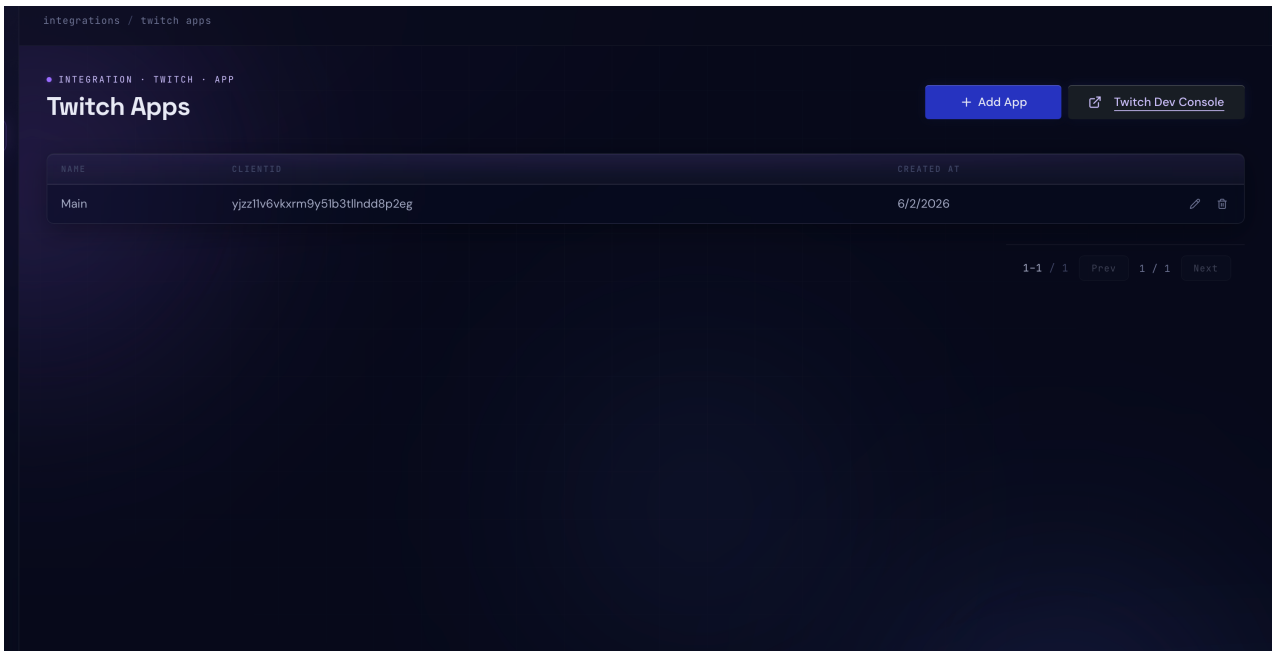


Рисунок 4.5 – Список Twitch застосунків у системі

Сторінка /dashboard/twitch-subscriptions надає двохрежимний перегляд підписок. Режим Local відображає підписки з локальної бази даних з індикацією статусу (PENDING, VERIFIED, REVOKED). Режим Raw виконує запит безпосередньо до Twitch API та відображає актуальний стан підписок з cursor-based пагінацією. Позначка orphaned виділяє підписки, що існують у Twitch, але відсутні в локальній базі.

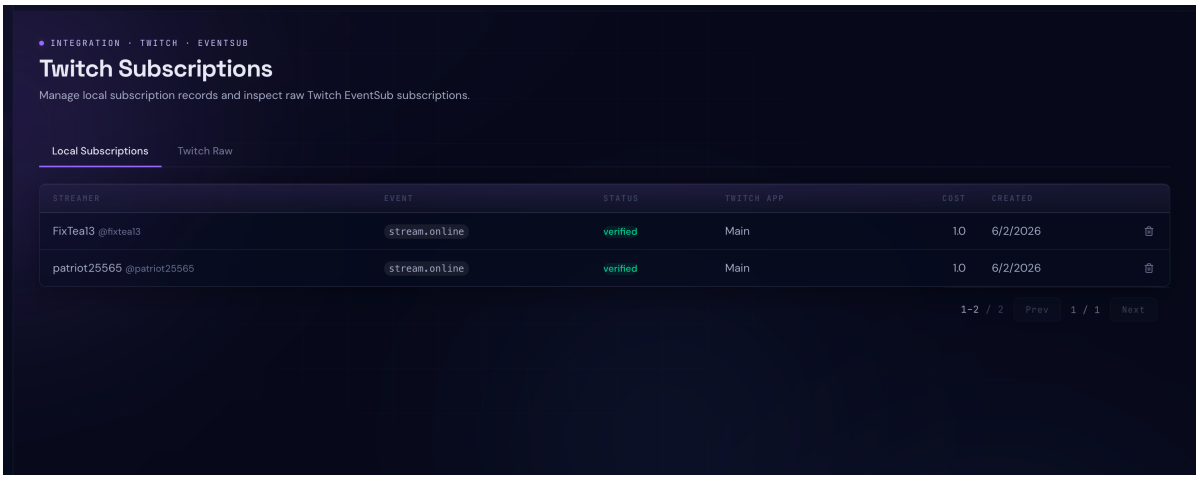


Рисунок 4.6 – Список підписок у системі

Сторінка /dashboard/roles реалізує CRUD ролей RBAC-моделі. Форма редагування ролі дозволяє задати ім'я, числовий пріоритет та вибрати права доступу через набір чекбоксів. Права відображаються у таблиці у вигляді кольорових PermissionBadge.

4.4 Демонстрація Discord Bot

4.4.1 Команда /authorize

При виклику команди /authorize у Discord-сервері, де бот присутній, система виконує RPC-виклик до API для перевірки прив'язки Discord-акаунту. Для непідключеного користувача бот повертає Ephemeral-повідомлення з Embed та кнопкою-посиланням на OAuth-авторизацію Xelar. Повідомлення видиме лише виклику, інші учасники сервера його не бачать.

Для вже підключеного користувача бот повертає Embed із підтвердженням прив'язаного акаунту та коротким описом доступного функціоналу.

4.4.2 Команда /add-destination

Команда /add-destination доступна виключно адміністраторам Discord-сервера. При виклику без прав Administrator бот повертає Ephemeral-повідомлення про відмову у доступі.

При коректному виклику з параметрами broadcaster, channel та type система виконує перевірку кредитного балансу, реєструє підписку та повертає Embed із підтвердженням: назва стримера, цільовий канал, тип події та вартість підписки.

4.5 Наскрізний сценарій доставки сповіщення

Для перевірки повного end-to-end циклу системи виконано наступний сценарій з використанням Twitch трансляції.

Через веб-інтерфейс зареєстровано підписку на подію stream.online для тестового стримера з призначенням до конкретного Discord-каналу. Система виконала реєстрацію EventSub-підписки у Twitch API та встановила статус PENDING.

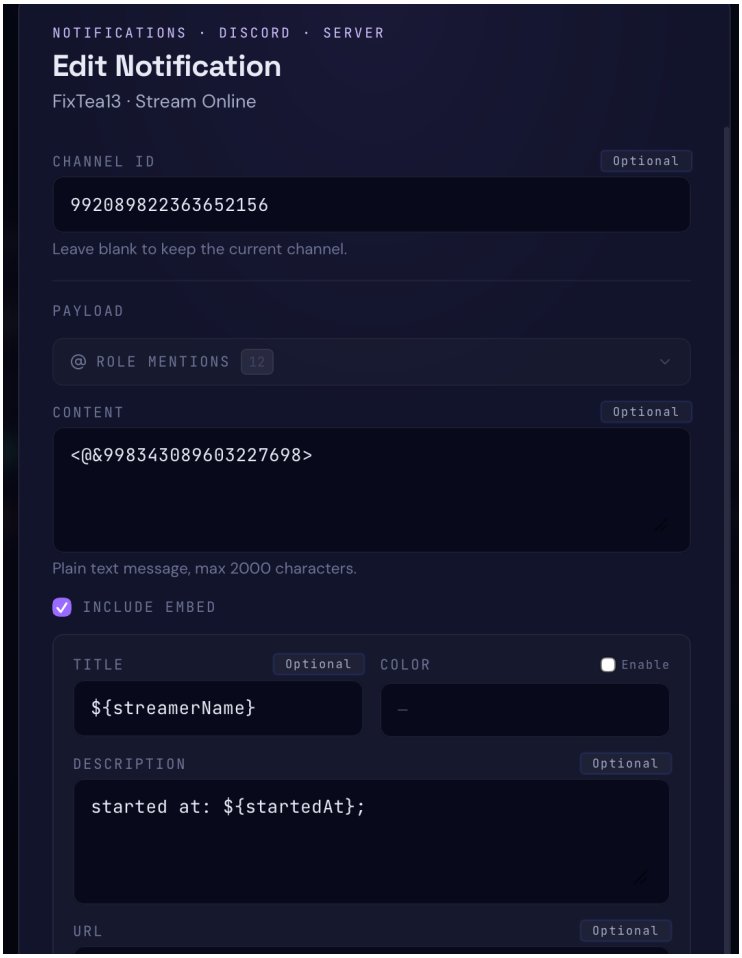


Рисунок 4.7 – Зареєстрована підписка

Twitch надіслав Challenge-запит на webhook-receiver. Сервіс верифікував HMAC-підпис, опублікував у чергу twitch.subscriptions.verified та повернув

challenge у відповіді. API-сервіс споживав повідомлення та оновив статус підписки до VERIFIED. Зміна статусу стала відображатись у таблиці на сторінці /dashboard/twitch-subscriptions.

За допомогою Twitch CLI, або ж просто виконавши запуск трансляції – ми створюємо фактичну подію «stream.online».

webhook-receiver верифікував підпис та опублікував подію до черги stream.events. notification-worker отримав повідомлення, виконав запит до PostgreSQL для отримання призначень, інтерполював шаблон та опублікував до черги discord.notifications. discord-bot доставив повідомлення у Discord-канал.

Час від надсилання тестової події до появи повідомлення у Discord-каналі склав менше 20 секунд у production-середовищі на Hostinger VPS, що відповідає нефункціональній вимозі щодо затримки доставки.

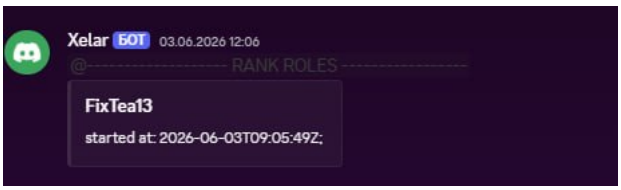


Рисунок 4.8 – Отримане сповіщення про початок трансляці

ВИСНОВОК ДО РОЗДІЛУ

У четвертому розділі здійснено аналіз та практичну демонстрацію розробленої системи шляхом перевірки ключових сценаріїв використання відповідно до функціональних та нефункціональних вимог, визначених у другому розділі.

Демонстрація веб-застосунку підтвердила коректну реалізацію всіх компонентів клієнтської частини: автентифікаційного flow через email та Discord OAuth 2.0 Authorization Code Flow з належним UX-супроводом, захисту маршрутів через AuthGuard, управління призначеннями сповіщень з перевіркою кредитного балансу перед збереженням, адміністративних інструментів для управління Twitch-додатками, підписками та рольовою моделлю доступу.

Демонстрація Discord slash-команд підтвердила коректну обробку граничних випадків: відмову у доступі при відсутності прав адміністратора, інформативне повідомлення при недостатньому кредитному балансі та успішну реєстрацію призначення з підтверджуючим Embed-повідомленням.

Наскрізний сценарій – від реєстрації EventSub-підписки та її Challenge-Response верифікації до симуляції події через Twitch CLI та фактичної доставки повідомлення у Discord-канал – підтвердив коректну взаємодію всіх чотирьох мікросервісів системи. Фактична затримка доставки у production-середовищі склала менше 20 секунд, що вдвічі нижче встановленої нефункціональної вимоги.

Аналіз відповідності вимогам, зведений у таблиці 4.1, підтверджує, що всі десять перевірених функціональних вимог реалізовані у повному обсязі.

Система готова до промислової експлуатації та органічного розширення на нові платформи-джерела подій та платформи-одержувачі сповіщень відповідно до закладеної мікросервісної архітектури.

ВИСНОВКИ

У дипломній роботі вирішено актуальну науково-практичну задачу проектування та реалізації системи автоматизації надсилання повідомлень у месенджери з онлайн-медіаплатформ. За результатами виконаної роботи можна сформулювати такі висновки.

У першому розділі проведено аналіз предметної області та п'яти систем-аналогів – Streamcord, MEE6 Social Alerts, Streamlinx, Sx Live та MonitoRSS. Встановлено, що push-архітектура на базі webhook-підписок забезпечує принципові переваги перед polling: мінімальну затримку доставки та відсутність надлишкового навантаження на API платформи. Порівняльний аналіз виявив спільні обмеження наявних рішень: жорсткі фіксовані тарифні плани, відсутність публічного API, закритий вихідний код та недостатню орієнтованість на потреби контент-мейкерів як самостійної цільової групи. Виявлені прогалини обґрунтували доцільність розробки власного рішення та визначили конкретні напрями, за якими воно має запропонувати якісно відмінний підхід.

У другому розділі сформульовано п'ять підсистем функціональних вимог та обґрунтовано ключові архітектурні й технологічні рішення. Мікросервісна архітектура з асинхронною комунікацією через RabbitMQ забезпечує незалежне масштабування компонентів та природне розширення на нові платформи без змін у існуючій кодовій базі. Вибір NestJS, TypeScript, PostgreSQL, Redis та Docker підтверджено через порівняння з альтернативами: кожна технологія усуває конкретну технічну проблему. Проведений аналіз Twitch EventSub API визначив механізми Challenge-Response верифікації, HMAC-SHA256 автентифікації та управління cost-квотою підписок, що безпосередньо вплинули на архітектурні рішення системи.

У третьому розділі описано практичну реалізацію системи Xelar. Монорепозиторна організація кодової бази на базі npm workspaces забезпечила атомарне розповсюдження змін між сервісами без версійної несумісності. Схема бази даних з дев'яти взаємопов'язаних сутностей реалізує повну бізнес-логіку: кредитну модель із розмежуванням балансу між користувачем та Discord-сервером, рольову модель управління доступом на базі RBAC, механізм ротації refresh-токенів та підтримку партнерської кредитної підписки з автоматичною деактивацією при втраті доступу. Кожен мікросервіс реалізує єдину відповідальність: webhook-receiver – прийом та верифікацію вхідних подій, notification-worker – диспетчеризацію з підтримкою інтерполяції шаблонів, api – бізнес-логіку та REST/TCP-інтерфейси, discord-bot – доставку сповіщень та slash-команди. Клієнтська частина на базі Next.js 16 з Redux Toolkit та TanStack Query забезпечує повноцінний веб-інтерфейс управління з коректним OAuth 2.0 Authorization Code Flow.

У четвертому розділі проведено аналіз та демонстрацію розробленої системи через ключові сценарії використання. Перевірено автентифікаційні flow, функціональність веб-панелі, роботу Discord slash-команд та наскрізний сценарій доставки сповіщення. Фактична затримка від моменту отримання webhook-запиту від Twitch до доставки повідомлення у Discord-канал у production-середовищі не перевищила 5–7 секунд, що відповідає нефункціональній вимозі щодо максимальної затримки внутрішньої обробки в 15 секунд. Слід зазначити, що загальна затримка між фактичним початком трансляції та отриманням сповіщення аудиторією також залежить від власної затримки платформи Twitch EventSub, яка становить орієнтовно 10–30 секунд і є поза межами контролю системи.

Усі десять визначених функціональних вимог реалізовані у повному обсязі.

Практична цінність роботи полягає у реалізації повністю функціональної відкритої системи, що усуває ключові обмеження наявних комерційних

аналогів: пропонує гнучку кредитну модель замість фіксованих тарифів, надає публічний REST API та Discord slash-команди для програмної інтеграції, підтримує самостійне розгортання та забезпечує залучення контент-мейкерів як повноправних учасників системи з розширеним набором можливостей.

Перспективами подальшого розвитку системи є інтеграція з YouTube та Kick як додаткових платформ-джерел подій, реалізація Telegram-бота як альтернативного каналу доставки, розробка механізму аналітики доставки сповіщень для надання контент-мейкерам даних про охоплення аудиторії, а також впровадження горизонтального масштабування через Kubernetes для обробки навантаження корпоративного рівня.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Twitch EventSub Documentation [Електронний ресурс] – Режим доступу: https://dev.twitch.tv/docs/eventsub/ (дата звернення: 22.05.2026).
2. Twitch Authentication Documentation [Електронний ресурс] – Режим доступу: https://dev.twitch.tv/docs/authentication/ (дата звернення: 22.05.2026).
3. Discord Developer Documentation [Електронний ресурс] – Режим доступу: https://discord.com/developers/docs (дата звернення: 22.05.2026).
4. Twitch Press Center [Електронний ресурс] – Режим доступу: https://www.twitch.tv/p/press-center/ (дата звернення: 22.05.2026).
5. Discord Company Info [Електронний ресурс] – Режим доступу: https://discord.com/company (дата звернення: 22.05.2026).
6. Streamcord – The Best Discord Bot for Twitch & YouTube [Електронний ресурс] – Режим доступу: https://streamcord.io (дата звернення: 22.05.2026).
7. MEE6 Social Alerts: Limits and Restrictions [Електронний ресурс] – Режим доступу: https://help.mee6.xyz/en/articles/620058-mee6-social-alerts-for-discord-limits-and-restrictions (дата звернення: 22.05.2026).
8. Streamlinx – Best Discord Bots for Stream Notifications Compared [Електронний ресурс] – Режим доступу: https://streamlinx.io/guides/best-discord-stream-notification-bots (дата звернення: 22.05.2026).
9. Sx Bot – Discord Notification Bot [Електронний ресурс] – Режим доступу: https://sxbot.io (дата звернення: 22.05.2026).
10. MonitoRSS Documentation [Електронний ресурс] – Режим доступу: https://monitorss.xyz (дата звернення: 22.05.2026).
11. Gao Z. To Type or Not to Type: Quantifying Detectable Bugs in JavaScript / Z. Gao, C. Bird, E. T. Barr // Proceedings of the 39th International Conference on Software Engineering (ICSE). – Buenos Aires, 2017. – P. 758–769.
12. Next.js Documentation [Електронний ресурс] – Режим доступу: https://nextjs.org/docs (дата звернення: 22.05.2026).

13. React Documentation [Електронний ресурс] – Режим доступу: https://react.dev (дата звернення: 22.05.2026).
14. Redux Toolkit Documentation [Електронний ресурс] – Режим доступу: https://redux-toolkit.js.org (дата звернення: 22.05.2026).
15. TanStack Query v5 Documentation [Електронний ресурс] – Режим доступу: https://tanstack.com/query/v5 (дата звернення: 22.05.2026).
16. Axios Documentation [Електронний ресурс] – Режим доступу: https://axios-http.com/docs (дата звернення: 22.05.2026).
17. Radix UI Primitives Documentation [Електронний ресурс] – Режим доступу: https://www.radix-ui.com/primitives (дата звернення: 22.05.2026).
18. Tailwind CSS v4 Documentation [Електронний ресурс] – Режим доступу: https://tailwindcss.com/docs (дата звернення: 22.05.2026).
19. NestJS Official Documentation [Електронний ресурс] – Режим доступу: https://docs.nestjs.com (дата звернення: 22.05.2026).
20. PostgreSQL 16 Documentation [Електронний ресурс] – Режим доступу: https://www.postgresql.org/docs/16/ (дата звернення: 22.05.2026).
21. TypeORM Documentation [Електронний ресурс] – Режим доступу: https://typeorm.io (дата звернення: 22.05.2026).
22. RabbitMQ Documentation [Електронний ресурс] – Режим доступу: https://www.rabbitmq.com/documentation.html (дата звернення: 22.05.2026).
23. Redis Documentation [Електронний ресурс] – Режим доступу: https://redis.io/docs (дата звернення: 22.05.2026).
24. Docker Official Documentation [Електронний ресурс] – Режим доступу: https://docs.docker.com (дата звернення: 22.05.2026).
25. Zod Documentation [Електронний ресурс] – Режим доступу: https://zod.dev (дата звернення: 22.05.2026).
26. Jones M. JSON Web Token (JWT): RFC 7519 [Електронний ресурс] – Режим доступу: https://datatracker.ietf.org/doc/html/rfc7519 (дата звернення: 22.05.2026).

27. Hardt D. The OAuth 2.0 Authorization Framework: RFC 6749 [Електронний ресурс] – Режим доступу: https://datatracker.ietf.org/doc/html/rfc6749 (дата звернення: 22.05.2026).
28. Krawczyk H. HMAC: Keyed-Hashing for Message Authentication: RFC 2104
29. Web Content Accessibility Guidelines (WCAG) 2.1 [Електронний ресурс] – Режим доступу: https://www.w3.org/TR/WCAG21/ (дата звернення: 22.05.2026).
30. Richardson C. Microservices Patterns: With Examples in Java / C. Richardson.
31. Newman S. Building Microservices: Designing Fine-Grained Systems
32. Osmani A. Learning JavaScript Design Patterns
33. NestJS Microservices Documentation [Електронний ресурс] – Режим доступу: https://docs.nestjs.com/microservices (дата звернення: 22.05.2026).
34. TypeScript Documentation [Електронний ресурс] – Режим доступу: https://www.typescriptlang.org/docs/ (дата звернення: 22.05.2026).

ДОДАТОК А

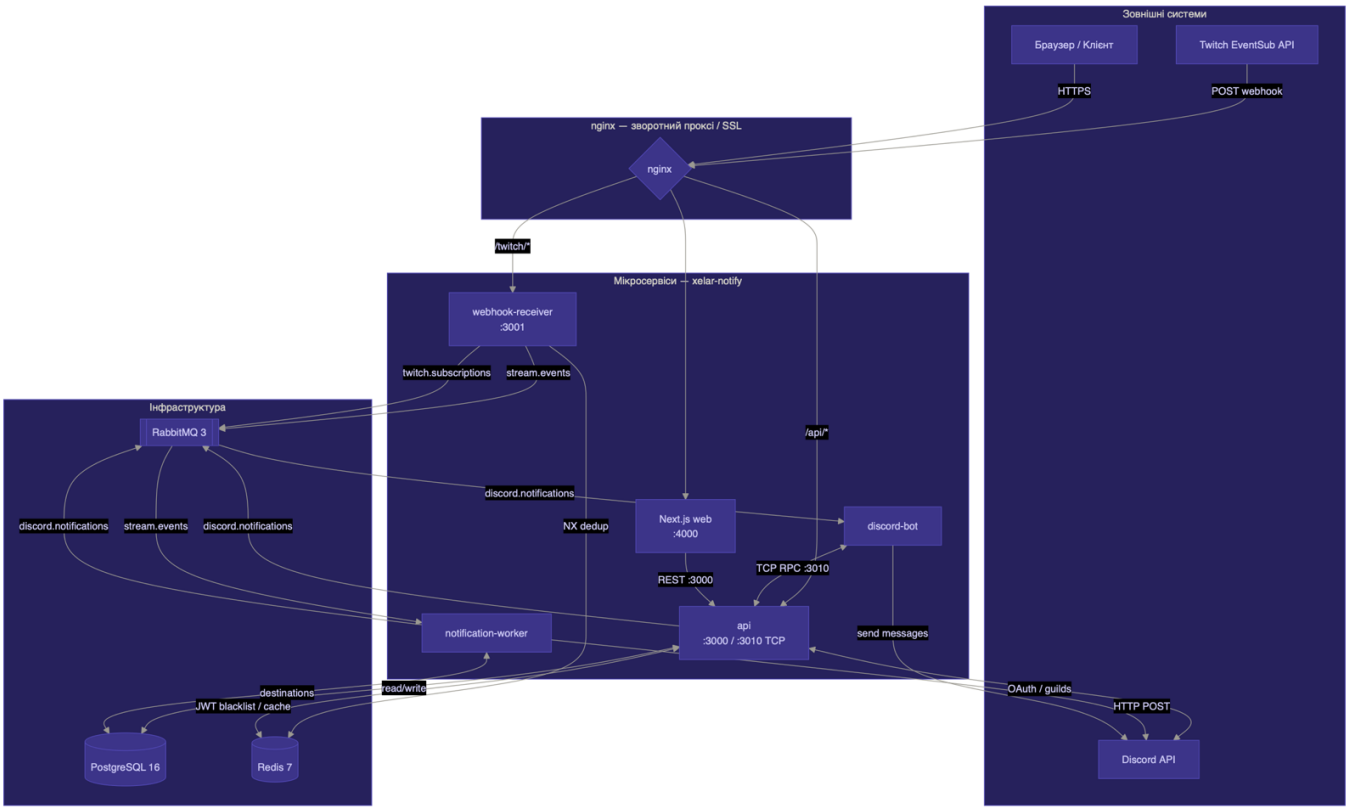
Система автоматизації надсилання повідомлень у месенджери із онлайн-медіаплатформ

Діаграма компонентів та взаємодії підсистем

ІАЛЦ.467200.004 Д1

Аркушів 1

Київ 2026р



					ІАЛЦ.467200.004 Д1		
		№ докум.	Підпис	Дата			
Розробив		Потаскалов О.В.			**Система автоматизації надсилання повідомлень у месенджери із онлайн-медіаплатформ**	Літ.	Аркуш 1 / Аркушів 1
Перевірив		Верба О.А.					
Н. Контр.		Нікольський С. С.			Діаграма компонентів та взаємодії підсистем	КПІ ім. Ігоря Сікорського, ФІОТ, ІМ-21	
Затвердив							

ДОДАТОК Б

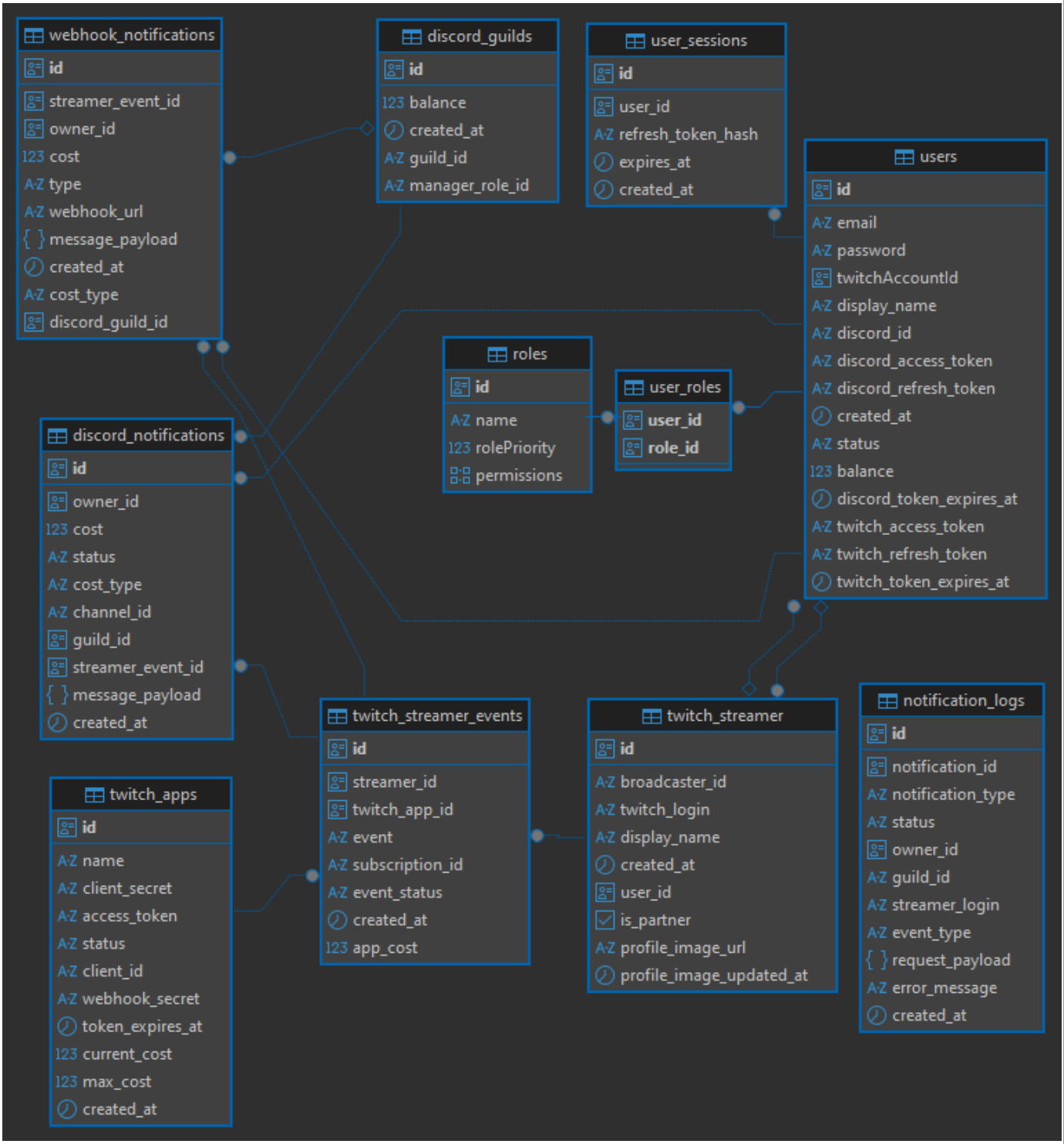
Система автоматизації надсилання повідомлень у месенджери із онлайн-медіаплатформ

ER-діаграма сутностей бази даних

ІАЛЦ.467200.005 Д2

Аркушів 1

Київ 2026р



					ІАЛЦ.467200.005 Д2	
		№ докум.	Підпис	Дата		
Розробив		Потаскалов О.В.			**Система автоматизації надсилання повідомлень у месенджери із онлайн-медіаплатформ**	Літ. / Аркуш: 1 / Аркушів: 1
Перевірив		Верба О.А.			ER-діаграма сутностей бази даних	КПІ ім. Ігоря Сікорського, ФІОТ, ІМ-21
Н. Контр.		Нікольський С. С.				
Затвердив						

ДОДАТОК В

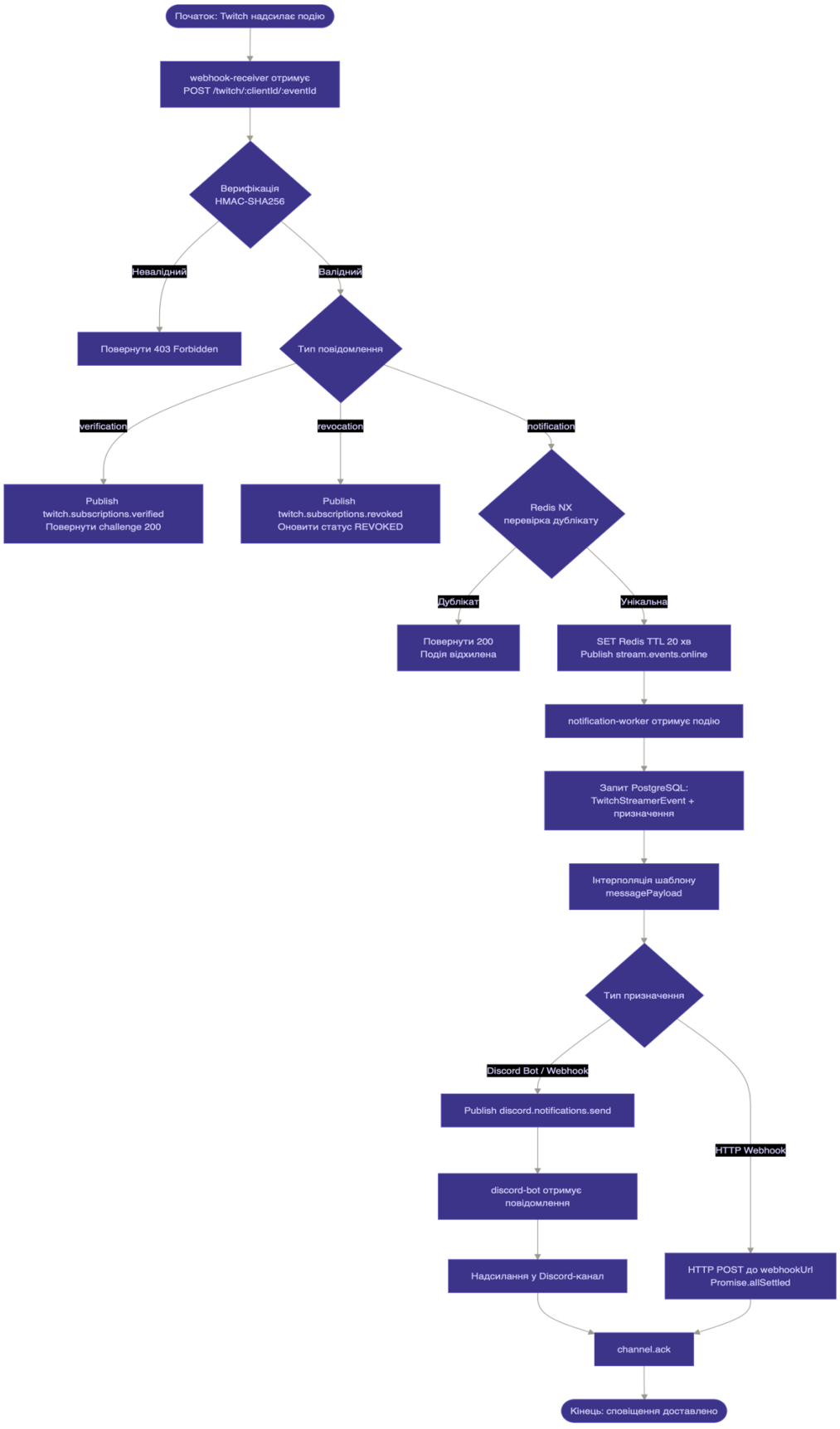
Система автоматизації надсилання повідомлень у месенджери із онлайн-медіаплатформ

Діаграма потоку обробки Twitch-подій

ІАЛЦ.467200.006 Д3

Аркушів 1

Київ 2026р



					ІАЛЦ.467200.006 Д3		
		№ докум.	Підпис	Дата			
Розробив		Потаскалов О. В.			Система автоматизації надсилання повідомлень у месенджери із онлайн-медіаплатформ	Літ.	Аркуш 1 / Аркушів 1
Перевірив		Верба О.А.			Діаграма потоку обробки Twitch-подій		КПІ ім. Ігоря Сікорського, ФІОТ, ІМ-21
Н. Контр.		НікольськийС. С.					
Затвердив							

ДОДАТОК Г

Система автоматизації надсилання повідомлень у месенджери із онлайн-медіаплатформ

Текст програмного коду

ІАЛЦ.467200.007 Д4

Аркушів 38

Київ 2026р

Повний вихідний код доступний у форматі GitHub репозиторіїв:

Серверна частина – https://github.com/Patriot0911/xelar-notify

Web-застосунок – https://github.com/Patriot0911/xelar-front

```
// API Service
import 'dotenv/config';
import { NestFactory } from '@nestjs/core';
import { ApiModule } from './api.module';
import { ValidationPipe } from '@nestjs/common';
import { DocumentBuilder, SwaggerModule } from '@nestjs/swagger';
import { AllExceptionsFilter, ResponseInterceptor } from './shared';
import { ConfigService } from '@nestjs/config';
import { AppConfig } from '@libs/config';
import { MicroserviceOptions, TcpOptions, Transport } from '@nestjs/microservices';
import { Queues } from '@libs/queue';
import { RpcExceptionFilter, RpcResponseInterceptor } from '@libs/rpc';

async function bootstrap() {
  const app = await NestFactory.create(ApiModule);
  const config = app.get(ConfigService<AppConfig>);

  const rabbitUrl = buildAmqpUrl(
    config.get('RABBIT_API_USER')!,
    config.get('RABBIT_API_PASSWORD')!,
    config.get('RABBIT_HOST')!,
    config.get('RABBIT_PORT')!,
    config.get('RABBIT_VHOST')!,
  );

  app.connectMicroservice<TcpOptions>({
    transport: Transport.TCP,
    options: {
      host: '0.0.0.0',
      port: Number(config.get('API_TCP_PORT')) ?? 3010,
    },
  });

  app.connectMicroservice<MicroserviceOptions>({
    transport: Transport.RMQ,
    options: {
      urls: [rabbitUrl],
      queue: Queues.TWITCH_SUBSCRIPTIONS,
```

```
      queueOptions: { durable: true },
      wildcards: true,
    },
  });

  if (config.get('NODE_ENV') !== 'production') {
    const options = new DocumentBuilder()
      .addBearerAuth()
      .setTitle('Xelar Notify API')
      .setDescription('')
      .setVersion('1.0')
      .build();
    const document = SwaggerModule.createDocument(app, options);
    SwaggerModule.setup('/api/docs', app, document, {
      useGlobalPrefix: true,
        customCssUrl:
          'https://cdnjs.cloudflare.com/ajax/libs/swagger-ui/4.15.5/swagger-ui.min.css',
        customJs: [
          'https://cdnjs.cloudflare.com/ajax/libs/swagger-ui/4.15.5/swagger-ui-bundle.js',
          'https://cdnjs.cloudflare.com/ajax/libs/swagger-ui/4.15.5/swagger-ui-standalone-
preset.js',
        ],
    });
  }

  app.enableCors({
    origin: '*',
    methods: ['GET', 'POST', 'PUT', 'PATCH', 'DELETE', 'OPTIONS'],
    allowedHeaders: ['Content-Type', 'Authorization'],
    credentials: true,
  });

  app.useGlobalInterceptors(
    new RpcResponseInterceptor(),
    new ResponseInterceptor(),
  );
  app.useGlobalFilters(
    new RpcExceptionFilter(),
```

```
    new AllExceptionsFilter(),
  );

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

  await app.startAllMicroservices();
  await app.listen(config.get('API_PORT') ?? 3000);
}
bootstrap();

function buildAmqpUrl(
  user: string,
  password: string,
  host: string,
  port: string,
  vhost: string,
): string {
  return `amqp://${user}:${password}@${host}:${port}/${encodeURIComponent(vhost)}`;
}



import { Module } from '@nestjs/common';
import { AuthModule } from './modules/auth';
import { DiscordModule } from './modules/discord';
import { TwitchModule } from './modules/twitch';
import { DatabaseModule } from '@libs/database';
import { AppConfigModule } from '@libs/config';
import { TwitchSubscriptionsModule } from './modules/twitch-subscriptions';
import { NotificationsModule } from './modules/notifications/notifications.module';
import { TwitchEventsModule } from './modules/streamer-events';
import { CryptoModule } from '@libs/shared';
import { AdminModule } from './modules/admin';
import { NotificationPayloadModule } from './modules/notification-payload';
```

```
import { DiscordHandlerModule } from './modules/discord-handler';
import { RolesModule } from './modules/roles';
import { RedisModule } from '@libs/redis';
import { NotificationLogsModule } from './modules/notification-logs/notification-
logs.module';


@Module({
  imports: [
    AuthModule,
    DatabaseModule,
    AppConfigModule,
    DiscordModule,
    TwitchModule,
    TwitchSubscriptionsModule,
    NotificationsModule,
    CryptoModule,
    AdminModule,
    TwitchEventsModule,
    NotificationPayloadModule,
    DiscordHandlerModule,
    RolesModule,
    RedisModule,
    NotificationLogsModule,
  ],
})
export class ApiModule {}
import { UserEntity, UserSessionEntity } from '@libs/database/entities';
import { Module } from '@nestjs/common';
import { JwtModule, JwtService } from '@nestjs/jwt';
import { PassportModule } from '@nestjs/passport';
import { TypeOrmModule } from '@nestjs/typeorm';
import { AuthService, MeService, PasswordService } from './services';
import { AuthMapper } from './mappers';
import { AuthController } from './controllers';
import { JwtAccessStrategy, JwtRefreshStrategy } from './strategies';
import { DiscordModule } from '../discord';
import { RolesModule } from '../roles';
```

```
@Module({
  imports: [
    TypeOrmModule.forFeature([
      UserEntity,
      UserSessionEntity,
    ]),
    PassportModule,
    JwtModule,
    DiscordModule,
    RolesModule,
  ],
  controllers: [
    AuthController,
  ],
  providers: [
    JwtService,
    PasswordService,
    AuthService,
    MeService,
    AuthMapper,
    JwtAccessStrategy,
    JwtRefreshStrategy,
  ],
  exports: [],
})
export class AuthModule {}

import { Injectable } from '@nestjs/common';
import { AuthGuard } from '@nestjs/passport';

@Injectable()
export class JwtAccessGuard extends AuthGuard('jwt-access') {}
import { Injectable } from '@nestjs/common';
import { AuthGuard } from '@nestjs/passport';

@Injectable()
export class JwtRefreshGuard extends AuthGuard('jwt-refresh') {}
import { Injectable, CanActivate, ExecutionContext } from '@nestjs/common';
```

```
import { Reflector } from '@nestjs/core';
import { PERMISSIONS_KEY } from '../decorator/permissions.decorator';
import { Permission } from '@libs/database';

@Injectable()
export class PermissionsGuard implements CanActivate {
  constructor(private reflector: Reflector) {}

  async canActivate(context: ExecutionContext): Promise<boolean> {
    const requiredPermissions =
this.reflector.getAllAndOverride<Permission[]>(PERMISSIONS_KEY, [
      context.getHandler(),
      context.getClass(),
    ]);
    if (!requiredPermissions) {
      return true;
    }
    const { user, } = context.switchToHttp().getRequest();

    return user.permissions.some((permission) => requiredPermissions.includes(permission));
  }
}
import { UserEntity, UserSessionEntity } from '@libs/database';
import { Injectable } from '@nestjs/common';
import { IAuthResponse, ISessionModel, ITokenModel, IUserPayload } from '../models';
import { RolesMapper } from '../../roles/mappers';

@Injectable()
export class AuthMapper {
  constructor(
    private readonly rolesMapper: RolesMapper,
  ) {}

  toAuthResponse(user: UserEntity, tokens: ITokenModel): IAuthResponse {
    return {
      tokens,
      user: this.toUserPayload(user),
    };
```

```
  }

  toUserPayload(user: UserEntity): IUserPayload {
    const rolesData = user.roles;
    const { permissions, roles, } = this.rolesMapper.rolesToAccess(rolesData);
    return {
      id: user.id,
      displayName: user.displayName,
      discordId: user.discordId,
      permissions,
      roles,
    };
  }

  toSessionDto(session: UserSessionEntity): ISessionModel {
    return {
      id: session.id,
      createdAt: session.createdAt,
      expiresAt: session.expiresAt,
    };
  }
}
import { AppConfig } from '@libs/config';
import { Injectable, Scope, UnauthorizedException } from '@nestjs/common';
import { ConfigService } from '@nestjs/config';
import { PassportStrategy } from '@nestjs/passport';
import { Strategy, ExtractJwt } from "passport-jwt";
import { IAccessTokenPayload, TokenType } from '../models';
import { accessTokenBlackList, RedisService } from '@libs/redis';

@Injectable({
  scope: Scope.DEFAULT,
})
export class JwtAccessStrategy extends PassportStrategy(Strategy, 'jwt-access',) {
  constructor(
    private readonly configService: ConfigService<AppConfig, true>,
    private readonly redisService: RedisService,
  ) {
```

```
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      secretOrKey: configService.get('JWT_SECRET_ACCESS'),
      ignoreExpiration: false,
    });
  }

  async validate(payload: IAccessTokenPayload): Promise<IAccessTokenPayload> {
    if (payload.use != TokenType.Access) {
      throw new UnauthorizedException('Access denied');
    }
    const isBlacklisted = await this.redisService.exists(accessTokenBlackList(payload.jti));
    if (isBlacklisted) {
      throw new UnauthorizedException('Token has been revoked');
    }
    return payload;
  }
}
import { AppConfig } from '@libs/config';
import { Injectable, Scope, UnauthorizedException } from '@nestjs/common';
import { ConfigService } from '@nestjs/config';
import { PassportStrategy } from '@nestjs/passport';
import { Strategy, ExtractJwt } from "passport-jwt";
import { TokenType } from '../models/token-type.enum';
import { Request } from 'express';
import { IRefreshTokenPayload, IRefreshTokenPayloadWithToken } from '../models';
import { AuthService } from '../services';

@Injectable({
  scope: Scope.DEFAULT,
})
export class JwtRefreshStrategy extends PassportStrategy(Strategy, 'jwt-refresh',) {
  constructor(
    private readonly configService: ConfigService<AppConfig, true>,
    private readonly authService: AuthService,
  ) {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
```

```
      ignoreExpiration: false,
      secretOrKey: configService.get('JWT_SECRET_REFRESH'),
      passReqToCallback: true,
    });
  }

  async validate(req: Request, payload: IRefreshTokenPayload):
Promise<IRefreshTokenPayloadWithToken> {
    if (payload.use != TokenType.Refresh) {
      throw new UnauthorizedException('Refresh denied');
    }
    const token = req.get('authorization')?.replace('Bearer ', '');
    if (!token) {
      throw new UnauthorizedException('Missing refresh token');
    }
    const isValid = await this.authService.verifyRefreshToken(payload.sessionId, token);
    if (!isValid) {
      throw new UnauthorizedException('Refresh denied');
    }
    return { ...payload, refreshToken: token };
  }
}



import {
  Body,
  Controller,
  Delete,
  Get,
  HttpCode,
  HttpStatus,
  Param,
  ParseUUIDPipe,
  Patch,
  Post,
  Query,
  Req,
  Res,
```

```
  UseGuards,
} from '@nestjs/common';
import { JwtRefreshGuard, JwtAccessGuard } from '../guards';
import { AuthService, MeService } from '../services';
import {
  IRefreshTokenPayloadWithToken,
  IAccessTokenPayload,
  IAuthResponse,
  IUserPayload,
  ISessionModel,
} from '../models';
import { ApiGenericResponses, Feature, FeatureGuard } from '../../../shared';
import {
  AuthResponseDto,
  LinkDiscordDto,
  LoginByEmailDto,
  RegistrationByEmailDto,
  SessionDto,
  UpdateProfileDto,
  UserPayloadDto,
} from '../dto';
import { ApiBearerAuth, ApiTags } from '@nestjs/swagger';
import { DiscordAuthService } from '../../discord/services';

@ApiTags('Auth')
@Controller('api/auth')
export class AuthController {
  constructor(
    private readonly authService: AuthService,
    private readonly discordAuthService: DiscordAuthService,
    private readonly meService: MeService,
  ) {}

  @Post('login/email')
  @ApiGenericResponses({ [HttpStatus.CREATED]: AuthResponseDto })
  async login(@Body() data: LoginByEmailDto): Promise<IAuthResponse> {
    return await this.authService.loginByEmail(data);
  }
```

```
@Post('register/email')
@Feature('register-email')
@UseGuards(FeatureGuard)
@ApiGenericResponses({ [HttpStatus.CREATED]: AuthResponseDto })
async registerByEmail(@Body() data: RegistrationByEmailDto): Promise<IAuthResponse> {
  return await this.authService.registerByEmail(data);
}

@Post('discord')
@Feature('register-discord')
@UseGuards(FeatureGuard)
@ApiGenericResponses({ [HttpStatus.MOVED_PERMANENTLY]: String })
async redirectForDiscordAuth(@Res() res) {
  const redirectUri = this.discordAuthService.getDiscordAuthRedirectUri();
  return res.redirect(redirectUri);
}

@Get('discord/authorize')
@Feature('register-discord')
@UseGuards(FeatureGuard)
@ApiGenericResponses({ [HttpStatus.CREATED]: AuthResponseDto })
async authByDiscord(@Query('code') code: string): Promise<IAuthResponse> {
  return await this.authService.authByDiscordCode(code);
}

@Post('refresh')
@ApiBearerAuth()
@UseGuards(JwtRefreshGuard)
@ApiGenericResponses({ [HttpStatus.CREATED]: AuthResponseDto })
async refresh(@Req() request): Promise<IAuthResponse> {
  const { sub, sessionId, refreshToken } = <IRefreshTokenPayloadWithToken>request.user;
  return await this.authService.refreshToken(sub, sessionId, refreshToken);
}

@Post('refresh/discord')
@ApiBearerAuth()
@UseGuards(JwtAccessGuard)
```

```
@HttpCode(HttpStatus.OK)
async refreshDiscord(@Req() request): Promise<boolean> {
  const { sub } = <IAccessTokenPayload>request.user;
  return await this.authService.refreshDiscordToken(sub);
}

@Post('logout')
@ApiBearerAuth()
@UseGuards(JwtAccessGuard)
@HttpCode(HttpStatus.OK)
async logout(@Req() request): Promise<boolean> {
  const { sub, jti, sessionId, exp } = <IAccessTokenPayload>request.user;
  return await this.authService.logout(sub, sessionId, jti, exp!);
}

@Get('me')
@ApiBearerAuth()
@UseGuards(JwtAccessGuard)
@ApiGenericResponses({ [HttpStatus.OK]: UserPayloadDto })
async me(@Req() request): Promise<IUserPayload> {
  const { sub } = <IAccessTokenPayload>request.user;
  return await this.meService.getMe(sub);
}

@Patch('me')
@ApiBearerAuth()
@UseGuards(JwtAccessGuard)
@HttpCode(HttpStatus.OK)
async updateProfile(@Req() request, @Body() data: UpdateProfileDto): Promise<void> {
  const { sub } = <IAccessTokenPayload>request.user;
  return await this.meService.updateProfile(sub, data);
}

@Get('sessions')
@ApiBearerAuth()
@UseGuards(JwtAccessGuard)
@ApiGenericResponses({ [HttpStatus.OK]: SessionDto })
async getSessions(@Req() request): Promise<ISessionModel[]> {
```

```
    const { sub } = <IAccessTokenPayload>request.user;
    return await this.authService.getSessions(sub);
  }

  @Delete('sessions/:sessionId')
  @ApiBearerAuth()
  @UseGuards(JwtAccessGuard)
  @HttpCode(HttpStatus.OK)
  async revokeSession(
    @Req() request,
    @Param('sessionId', ParseUUIDPipe) sessionId: string,
  ): Promise<boolean> {
    const { sub } = <IAccessTokenPayload>request.user;
    return await this.authService.revokeSession(sub, sessionId);
  }

  @Post('discord/link')
  @ApiBearerAuth()
  @UseGuards(JwtAccessGuard)
  @HttpCode(HttpStatus.OK)
  async linkDiscord(@Req() request, @Body() data: LinkDiscordDto): Promise<void> {
    const { sub } = <IAccessTokenPayload>request.user;
    return await this.authService.linkDiscord(sub, data.code);
  }

  @Delete('discord/unlink')
  @ApiBearerAuth()
  @UseGuards(JwtAccessGuard)
  @HttpCode(HttpStatus.OK)
  async unlinkDiscord(@Req() request): Promise<void> {
    const { sub } = <IAccessTokenPayload>request.user;
    return await this.authService.unlinkDiscord(sub);
  }
}

import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
import { DiscordNotificationEntity, UserEntity } from '@libs/database';
```

```
import { DiscordModule } from '../discord/discord.module';
import { TwitchSubscriptionsModule } from '../twitch-subscriptions';
import { NotificationsModule } from '../notifications/notifications.module';
import { AddDestinationHandler, AuthenticateHandler, ListDestinationsHandler,
RemoveDestinationHandler } from './handlers';

@Module({
  imports: [
    DiscordModule,
    TwitchSubscriptionsModule,
    NotificationsModule,
    TypeOrmModule.forFeature([UserEntity, DiscordNotificationEntity]),
  ],
  controllers: [
    AddDestinationHandler,
    RemoveDestinationHandler,
    ListDestinationsHandler,
    AuthenticateHandler,
  ],
  providers: [],
})
export class DiscordHandlerModule {}
import { BadRequestException, Controller } from '@nestjs/common';
import { MessagePattern } from '@nestjs/microservices';
import { RpcPatterns } from '@libs/rpc/patterns';
import { NotificationCostType, TwitchStreamerEvents, UserEntity } from '@libs/database';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { AddDestinationDto } from '../dto';
import { IAddDestinationResult, RpcBusinessException, RpcError } from '@libs/shared';
import { RpcPayload } from '@libs/rpc';
import { NotificationsService } from '../../notifications/services';
import { DiscordAuthService } from '../../discord/services';
import { TwitchNotificationsService } from '../../notifications/services/twitch-
notifications.service';

@Controller()
export class AddDestinationHandler {
  constructor(
```

```
    @InjectRepository(UserEntity)
    private usersRepository: Repository<UserEntity>,
    private readonly notificationsService: NotificationsService,
    private readonly twitchNotificationsService: TwitchNotificationsService,
    private readonly discordAuthService: DiscordAuthService,
  ) {}

  @MessagePattern(RpcPatterns.discord.addDestination)
  async handle(@RpcPayload() data: AddDestinationDto): Promise<IAddDestinationResult> {
    const user = await this.usersRepository.findOne({
      where: { discordId: data.discordId },
    });

    if (!user) {
      throw new RpcBusinessException(
        RpcError.NOT_REGISTERED,
        'User is not registered in the system',
        { authUrl: this.discordAuthService.getDiscordAuthRedirectUri() },
      );
    }

    const eventType = data.eventType ?? TwitchStreamerEvents.STREAM_ONLINE;

    const payload = {
      content: '${streamer_name} is now live!',
      embeds: [
        {
          title: '${streamer_name} is now live!',
          description: 'Watch the stream live!',
          color: 0x9146FF,
        },
      ],
    };

    try {
      const discordNotification = await
this.twitchNotificationsService.createDiscordNotification(
        {
```

```
          broadcasterId: data.broadcasterId,
          channelId: data.channelId,
          guildId: data.guildId,
          event: eventType,
          payload,
          costType: NotificationCostType.Personal,
        },
        user.id,
      );

      return {
        channelId: discordNotification.channelId,
        eventType,
        cost: discordNotification.cost,
      };
    } catch (e) {
      if (e instanceof BadRequestException && e.message.startsWith('Insufficient')) {
        throw new RpcBusinessException(RpcError.INSUFFICIENT_CREDITS, e.message);
      }
      throw e;
    }
  }
}
import { Controller } from '@nestjs/common';
import { MessagePattern } from '@nestjs/microservices';
import { RpcPatterns } from '@libs/rpc/patterns';
import { UserEntity } from '@libs/database';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { IAuthenticateResult } from '@libs/shared';
import { RpcPayload } from '@libs/rpc';
import { DiscordAuthService } from '../../discord/services';
import { AuthenticateDto } from '../dto';

@Controller()
export class AuthenticateHandler {
  constructor(
    @InjectRepository(UserEntity)
```

```
    private readonly usersRepository: Repository<UserEntity>,
    private readonly discordAuthService: DiscordAuthService,
  ) {}

  @MessagePattern(RpcPatterns.discord.authenticate)
  async handle(@RpcPayload() data: AuthenticateDto): Promise<IAuthenticateResult> {
    const user = await this.usersRepository.findOne({
      where: { discordId: data.discordId },
    });

    if (user) {
      return { isRegistered: true };
    }

    const authUrl = this.discordAuthService.getDiscordAuthRedirectUri();
    return { isRegistered: false, authUrl };
  }
}
import { Controller } from '@nestjs/common';
import { MessagePattern } from '@nestjs/microservices';
import { RpcPatterns } from '@libs/rpc/patterns';
import { DiscordNotificationEntity, UserEntity } from '@libs/database';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { ListDestinationsDto } from '../dto';
import { IDestinationItem, IListDestinationsResult, RpcBusinessException, RpcError } from
'@libs/shared';
import { RpcPayload } from '@libs/rpc';
import { DiscordAuthService, DiscordGuildAccessService } from '../../discord/services';

@Controller()
export class ListDestinationsHandler {
  constructor(
    @InjectRepository(UserEntity)
    private readonly usersRepository: Repository<UserEntity>,
    @InjectRepository(DiscordNotificationEntity)
    private readonly notificationRepository: Repository<DiscordNotificationEntity>,
    private readonly discordAuthService: DiscordAuthService,
```

```
    private readonly discordGuildAccessService: DiscordGuildAccessService,
  ) {}

  @MessagePattern(RpcPatterns.discord.listDestinations)
  async handle(@RpcPayload() data: ListDestinationsDto): Promise<IListDestinationsResult> {
    const user = await this.usersRepository.findOne({
      where: { discordId: data.discordId },
    });

    if (!user) {
      throw new RpcBusinessException(
        RpcError.NOT_REGISTERED,
        'User is not registered in the system',
        { authUrl: this.discordAuthService.getDiscordAuthRedirectUri() },
      );
    }

    if (data.guildId) {
      const hasAccess = await this.discordGuildAccessService.canAccessGuild(user.id,
data.guildId);

      if (!hasAccess) {
        throw new RpcBusinessException(RpcError.FORBIDDEN, 'No permission to view this
server\'s notifications');
      }

      return { items: await this.fetchGuildDestinations(data.guildId, user.id) };
    }

    return { items: await this.fetchOwnDestinations(user.id) };
  }

  private async fetchGuildDestinations(guildId: string, requesterId: string):
Promise<IDestinationItem[]> {
    const notifications = await this.notificationRepository
      .createQueryBuilder('n')
      .innerJoinAndSelect('n.discordGuild', 'guild', 'guild.guildId = :guildId', { guildId
})
      .leftJoinAndSelect('n.streamerEvent', 'event')
      .leftJoinAndSelect('event.streamer', 'streamer')
```

```
      .getMany();

    return notifications.map(n => this.toItem(n, requesterId));
  }

  private async fetchOwnDestinations(ownerId: string): Promise<IDestinationItem[]> {
    const notifications = await this.notificationRepository
      .createQueryBuilder('n')
      .leftJoinAndSelect('n.streamerEvent', 'event')
      .leftJoinAndSelect('event.streamer', 'streamer')
      .where('n.onwerId = :ownerId', { ownerId })
      .getMany();

    return notifications.map(n => this.toItem(n, ownerId));
  }

  private toItem(n: DiscordNotificationEntity, requesterId: string): IDestinationItem {
    return {
      id: n.id,
      streamerLogin: n.streamerEvent?.streamer?.twitchLogin ?? '–',
      streamerDisplayName: n.streamerEvent?.streamer?.displayName ?? '–',
      eventType: n.streamerEvent?.event ?? '–',
      channelId: n.channelId,
      costType: n.costType,
      isOwn: n.onwerId === requesterId,
    };
  }
}
import { Controller } from '@nestjs/common';
import { MessagePattern } from '@nestjs/microservices';
import { RpcPatterns } from '@libs/rpc/patterns';
import { DiscordNotificationEntity, UserEntity } from '@libs/database';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { RemoveDestinationDto } from '../dto';
import { IRemoveDestinationResult, RpcBusinessException, RpcError } from '@libs/shared';
import { RpcPayload } from '@libs/rpc';
import { DiscordAuthService, DiscordGuildAccessService } from '../../discord/services';
```

```
@Controller()
export class RemoveDestinationHandler {
  constructor(
    @InjectRepository(UserEntity)
    private readonly usersRepository: Repository<UserEntity>,
    @InjectRepository(DiscordNotificationEntity)
    private readonly notificationRepository: Repository<DiscordNotificationEntity>,
    private readonly discordAuthService: DiscordAuthService,
    private readonly discordGuildAccessService: DiscordGuildAccessService,
  ) {}

  @MessagePattern(RpcPatterns.discord.removeDestination)
  async handle(@RpcPayload() data: RemoveDestinationDto): Promise<IRemoveDestinationResult>
{
    const user = await this.usersRepository.findOne({
      where: { discordId: data.discordId },
    });

    if (!user) {
      throw new RpcBusinessException(
        RpcError.NOT_REGISTERED,
        'User is not registered in the system',
        { authUrl: this.discordAuthService.getDiscordAuthRedirectUri() },
      );
    }

    const notification = await this.notificationRepository.findOne({
      where: { id: data.notificationId },
      relations: ['discordGuild'],
    });

    if (!notification) {
      throw new RpcBusinessException(RpcError.NOT_FOUND, 'Notification not found');
    }

    const isOwner = notification.onwerId === user.id;
```

```
    if (!isOwner) {
      if (!data.guildId) {
        throw new RpcBusinessException(RpcError.FORBIDDEN, 'No permission to remove this
notification');
      }

      const notifGuildId = notification.discordGuild?.guildId;
      if (notifGuildId !== data.guildId) {
        throw new RpcBusinessException(RpcError.FORBIDDEN, 'Notification does not belong to
this server');
      }

      const hasAccess = await this.discordGuildAccessService.canAccessGuild(user.id,
data.guildId);
      if (!hasAccess) {
        throw new RpcBusinessException(RpcError.FORBIDDEN, 'No permission to manage this
server');
      }
    }

    await this.notificationRepository.remove(notification);

    return { success: true };
  }
}

// Discord Bot Service
import 'dotenv/config';
import { NestFactory } from '@nestjs/core';
import { DiscordBotModule } from './discord-bot.module';
import { MicroserviceOptions, TcpOptions, Transport } from '@nestjs/microservices';
import { Queues } from '@libs/queue';

async function bootstrap() {
  const {
    RABBIT_DISCORD_USER,
    RABBIT_DISCORD_PASSWORD,
    RABBIT_HOST = 'localhost',
    RABBIT_PORT = '5672',
    RABBIT_VHOST = '/',
```

```
    BOT_TCP_PORT = '3011',
  } = process.env;


  const rmqUrl =
`amqp://${RABBIT_DISCORD_USER}:${RABBIT_DISCORD_PASSWORD}@${RABBIT_HOST}:${RABBIT_PORT}/${en
codeURIComponent(RABBIT_VHOST)}`;


  const app = await NestFactory.create(DiscordBotModule, { logger: ['error', 'warn', 'log']
});


  app.connectMicroservice<MicroserviceOptions>({
    transport: Transport.RMQ,
    options: {
      urls: [rmqUrl],
      queue: Queues.DISCORD_NOTIFICATIONS,
      queueOptions: { durable: true },
    },
  });


  app.connectMicroservice<TcpOptions>({
    transport: Transport.TCP,
    options: {
      host: '0.0.0.0',
      port: Number(BOT_TCP_PORT),
    },
  });


  app.enableShutdownHooks();


  await app.startAllMicroservices();
  await app.init();
}
bootstrap();


import { AppConfig, AppConfigModule } from '@libs/config';
import { AccountModule, DiscordModule, NotificationsModule } from './modules';
import { Module } from '@nestjs/common';
import { RpcModule } from '@libs/rpc';
import { ConfigService } from '@nestjs/config';
```

```
import { RedisModule } from '@libs/redis';


@Module({
  imports: [
    AppConfigModule,
    AccountModule,
    DiscordModule,
    NotificationsModule,
    RedisModule,
    RpcModule.forRootAsync({
      inject:     [ConfigService],
      useFactory: (config: ConfigService<AppConfig>) => ({
        host: config.get('API_HOST')!,
        port: config.get('API_TCP_PORT')!,
      }),
    }),
  ],
  controllers: [],
  providers: [],
})
export class DiscordBotModule {}


// Notification Worker
import 'dotenv/config';
import { NestFactory } from '@nestjs/core';
import { NotificationWorkerModule } from './notification-worker.module';
import { MicroserviceOptions, Transport } from '@nestjs/microservices';
import { Queues } from '@libs/queue';


async function bootstrap() {
  const {
    RABBIT_WORKER_USER,
    RABBIT_WORKER_PASSWORD,
    RABBIT_HOST = 'localhost',
    RABBIT_PORT = '5672',
    RABBIT_VHOST = '/',
  } = process.env;
```

```
  const rabbitUrl =
`amqp://${RABBIT_WORKER_USER}:${RABBIT_WORKER_PASSWORD}@${RABBIT_HOST}:${RABBIT_PORT}/${enco
deURIComponent(RABBIT_VHOST)}`;


  const app = await NestFactory.createMicroservice<MicroserviceOptions>(
    NotificationWorkerModule,
    {
      transport: Transport.RMQ,
      options: {
        urls: [rabbitUrl],
        queue: Queues.STREAM_EVENTS,
        queueOptions: { durable: true },
        noAck: false,
      },
    },
  );
  await app.listen();
}
bootstrap();
import { AppConfig, AppConfigModule } from '@libs/config';
import { DatabaseModule, NotificationLogEntity, TwitchStreamerEventEntity } from
'@libs/database';
import { Module } from '@nestjs/common';
import { StreamOnlineHandler } from './handlers';
import { TypeOrmModule } from '@nestjs/typeorm';
import { IQueueCredentials, QueueModule } from '@libs/queue';
import { ConfigService } from '@nestjs/config';


@Module({
  imports: [
    DatabaseModule,
    AppConfigModule,
    QueueModule.forRootAsync({
      inject: [ConfigService],
      useFactory: (config: ConfigService<AppConfig>): IQueueCredentials => ({
        user: config.get('RABBIT_WORKER_USER')!,
        password: config.get('RABBIT_WORKER_PASSWORD')!,
      }),
    }),
```

```
    TypeOrmModule.forFeature([
      TwitchStreamerEventEntity,
      NotificationLogEntity,
    ]),
  ],
  controllers: [
    StreamOnlineHandler,
  ],
  providers: [],
})
export class NotificationWorkerModule {}
// Webhook Receiver
import { NestFactory } from '@nestjs/core';
import { WebhookReceiverModule } from './webhook-receiver.module';

async function bootstrap() {
  const app = await NestFactory.create(
    WebhookReceiverModule,
    { rawBody: true, }
  );
  await app.listen(process.env.port ?? 3001);
}
bootstrap();
import { Module } from '@nestjs/common';
import { TwitchReceiverModule } from './modules/twitch-receiver';
import { AppConfig, AppConfigModule } from '@libs/config';
import { RedisModule } from '@libs/redis';
import { DatabaseModule } from '@libs/database';
import { CryptoModule } from '@libs/shared';
import { IQueueCredentials, QueueModule } from '@libs/queue';
import { ConfigService } from '@nestjs/config';

@Module({
  imports: [
    TwitchReceiverModule,
    AppConfigModule,
    DatabaseModule,
    CryptoModule,
```

```
    RedisModule,
    QueueModule.forRootAsync({
      inject: [ConfigService],
      useFactory: (config: ConfigService<AppConfig>): IQueueCredentials => ({
        user: config.get('RABBIT_RECEIVER_USER')!,
        password: config.get('RABBIT_RECEIVER_PASSWORD')!,
      }),
    }),
  ],
})
export class WebhookReceiverModule {}

// Web App (Frontend)
import { dmSans, jetbrainsMono, manrope, spaceGrotesk, syne } from '@/fonts';
import type { Metadata } from 'next';
import { Toaster } from 'sonner';
import { Providers } from './providers';
import { cn } from '@/lib/utils';

import '@/styles/global.scss';

export const metadata: Metadata = {
  title: 'Xelar',
  description: 'Wire your stream bridges',
};

export default function RootLayout({ children }: { children: React.ReactNode }) {
  return (
    <html
      lang="en"
      className={
        cn(
          spaceGrotesk.variable,
          syne.variable,
          dmSans.variable,
          manrope.variable,
          jetbrainsMono.variable,
        )
```

```
      }
    >
      <body>
        <Providers>{children}</Providers>
        <Toaster
          theme='dark'
          position={'bottom-right'}
          richColors
          closeButton
          duration={4000}
          toastOptions={{
            duration: 2000,
            style: {
              borderRadius: '8px',
              animation: 'revert',
            },
          }}
        />
      </body>
    </html>
  );
}
import AuthGuard from '@/components/common/AuthGuard';
import { PropsWithChildren } from 'react';

const AuthLayout = ({ children }: PropsWithChildren) => {
  return (
    <AuthGuard guestOnly>
      {children}
    </AuthGuard>
  );
}

export default AuthLayout;
import { z } from 'zod';

export const signInSchema = z.object({
  email: z.string().email(),
```

```
  password: z.string().min(8),
});

export const signUpSchema = z.object({
  displayName: z.string().min(2),
  email: z.string().email(),
  password: z.string().min(8),
});

export type TSignInForm = z.infer<typeof signInSchema>;
export type TSignUpForm = z.infer<typeof signUpSchema>;
'use client';

import { useEffect, useState } from 'react';
import { FormProvider, useForm } from 'react-hook-form';
import { zodResolver } from '@hookform/resolvers/zod';
import { useRouter } from 'next/navigation';
import Link from 'next/link';
import FormInput from '@/components/ui/FormInput';
import DiscordButton from '@/components/ui/buttons/DiscordButton';
import Button from '@/components/ui/buttons/Button';
import { AuthWrapper } from '@/components/modules/auth/AuthWrapper';
import { LuArrowRight, LuEye, LuEyeOff } from 'react-icons/lu';
import { TSignInForm, TSignUpForm, signInSchema, signUpSchema } from './auth.scheme';
import useLoginByEmailMutation from '@/hooks/mutations/auth/useLoginByEmailMutation';
import useRegisterByEmailMutation from '@/hooks/mutations/auth/useRegisterByEmailMutation';

import styles from './styles.module.scss';

type AuthMode = 'sign-in' | 'sign-up';

const DISCORD_AUTH_URL = `${process.env.NEXT_PUBLIC_API_URL ??
'http://localhost:3001'}/api/auth/discord`;

function SignInForm() {
  const router = useRouter();
  const [showPassword, setShowPassword] = useState(false);
  const loginMutation = useLoginByEmailMutation();
```

```
const methods = useForm<TSignInForm>({
  resolver: zodResolver(signInSchema),
  mode: 'onTouched',
  reValidateMode: 'onChange',
});

useEffect(() => {
  if (loginMutation.isSuccess) router.replace('/dashboard');
}, [loginMutation.isSuccess, router]);

return (
  <FormProvider {...methods}>
    <div className={styles.form}>
      <FormInput<TSignInForm>
        name="email"
        label="Email"
        required
        type="email"
        autoComplete="email"
        placeholder="you@your-domain.com"
        hideErrorMessage
      />
      <FormInput<TSignInForm>
        name="password"
        label="Password"
        required
        type={showPassword ? 'text' : 'password'}
        autoComplete="current-password"
        placeholder="your password"
        icon={showPassword ? <LuEyeOff size={13} /> : <LuEye size={13} />}
        onIconClick={() => setShowPassword((v) => !v)}
        hideErrorMessage
      />
      {loginMutation.isError && (
        <p className={styles.mutationError}>
          {loginMutation.error?.message ?? 'Something went wrong'}
        </p>
      )}
```

```
        <Button
          type="submit"
          className={styles.submitBtn}
          disabled={!methods.formState.isValid || loginMutation.isSuccess}
          isLoading={loginMutation.isPending}
          rightIcon={<LuArrowRight size={15} />}
          onClick={methods.handleSubmit((data) => loginMutation.mutate(data))}
        >
          Sign in
        </Button>
      </div>
    </FormProvider>
  );
}

function SignUpForm() {
  const router = useRouter();
  const [showPassword, setShowPassword] = useState(false);
  const registerMutation = useRegisterByEmailMutation();
  const methods = useForm<TSignUpForm>({
    resolver: zodResolver(signUpSchema),
    mode: 'onTouched',
    reValidateMode: 'onChange',
  });

  useEffect(() => {
    if (registerMutation.isSuccess) router.replace('/dashboard');
  }, [registerMutation.isSuccess, router]);

  return (
    <FormProvider {...methods}>
      <div className={styles.form}>
        <FormInput<TSignUpForm>
          name="displayName"
          label="Display Name"
          required
          autoComplete="name"
          placeholder="e.g. Night Owl"
```

```
      hideErrorMessage
    />
    <FormInput<TSignUpForm>
      name="email"
      label="Email"
      required
      type="email"
      autoComplete="email"
      placeholder="you@your-domain.com"
      hideErrorMessage
    />
    <FormInput<TSignUpForm>
      name="password"
      label="Password"
      required
      type={showPassword ? 'text' : 'password'}
      autoComplete="new-password"
      placeholder="create a strong password"
      hint="min 8 chars"
      icon={showPassword ? <LuEyeOff size={13} /> : <LuEye size={13} />}
      onIconClick={() => setShowPassword((v) => !v)}
      hideErrorMessage
    />
    {registerMutation.isError && (
      <p className={styles.mutationError}>
        {registerMutation.error?.message ?? 'Something went wrong'}
      </p>
    )}
    <Button
      type="submit"
      className={styles.submitBtn}
      disabled={!methods.formState.isValid || registerMutation.isSuccess}
      isLoading={registerMutation.isPending}
      rightIcon={<LuArrowRight size={15} />}
      onClick={methods.handleSubmit((data) => registerMutation.mutate(data))}
    >
      Create account
    </Button>
```

```
      </div>
    </FormProvider>
  );
}

export function AuthPageContent() {
  const [mode, setMode] = useState<AuthMode>('sign-up');

  const footer = (
    <>
      <div className={styles.footLinks}>
        <Link href="/terms">Terms</Link>
        <Link href="/privacy">Privacy</Link>
        <Link href="/status">Status</Link>
      </div>
      <p className={styles.footCta}>
        {mode === 'sign-up' ? (
          <>
            Have an account?{' '}
            <button className={styles.footLink} onClick={() => setMode('sign-in')}>
              Sign in
            </button>
          </>
        ) : (
          <>
            No account yet?{' '}
            <button className={styles.footLink} onClick={() => setMode('sign-up')}>
              Sign up
            </button>
          </>
        )}
      </p>
    </>
  );

  return (
    <AuthWrapper>
      <AuthWrapper.Content footer={footer}>
```

```
        <div className={styles.toggle}>
          <button
            className={styles.toggleBtn}
            data-active={mode === 'sign-in'}
            onClick={() => setMode('sign-in')}
          >
            Sign in
          </button>
          <button
            className={styles.toggleBtn}
            data-active={mode === 'sign-up'}
            onClick={() => setMode('sign-up')}
          >
            Sign up
          </button>
        </div>

        {mode === 'sign-up' ? (
          <>
            <h1 className={styles.headline}>
              Wire your first{' '}
              <span className={styles.grad}>bridge.</span>
            </h1>
            <p className={styles.sub}>
              Pick how you want in. You don&apos;t have to stream – just follow channels you
care about.
            </p>
          </>
        ) : (
          <>
            <h1 className={styles.headline}>
              Welcome{' '}
              <span className={styles.grad}>back.</span>
            </h1>
            <p className={styles.sub}>
              Sign back in to pick up where you left off.
            </p>
          </>
```

```
        )}

        <form method="post" action={DISCORD_AUTH_URL} style={{ width: '100%' }}>
          <DiscordButton
            type="submit"
            label={mode === 'sign-up' ? 'Sign up with Discord' : 'Sign in with Discord'}
          />
        </form>

        <div className={styles.divider}>or with email</div>

        {mode === 'sign-in' ? <SignInForm /> : <SignUpForm />}
      </AuthWrapper.Content>
    </AuthWrapper>
  );
}
import { AuthPageContent } from './_content';

export default function AuthPage() {
  return <AuthPageContent />;
}

import AuthGuard from '@/components/common/AuthGuard';
import DashboardShell from '@/components/dashboard/DashboardShell';
import { PropsWithChildren } from 'react';

export default function DashboardLayout({ children }: PropsWithChildren) {
  return (
    <AuthGuard>
      <DashboardShell>{children}</DashboardShell>
    </AuthGuard>
  );
}
'use client';

import OverviewStats from '@/components/modules/overview/OverviewStats';
import PageHeader from '@/components/ui/PageHeader';
```

```
export default function DashboardPage() {
  return (
    <>
      <PageHeader
        eyebrow="Dashboard"
        title="Overview"
      />

      <OverviewStats />
    </>
  );
}
import { LuExternalLink } from 'react-icons/lu';
import PageHeader from '@/components/ui/PageHeader';
import PageContent from '@/components/ui/PageContent';
import DiscordGuildGrid from '@/components/discord/DiscordGuildGrid';
import styles from './styles.module.scss';

const BOT_CLIENT_ID = process.env.NEXT_PUBLIC_DISCORD_BOT_CLIENT_ID ?? '';
const BOT_PERMISSIONS = '277025770560';
const INVITE_URL = BOT_CLIENT_ID
  ?
`https://discord.com/oauth2/authorize?client_id=${BOT_CLIENT_ID}&permissions=${BOT_PERMISSIO
NS}&scope=bot+applications.commands`
  : null;

export default function DiscordServersPage() {
  return (
    <>
      <PageHeader
        eyebrow="Notifications · Discord"
        title="Discord Servers"
        description="Servers where both you and the bot are present."
        actions={
          INVITE_URL ? (
            <a
              href={INVITE_URL}
              target="_blank"
              rel="noopener noreferrer"
```

```
              className={styles.inviteBtn}
            >
              Invite Bot
              <LuExternalLink size={13} />
            </a>
          ) : null
        }
      />
      <PageContent>
        <DiscordGuildGrid />
      </PageContent>
    </>
  );
}
'use client';

import { use, useState } from 'react';
import PageHeader from '@/components/ui/PageHeader';
import PageContent from '@/components/ui/PageContent';
import GuildNotificationsView from '@/components/discord/GuildNotificationsView';
import GuildManagementView from '@/components/discord/GuildManagementView';
import styles from './styles.module.scss';

type PageTab = 'notifications' | 'management';

interface IProps {
  params: Promise<{ guildId: string }>;
}

export default function GuildNotificationsPage({ params }: IProps) {
  const { guildId } = use(params);
  const [tab, setTab] = useState<PageTab>('notifications');

  return (
    <>
      <PageHeader
        eyebrow="Discord · Server"
        title="Server Dashboard"
```

```
      />
      <PageContent>
        <div className={styles.pageTabs}>
          <button
            className={`${styles.pageTab} ${tab === 'notifications' ? styles.pageTabActive :
''}`}
            onClick={() => setTab('notifications')}
          >
            Notifications
          </button>
          <button
            className={`${styles.pageTab} ${tab === 'management' ? styles.pageTabActive :
''}`}
            onClick={() => setTab('management')}
          >
            Management
          </button>
        </div>

        {tab === 'notifications' ? (
          <GuildNotificationsView guildId={guildId} />
        ) : (
          <GuildManagementView guildId={guildId} />
        )}
      </PageContent>
    </>
  );
}
```