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

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

**До захисту допущено:**
**Завідувач кафедри**
_____ Сергій СТІРЕНКО
(підпис)
“__” _____2024 р.

# Дипломний проєкт

**на здобуття ступеня бакалавра**

**освітньо-професійною програмою “Комп’ютерні системи та мережі”**

**спеціальності 123 “Комп’ютерна інженерія”**

на тему: Телеграм-бот для пошуку роботи

Виконав : студент __4__ курсу, групи _ІО-06_
(шифр групи)

_____Гриciняк Григорій Степанович_____
(прізвище, ім’я, по батькові) (підпис)

Керівник _____асистент Нечай Д. О._____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант (нормоконтроль) асистент Ковальчук О. М._
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент __доц., к.т.н., доцент кафедри АУТС_____
(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 – 2024 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

“Комп’ютерні системи та мережі”

спеціальність 123 “Комп’ютерна інженерія”

**ЗАТВЕРДЖУЮ**
**Завідувач кафедри**
_____ Сергій СТІРЕНКО
(підпис)
“__” _____2024 р.

**ЗАВДАННЯ**
на бакалаврський дипломний проєкт студента

Гриciняка Григорія Степановича

1. Тема проєкту: Телеграм-бот для пошуку роботи
**керівник проєкту:** Нечай Дмитро Олександрович, асистент
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від «27» травня 2024 року № 2112-с
2. Строк подання студентом проєкту «22» *червня 2024 р.*
3. Вихідні дані до проєкту: *технічна документація. теоретичні дані.*
4. Зміст пояснювальної записки (перелік завдань, які потрібно розробити) *Опис предметної області, налаштування інфрастуктури проєкту, розробка додатку, тестування.*
5. Перелік графічного матеріалу (з точним позначенням обов’язкових креслень) структурна схема системи, функціональна схема(діаграма класів), алгоритм дій програмного забезпечення
6. Консультанти розділів проєкту:

| Розділ | Прізвище, ініціали та посада консультанта | Підпис, дата | |
|---|---|---|---|
| | | Завдання видав | Завдання прийняв |
| Нормконтроль | Ковальчук О. М., асистент | | |

7. Дата видачі завдання: 01 вересня 2023 р.

Календарний план

| № п/п | Назва етапу дипломного проєкту | Строк виконання етапу проєкту | Примітка |
|---|---|---|---|
| 1. | *Затвердження теми проєкту* | *01.04.2023-15.04.2024* | |
| *2.* | *Вивчення літератури та предметної області.* | *15.04.2024-20.04.2024* | |
| *3.* | *Аналіз існуючих рішень.* | *20.04.2024-22.04.2024* | |
| *4.* | *Налаштування інфрастуктури проекту(Google Kubernetes Engine).* | *22.04.2024-29.04.2024* | |
| *5.* | *Налаштування Redis та дизайн бази даних PostgresSQL.* | *29.04.2024-03.05.2024* | |
| 6. | *Розробка архітектури Go застосунку.* | *03.05.2024-05.05.2024* | |
| *7.* | *Реалізація коду.* | *05.05.2024-15.05.2024* | |
| *8.* | *Мануальне тестування та розробка автоматичних тестів.* | *15.05.2024-19.05.2024* | |
| *9.* | *Оформлення документації дипломного проєкту* | *19.05.2024* | |
| *10.* | *Передзахист* | *15.06.2024* | |
| *11.* | *Захист* | *22.06.2024* | |

Студент Гриciняк Г.С.
(підпис)

Керівник проєкту Нечай Д.О.
(підпис)

## Анотація

У бакалаврському дипломному проєкті було розроблено комерційно-конкурентоспроможний Telegram-бот, призначений для допомоги у пошуку роботи.

Продукт дає можливість користувачу отримувати нотифікації про появу вакансій швидше, ніж його конкуренти, що підвищує шанси на працевлаштування.

Для реалізації цього завдання було впроваджено ефективні алгоритми для аналізу ринку, а також використано такі передові технології, як система оркестрації Kubernetes, in-memory база даних Redis та мова програмування Go.



## Annotation

In this bachelor's degree project, a commercially competitive Telegram bot was developed to assist in job searching.

The product allows users to receive notifications about new job openings faster than their competitors, increasing their chances of employment.

To achieve this, efficient market analysis algorithms were implemented, and advanced technologies such as the Kubernetes orchestration system, in-memory database Redis, and the Go programming language were utilized.

| справки | Формат | *Значення* | *Найменування* | Кіл. листів | № екземпля | *Додаток* |
|---|---|---|---|---|---|---|
| | | | Документація загальна | | | |
| | | | Знову розроблена | | | |
| | | | | | | |
| | ***А4*** | ***ІАЛЦ.467200.002 ТЗ*** | Telegram-бот для | 3 | | |
| | | | пошуку роботи | | | |
| | | | Технічне завдання | | | |
| | | | | | | |
| | | | | | | |
| | ***А4*** | ***ІАЛЦ.467200.003 ПЗ*** | Еволюційні алгоритми | 64 | | |
| | | | глобальної пошукової | | | |
| | | | оптимізації | | | |
| | | | Пояснювальна записка | | | |
| | | | | | | |
| | ***А4*** | ***ІАЛЦ.467200.004 Д1*** | Telegram-бот для | 1 | | |
| | | | пошуку роботи | | | |
| | | | Структурна схема | | | |
| | | | системи | | | |
| | | | | | | |
| | | | | | | |
| | ***А4*** | ***ІАЛЦ.4672008.005 Д2*** | Telegram-бот для | 1 | | |
| | | | пошуку роботи | | | |
| | | | Функціональна схема | | | |
| | | | (діаграма класів) | | | |
| | | | | | | |
| | | | | | | |
| | ***А4*** | ***ІАЛЦ.467200.006 Д3*** | Telegram-бот для | 1 | | |
| | | | пошуку роботи | | | |
| | | | Алгоритм дій програмного | | | |
| | | | забезпечення | | | |
| | | | | | | |
| | | | | | | |
| | ***А4*** | ***ІАЛЦ.467200.007 Д4*** | Telegram-бот для | 39 | | |
| | | | пошуку роботи | | | |
| | | | Текст програмного коду | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |

| | | | | | | |
|---|---|---|---|---|---|---|
| | | | | | ***ІАЛЦ.467200.002 ПЗ*** | |
| *Зм* | ***Лист*** | ***№ докум.*** | ***Підп*** | ***Дата*** | | |
| ***Розроб*** | | Гриciняк Г.С. | | | *Телеграм-бот для пошуку роботи* Опис альбому | Літ. / Аркуш 1 / Аркушів 1 |
| ***Перев*** | | Нечай Д.О | | | | КПІ ім. Ігоря Сікорського, ФІОТ, ІО-06 |

# ТЕХНІЧНЕ ЗАВДАННЯ
# ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: *«Telegram-бот для пошуку роботи»*

Київ – 2024

# ЗМІСТ

# 1. НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ВИКОРИСТАННЯ

Дане технічне завдання поширюється на розробку чат-боту для Telegram, який надасть можливість налаштовувати нотифікації про появу нової вакансії на ряді сайтів для пошуків роботи.

Область застосування: комерційний продукт.

# 2. ПРИЧИНИ ДЛЯ РОЗРОБКИ

Підставою для розробки є завдання на виконання бакалаврського проєкту по освітньо-професійної програми “Комп’ютерні системи та мережі” спеціальності 123 “Комп’ютерна інженерія”, затверджене кафедрою Обчислювальної техніки Національного технічного Університету України “Київський Політехнічний інститут імені Ігоря Сікорського”.

# 3. МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою даного проєкту є створення якісного продукту, що є легким та ефективним у підтримці розробниками, а також зручним з точки зору користувачів.

# 4. ДЖЕРЕЛА РОЗРОБКИ

Джерелом розробки є науково-технічна література по комп’ютерних мережах і дистанційного навчання, публікації в періодичних виданнях, довідники по платформах дистанційного навчання, публікації в Інтернеті з цих питань.

# 5. ТЕХНІЧНІ ВИМОГИ

## 5.1. Вимоги до розробленого продукту

Простота використання – продукт повинен мати простий та інтуїтивно зрозумілий інтерфейс для користувача.

Швидкість – продукт має надавати сповіщення про появу нової вакансії в діапазоні до 30 секунд, тобто суттєво перевершити існуючі рішення на ринку.

Масштабованість – продукт повинен бути легко масштабованим у випадку появи багатьох користувачів.

### 5.2. Вимоги до програмного забезпечення

- Kubernetes кластер з версією > 1.30.0.

### 5.3. Вимоги до апаратної частини.

- 256 Mb оперативної пам’яті.
- 0.5 CPU.
- 25 Mb постійної пам’яті.

## 6. ЕТАПИ РОЗРОБКИ

| Назва етапів виконання | Дата |
|---|---|
| Складання і узгодження технічного завдання. | 01.04.2024 |
| Вивчення літератури та предметної області. | 15.04.2024 |
| Аналіз існуючих рішень. | 20.04.2024 |
| Налаштування інфрастуктури проекту | 22.04.2024 |
| Налаштування Redis та дизайн бази даних PostgresSQL. | 29.04.2024 |
| Розробка архітектури Go застосунку. | 03.05.2024 |
| Реалізація коду. | 05.05.2024 |
| Тестування | 15.05.2024 |
| Оформлення документації дипломного проєкту | 19.05.2024 |
| Передзахист | 15.06.2024 |
| Захист | 22.06.2024 |

# ПОЯСНЮВАЛЬНА ЗАПИСКА
# ДО ДИПЛОМНОГО ПРОЕКТУ

на тему: *«Telegram-бот для пошуку роботи»*

**Київ – 2024**

# ЗМІСТ

# ПЕРЕЛІК СКОРОЧЕНЬ

| | |
|---|---|
| GKE | Google Kubernetes Engine |
| SQL | Structured Query Language |
| SQLC | SQL Compiler |
| Redis | REmote DIctionary Server |
| IT | Informational Technology |
| JSON | JavaScript Object Notation |

# ВСТУП

У сучасному світі ринок праці змінюється з неймовірною швидкістю. З кожним днем з'являються нові вакансії, і конкуренція серед претендентів на роботу стає все більш жорсткою. У таких умовах важливо мати інструменти, які допоможуть швидко та ефективно знаходити підходящі вакансії.

Мета даного проєкту полягає в розробці бота для пошуку роботи, який забезпечить оперативне та зручне отримання інформації про нові вакансії. Цей бот буде моніторити різні ресурси з оголошеннями про роботу і надавати користувачам сповіщення про нові можливості, які відповідають їхнім критеріям.

Основні причини, чому було вирішено розробити цього бота, включають:

Швидкий доступ до інформації: Багато людей витрачають значну кількість часу на пошук роботи, постійно перевіряючи різні сайти та ресурси. Бот автоматизує цей процес, дозволяючи отримувати нові оголошення миттєво.

Покращення шансів на працевлаштування: Миттєве отримання інформації про нові вакансії дозволяє користувачам бути серед перших претендентів на роботу, що значно підвищує їхні шанси на успіх.

Зручність використання: Інтуїтивно зрозумілий інтерфейс та налаштування бота роблять його доступним для широкого кола користувачів, незалежно від їхнього рівня технічних знань.

Персоналізація: Бот дозволяє налаштовувати критерії пошуку відповідно до індивідуальних потреб користувача, що забезпечує більш релевантні результати.

Таким чином, розробка бота для пошуку роботи спрямована на створення ефективного інструменту, який допоможе користувачам швидко та зручно знаходити підходящі вакансії, підвищуючи їхні шанси на успішне працевлаштування.

# РОЗДІЛ 1. ЗАГАЛЬНИЙ ОГЛЯД ІСНУЮЧИХ РІШЕНЬ

На українському ринку пошуку роботи існує декілька основних гравців: robota.ua, DOU та Djinni. Кожна з цих платформ має свої особливості, що відрізняють їх одна від одної, включаючи способи реалізації підписок на нагадування про вакансії, а також свої унікальні переваги і підходи до реалізації підписок на нагадування про вакансії, що дозволяє користувачам обирати найбільш зручний і ефективний спосіб пошуку роботи відповід-но до їх потреб та побажань. У цьому розділі буде детально розглянуте кожне з цих рішень.

## 1.1 Огляд robota.ua

Robota.ua – це найбільший сайт для пошуку роботи в Україні, де зібрані тисячі вакансій з різних галузей. Robota.ua зібрав тисячі різноманітних вакансій з різних галузей і регіонів України, що дозволяє користувачам знайти роботу відповідно до їхніх потреб і навичок. Платформа пропонує вакансії для різних категорій праці, від стажувань до висококваліфікованих спеціалістів, що робить її відмінним вибором для будь-якого шукача роботи.

Незважаючи на популярність і широке визнання, платформа має кілька значних недоліків, які можуть ускладнити користувачам процес пошуку роботи: На даний момент, Robota.ua не надає можливість отримувати сповіщення про нові вакансії через Telegram, що є популярним месен-джером серед багатьох користувачів. Це обмеження змушує людей покладатися на менш зручні канали, такі як електронна пошта. Сповіщення про нові вакансії через електронну пошту можуть надходити з певною затримкою, що може негативно вплинути на шанси швидкого реагування на нові можливості.

## 1.2 Огляд work.ua

Work.ua є однією з провідних українських платформ для пошуку роботи, яка орієнтована на широкий спектр вакансій і кандидатів.

Даний ресурс пропонує велику кількість вакансій у різних галузях, таких як IT, фінанси, маркетинг, продажі, адміністрація, та інші. Платформа підходить як для стартових позицій, так і для вакансій вищого рівня.

Користувачі можуть створювати підписки на отримання сповіщень у декількох мессенджерах(таких як Viber та Telegram), про нові вакансії за певними критеріями. Є можливість налаштування критеріїв включає місце роботи, ключові слова, зарплатні очікування тощо.

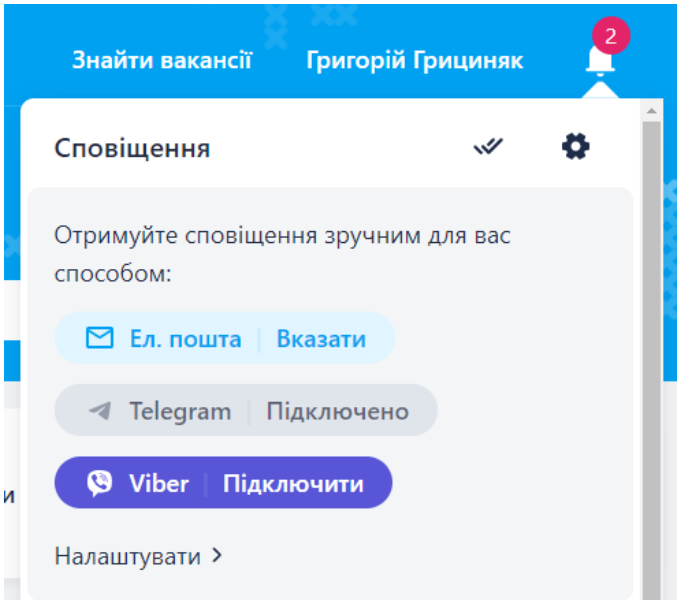



Рисунок 1.1 – приклад підписки на Work.ua

Розглянемо підписку на платформі Telegram. Вона має приємний інтерфейс та зрозумілий функціонал: користувач вводить пошуковий запит, а система надсилає сповіщення, якщо з’явилися будь-які вакансії, що відповідають цьому запиту.

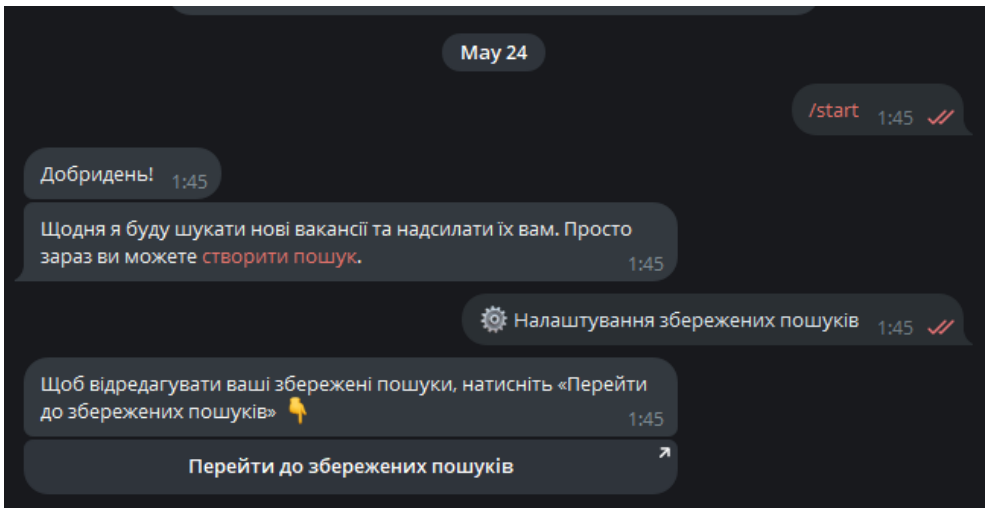



Рисунок 1.2 – приклад інтерфейсу на платформі Telegram

## 1.3 Огляд djinni.co

Djinni - це інноваційна українська платформа для пошуку роботи, спеціалізо-вана на IT-галузі. Ось основні характеристики та особливості платформи Djinni:

Спеціалізація на IT: Djinni орієнтований переважно на вакансії в сфері інформаційних технологій, включаючи розробку програмного забезпечення, веб-розробку, аналіз даних, тестування програмного забезпечення тощо.

Персоналізовані підписки: Користувачі можуть створювати персоналізовані підписки на вакансії, використовуючи різні фільтри, такі як технології, рівень досвіду, тип проекту, зарплатні очікування та локація.

Анонімність: Однією з ключових особливостей Djinni є можливість анонімного пошуку роботи. Кандидати можуть розглядати вакансії і надси-лати свої резюме анонімно, не розкриваючи свої особисті дані, поки вони не вирішать зв'язатися з роботодавцем.

Частота нагадувань: Користувачі можуть налаштовувати частоту отриман-ня повідомлень про нові вакансії - щоденно, щотижнево або негайно.

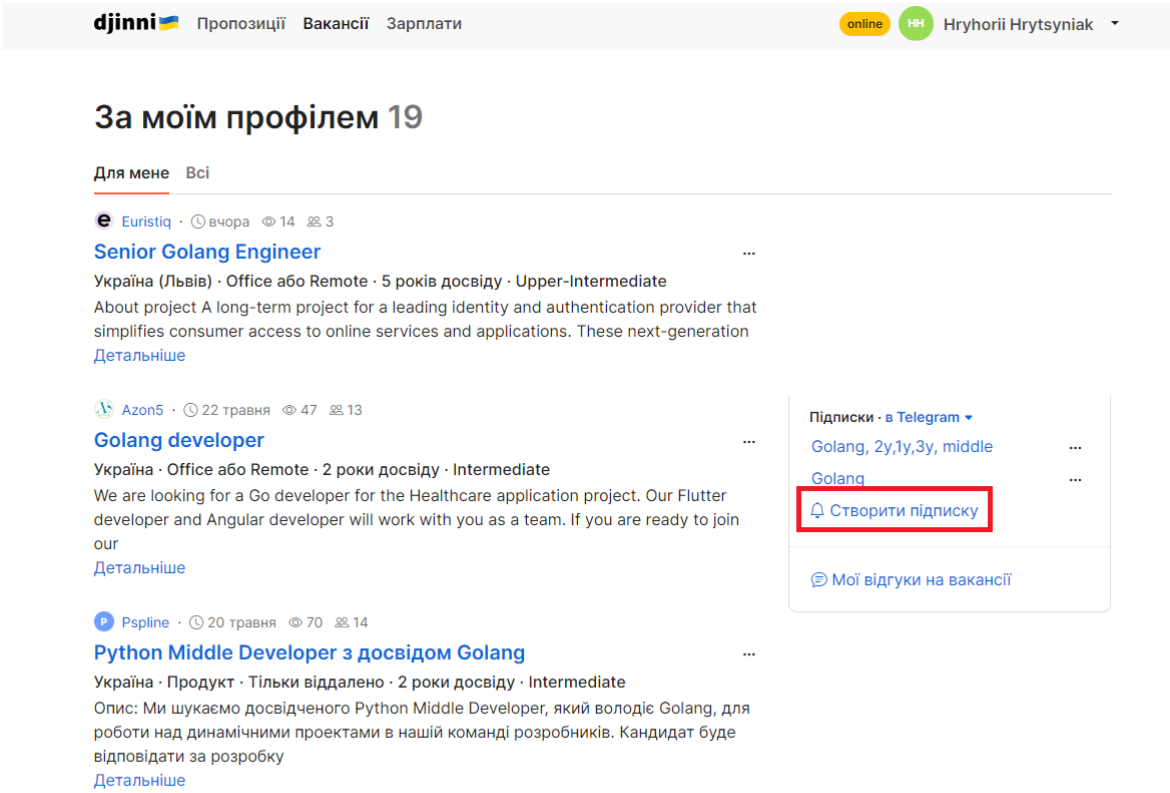



Рисунок 1.3 – створення підписки на djinni.co

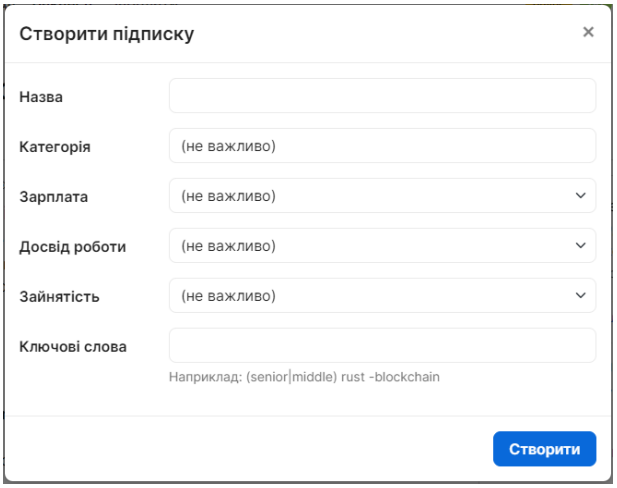



Рисунок 1.4 – ключові слова підписки на djinno.co

З цілями тестування було створено підписку «Golang». Як і очікувалося, було отримано сповіщення про появу нової вакансії «Senior Golang React Js developer у Softpositive» о 17:41.

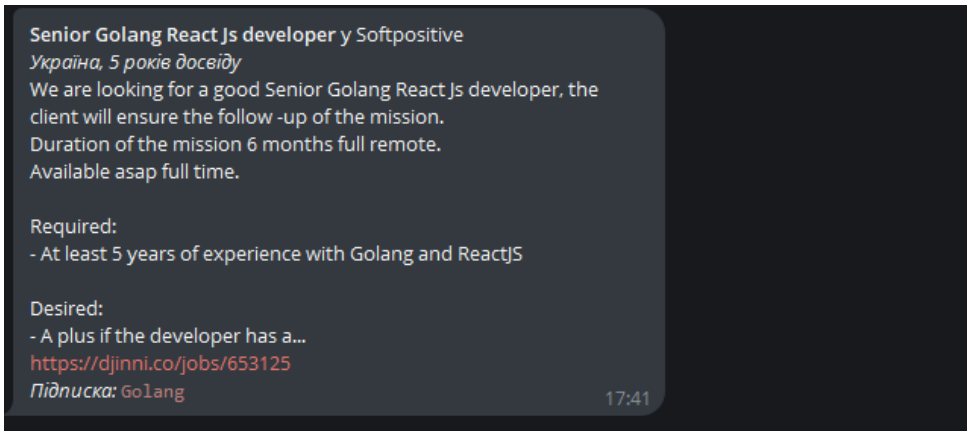



Рисунок 1.5 – сповіщення на djinno.co

Але в результаті відкриття цієї вакансії на платформі, можемо побачити, що вакансія було опублікована о 17:11, що свідчить про півгодинну затримку сповіщень на платформі.


Рисунок 1.5 – вакансія на платформі на djinno.co

# ВИСНОВОК ДО РОЗДІЛУ 1

Після аналізу існуючих рішень було виявлено, що не всі найпопулярніші платформи мають зручний функціонал для сповіщень. Серед тих платформ, де він є, було виявлено основну проблему: затримка сповіщень. У наступних розділах буде розроблено універсальне рішення, що дозволить створити сповіщення для всіх трьох платформ з максимальною затримкою 30 секунд. Користувачі даного сервісу будуть перевершувати конкурентів за швидкістю відгуку на вакансії, що дасть їм перевагу у працевлаштуванні.

# РОЗДІЛ 2. НАЛАШТУВАННЯ ІНФРАСТУКТУРИ ПРОЄКТУ

## 2.1 Технологія контейнеризації

### 2.1.2 Віртуалізація та її недоліки

Віртуалізація - це процес використання програмного забезпечення для створення віртуального ресурсу, який працює на шарі, відокремленому від фізичного обладнання. Найпоширеніший випадок використання віртуалізації - хмарні обчислення.

Завдяки віртуалізації можливо запускати кілька віртуальних машин (VM) на одному комп'ютері. Ці віртуальні машини є незалежними системами, але використовують ту саму фізичну IT-інфраструктуру та управляються гіпервізором.

Віртуалізація здобула значну популярність у сучасній сфері програмного забезпечення. Очікується, що світовий ринок додатків для віртуалізації буде оцінюватися у $5,76 мільярда до 2026 року. Це пов'язано з тим, що віртуалізація дозволяє користувачам отримувати доступ до додатків та функцій без їх встановлення на комп'ютері[8].

Ця хмарна технологія економить гроші, час і місце для зберігання, надаючи всі можливості хмарних обчислень. Однак, вона має ряд недоліків:

- Віртуалізація вимагає значних ресурсів оперативної пам'яті та процесора для одночасного запуску кількох операційних систем і копії віртуального обладнання. Перехід між приватними та публічними хмарами та дата-центрами ускладнює життєвий цикл розробки програмного забезпечення.
- Це монолітна технологія, яка запускає додатки як єдиний великий файл. Віртуалізація швидко споживає обчислювальні ресурси та цикли. Деякі

додатки можуть не працювати належним чином у віртуалізованих середовищах. Крім того, деякі старі або спеціалізовані додатки можуть бути несумісними з програмним забезпеченням для віртуалізації або потребуватимуть додаткової конфігурації для правильної роботи.

- Віртуалізовані середовища можуть бути менш зручними для масштабування порівняно з фізичними, особливо коли йдеться про додавання апаратних ресурсів

### 2.1.2 Контейнеризація

Контейнеризація - це форма віртуалізації операційної системи, яка використовує функції хостової операційної системи для ізоляції процесів і контролю їх доступу до пам'яті, дискового простору та процесорів[11].

Широке впровадження контейнеризації почалося з Docker, відкритої платформи для створення, розгортання та управління контейнеризованими додатками. З моменту свого впровадження у 2013 році, технологія контейнеризації та її екосистема значно еволюціонували.

Основними перевагами є:

• Зменшення використання ресурсів управління IT: Контейнеризація дозволяє краще використовувати ресурси IT, оскільки окремі контейнери можуть запускатись і управлятись незалежно. Це зменшує витрати на управління.

• Менші вимоги до розміру: Контейнери зазвичай мають менший розмір порівняно з традиційними віртуальними машинами, що дозволяє ефективніше використовувати дисковий простір і швидше розгортати контейнери.

• Швидке стартування і спрощені оновлення безпеки: Контейнери запускаються швидше, оскільки вони використовують ресурси хостової

операційної системи. Оновлення безпеки можуть бути простіше застосовувати, оскільки вони стосуються меншої кількості компонентів

Ці причини є критичними у випадку нашого додатку, саме тому буде використана ця технологія.

## 2.2 Docker

### 2.2.1 Контейнеризація Go додатку

В процесі контейнеризації використовуються наступні компоненти:

- Docker імедж (Docker image): статичний образ, який містить необхідні файли та конфігурацію для роботи програмного забезпечення всередині контейнера. Імеджі використовуються як основа для створення контейнерів. Вони містять всі необхідні залежності, бібліотеки, файли конфігурації і програмне забезпечення для запуску додатків всередині Docker контейнера.
- Docker контейнер (Docker container): створений і запущений інстанс імеджу Docker. Він є робочим середовищем, в якому працює програмне забезпечення. Контейнери ізольовані один від одного і від господарської системи.
- Dockerfile: це текстовий документ, що містить інструкції для автоматизованого створення імеджів Docker. Він визначає, як імедж Docker буде збудований, включаючи базовий образ, налаштування, копіювання файлів, налаштування змінних середовища і т.д.

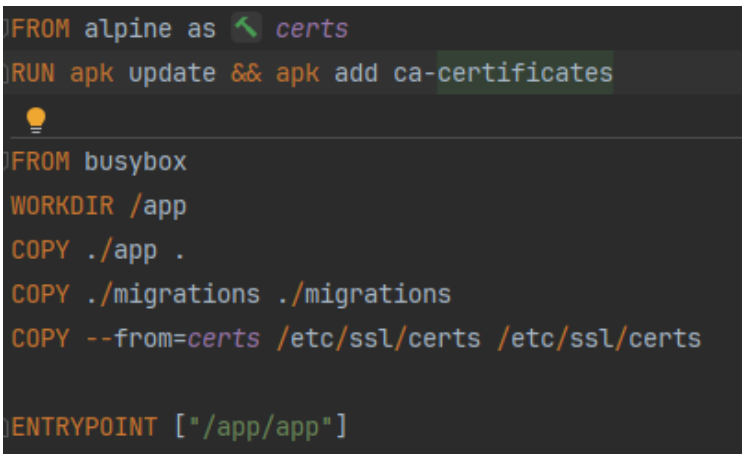



Рисунок 2.1 – Dockerfile для Go додатку

В ході побудови Dockerfile для проекту(рис. 2.1), було використано двухетапну побудуву (multi-stage build), дозволяє значно оптимізувати процес створення контейнерів, роблячи їх меншого розміру та більш ефективними. Цей підхід особливо корисний для компіляції та розгортання додатків, оскільки він дозволяє включати лише необхідні компоненти у фінальний образ[3].

Далі було використано команди docker build, docker tag, в результаті чого було локально збережено імедж додатку.

### 2.2.2 Dockerhub

Для того щоб Kubernetes кластер міг отримати Docker імедж, потрібно спочатку завантажити цей імедж на Docker Hub або інший Docker-репозиторій.

Docker Hub - це централізований репозиторій для зберігання імеджів Docker, який дозволяє легко ділитися імеджами з іншими користувачами і системами. Він надає безкоштовний рівень обслуговування для публічних репозиторіїв імеджів Docker. Однак для використання цих послуг вам необхідно мати обліковий запис на Docker Hub і зареєструватися(рис 2.2)[9].

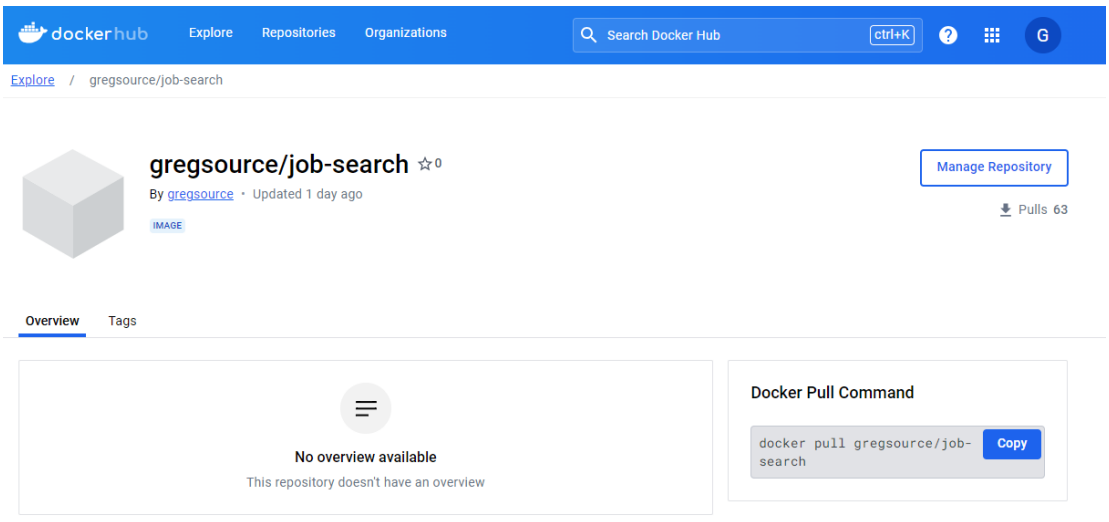



Рисунок 2.2 – Dockerfile для Go додатку

## 2.3 Kubernetes

### 2.3.1 Kubernetes

Основні види ресурсів у Kubernetes описуються у YAML-файлах, які використовуються для конфігурації об'єктів у кластері. Основні виді ресурсів що будуть використані у цьому проекті:

- Deployment - це об'єкт Kubernetes, який використовується для розгортання і керування додатками в контейнерах описується, скільки копій додатку (репліки) має бути запущено, які контейнери із якими імеджами вони мають використовувати, а також які конфігураційні параметри мають бути встановлені для кожного контейнера.
- Service - це об'єкт Kubernetes, який використовується для експонування додатків, що працюють у кластері. Він створює набір IP-адрес і портів, які відповідають умовам вибору, описаним у селекторі. Це дозволяє іншим додаткам і користувачам з'єднуватися з додатками.
- Pod - це найменша одиниця розгортання в Kubernetes. Він містить один або кілька контейнерів, які разом ділять мережу і сховище. Кожен контейнер у поді має свій власний IP-адрес[1].

### 2.3.2 Helm

Helm — це потужний інструмент для управління пакетами у Kubernetes, який дозволяє спрощувати розгортання, оновлення та управління додатками у кластері. Потужною та проривною особливістю є charts(чарти)

Chart - це пакет або шаблон, який містить всі необхідні ресурси Kubernetes для розгортання додатків. Він включає в себе файли конфігурації (manifests) для розгортання додатків, а також параметри, які можна настроювати під час розгортання. Helm дозволяє зберігати параметри розгортання у файлі значень, що дозволяє легко налаштовувати кожен конкретний розгортання. Також в helm є можливістю створювати “підчарти”, що спрощує процес побудови архітектури додатку[15].

На рис 2.3 зображена структура чарту для нашого проєкту.

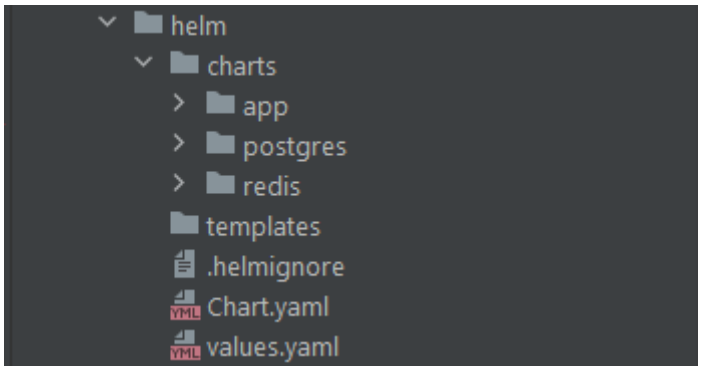



Рисунок 2.3 – Основна структура чарту проєкта

Варто звернути увагу на 4 основних компоненти:

- values.yaml: Це основний файл конфігурації, де описуються параметри за замовчуванням для додатку і всіх його компонентів. Він містить значення, які можна перевизначити при установці чарту. Це ефективний спосіб перенесення динамічних змінних на найвищий рівень, що дає можливість легко конфігурувати деплоймент для будь-якого середовища.

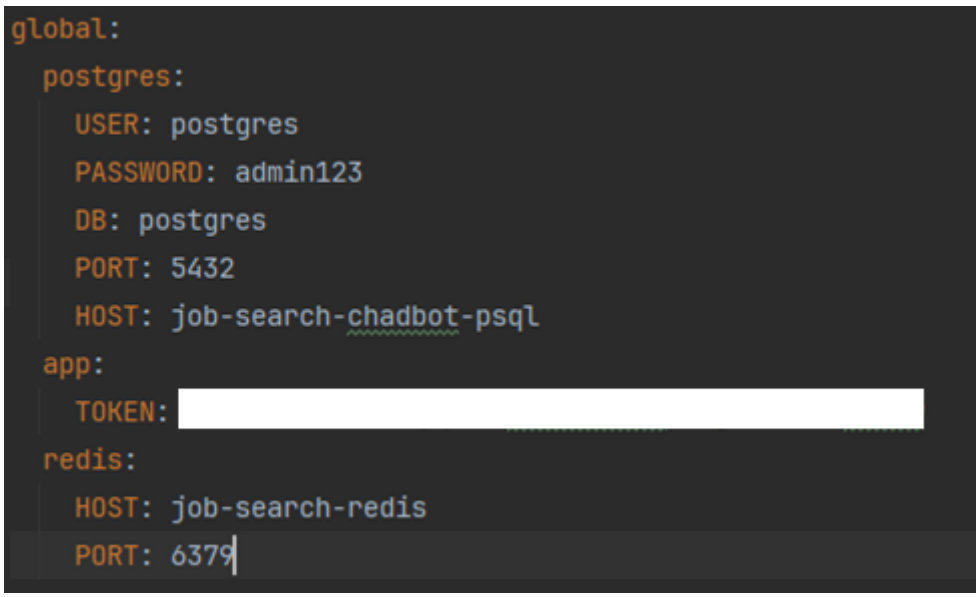



Рисунок 2.4 – приклад values.yaml

- app: Це сабчарт, який містить основний додаток.
- postgres: представляє собою базу даних PostgreSQL, яка використовується додатком. PostgreSQL є потужною реляційною базою даних з багатьма можливостями і налаштуваннями, які можна налаштовувати під різні потреби. Використання сабчарту дозволяє легко розгортати і управляти PostgreSQL в Kubernetes, забезпечуючи надійність та масштабованість для додатку.
- redis: redis представляє собою сервер Redis, який використовується для кешування та швидкодії додатку. Redis - це ключ-значення відкритий додаток для сховища даних.

## 2.4 Google Kubernetes Cluster

### 2.4.1 Причини використання

Google Kubernetes Engine (GKE) є сервісом контейнерного кластеру на базі Kubernetes від Google Cloud. Він пропонує розширені можливості для

розгортання, управління та масштабування додатків у контейнерах, порівняно з безкоштовним планом AWS для використання Kubernetes.

GKE забезпечує простоту налаштування та управління кластерами Kubernetes. Це дозволяє швидко створювати кластери, автоматизувати розширення та управління ресурсами. Він інтегрується з іншими сервісами Google Cloud, що робить його особливо зручним для користувачів, які вже використовують цей хмарний сервіс.

Однією з ключових переваг GKE є його висока доступність і масштабованість. Сервіс автоматично керує розширенням кластеру відповідно до навантаження, що дозволяє ефективно використовувати ресурси. Крім того, GKE забезпечує вбудовані механізми безпеки, такі як керування доступом і шифрування даних.

### 2.4.2 Налаштування

Для користування Google Kubernetes Engine (GKE) і отримання безкоштовної підписки потрібно зареєструватися в Google Cloud Platform (GCP) і створити обліковий запис. Google Cloud Platform (GCP) надає новим користувачам безкоштовний кредит у розмірі $300 на перший місяць використання. Це дозволяє випробувати різні послуги, включаючи Google Kubernetes Engine (GKE), без оплати.

Google Kubernetes Engine (GKE) — це високомасштабована система керування контейнерами, яка дозволяє автоматизувати розгортання, керування та масштабування контейнерних додатків, основана на оркестрації Kubernetes.

Для доступу до GCP можна використовувати веб-консоль GCP, яка надає зручний інтерфейс для управління послугами, включаючи GKE. Ця консоль дозволяє налаштовувати, моніторити і керувати різними аспектами проекту, використовуючи інтуїтивно зрозумілі інструменти та панелі керування. Також

доступний інтерфейс командного рядка для автоматизації процесів та взаємодії з платформою.

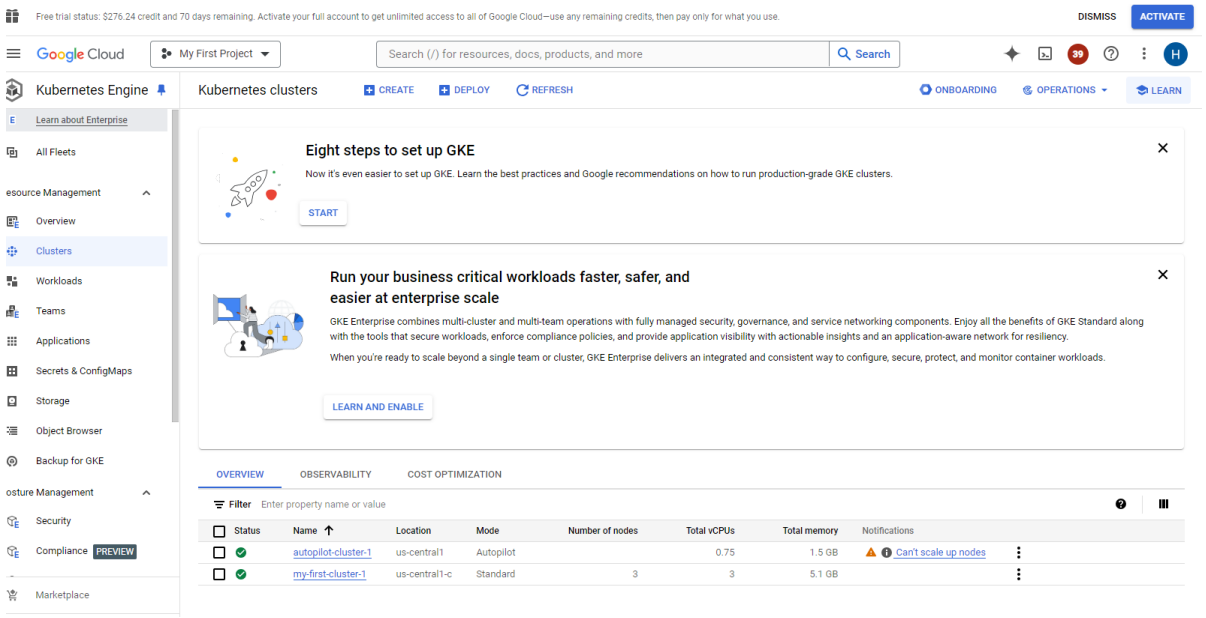



Рисунок 2.5 – приклад веб-консолі GCP

Для дослідження Google Kubernetes Engine (GKE) було вирішено створити Autopilot кластер.

Autopilot — це нове керування для GKE, яке автоматизує адміністрування кластера. Це відрізняється від традиційного підходу до керування Kubernetes тим, що замінюєте вручну налаштовані вузли на повністю керовані, автоматично масштабовані і адміністровані контейнерні середовища[25].

Ось деякі причини, чому було обрано даний спосіб:

• Автоматизація і масштабування: Autopilot кластери автоматично масштабуються в залежності від потреб додатку. Немає необхідності вручну керувати розміром кластера або вузлів.

• Управління витратами: Google Cloud автоматично налаштовує потужність кластера відповідно до потреб додатку, що може знизити загальні витрати на інфраструктуру.

• Зменшення адміністративного навантаження: Autopilot керує аспектами, такими як масштабування, конфігурація, резервне копіювання та оновлення кластера. Це звільняє вас від багатьох рутинних адміністративних завдань.

• Швидкість розгортання і налаштування: Створення Autopilot кластера є швидким і простим процесом, який дозволяє швидко розпочати розробку та тестування.

В порівнянні з традиційним підходом до керування Kubernetes, де ви самостійно керуєте вузлами, Autopilot надає більш автоматизований і менш працездатний підхід до управління кластерами. Традиційний підхід дозволяє більше контролю над конфігурацією і налаштуванням кластера, але вимагає більше уваги до адміністрування.

Autopilot ідеально підходить для сучасних архітектур мікросервісів і контейнерних додатків, оскільки спрощує рутинні завдання і дозволяє швидше реагувати на зміни в навантаженні та вимоги до додатків.

### 2.4.3 Моніторинг

Особливо важливими стали створені моніторингові дашборди(рис. 2.6) та налаштовані алерти. Вони допомагають вчасно виявляти потенційні проблеми, моніторити стан кластера і вчасно реагувати на зміни в навантаженні та трафіку.

Prometheus - це система моніторингу і сповіщення, яка спеціалізується на зборі та зберіганні даних часових рядів. Вона здатна автоматично збирати метрики з моніторюваних компонентів за допомогою свого власного механізму збору даних. Prometheus також підтримує сповіщення на основі заданих умов[12].

Ці інструменти значно спростили процес аналізу та управління інфраструктурою, що дозволило мені зосередитися на більш важливих завданнях, таких як розробка і вдосконалення додатків.

Таким чином, Autopilot кластер і моніторингові інструменти разом створили ефективну інфраструктуру для моїх досліджень, що дозволяє аналізувати та оптимізувати навантаження і трафік у реальному часі.

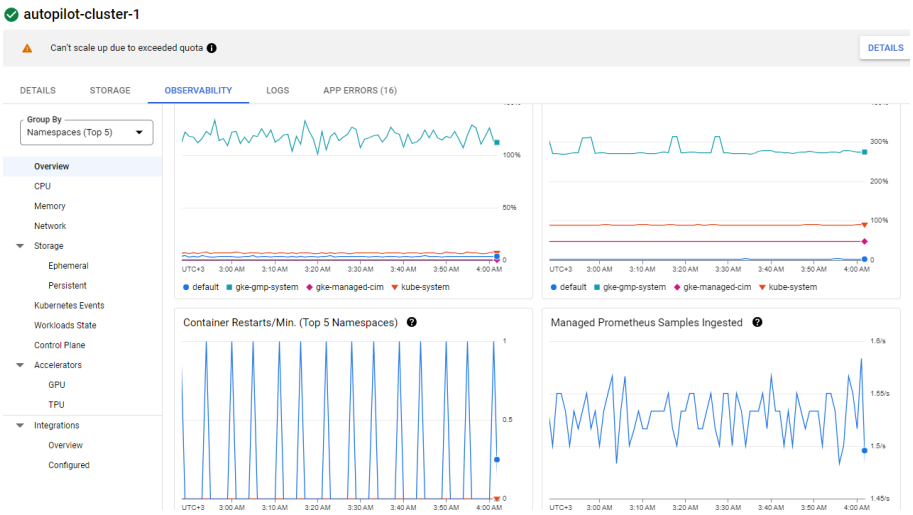



Рисунок 2.6 – приклад моніторингово інтерфейсу

# ВИСНОВОК ДО РОЗДІЛУ 2

У Розділі 2 "Налаштування інфраструктури проєкту" детально розглянуто сучасні технології, які використовуються для ефективного управління інфраструктурою програмного забезпечення, з особливим акцентом на контейнеризацію та її інструменти.

Docker був представлений як один із найпопулярніших інструментів для контейнеризації, що значно спрощує процес створення, розгортання та управління контейнеризованими додатками. Було розглянуто контейнеризацію Go-додатків за допомогою Docker, а також використання Dockerhub для зберігання та розповсюдження Docker-образів.

Ключове місце в розділі займає Kubernetes, який є потужною системою оркестрації контейнерів. Kubernetes забезпечує автоматизоване розгортання, масштабування та управління контейнеризованими додатками, що робить його незамінним інструментом для сучасної розробки. Додатково розглянуто Helm, інструмент для управління Kubernetes-ресурсами, що спрощує роботу з Kubernetes за допомогою пакетів (чартів).

Використання Google Kubernetes Engine (GKE) для розгортання Kubernetes-кластерів було детально проаналізовано, зокрема, розглянуто причини вибору цього сервісу, його переваги, процес налаштування та можливості для моніторингу кластерів. Інтеграція GKE з іншими сервісами Google Cloud Platform забезпечує додаткові переваги для розробників та DevOps-інженерів.

Розділ також торкнувся теми моніторингу та логування, що є критично важливими для забезпечення стабільності та продуктивності розгорнутих додатків. Були розглянуті інструменти, такі як Prometheus для збирання та візуалізації метрик, а також для централізованого збирання, обробки та аналізу логів.

Окрім технічних аспектів, у розділі також приділялася увага питанням безпеки та кращим практикам для забезпечення надійності та стійкості розгорнутих систем. Зокрема, було розглянуто питання мережевої безпеки, управління секретами, ролей та дозволів у Kubernetes, а також стратегії резервного копіювання та відновлення.

У підсумку, Розділ 2 продемонстрував, як використання технології контейнеризації та інструментів для її управління, таких як Docker і Kubernetes, забезпечує високу гнучкість, ефективність використання ресурсів та масштабованість. Google Kubernetes Engine додатково спрощує процес налаштування та управління кластерами, що робить його привабливим вибором для впровадження у сучасні проєкти з розробки програмного забезпечення.

# РОЗДІЛ 3. НАЛАШТУВАННЯ REDIS ТА ДИЗАЙН БАЗИ ДАНИХ

## 3.1 Основні елементи системи

### 3.1.1 Опис системи

Система складається з кількох основних компонентів(дод. 2), які забезпечують її функціональність.

- Серверна частина на Golang: обробляє запити від користувачів, виконує бізнес-логіку та взаємодіє з базою даних.
- Redis використовується для кешування даних: що підвищує швидкодію системи, особливо для операцій, які часто повторюються.
- PostgreSQL: є основною базою даних, де зберігаються всі необхідні дані для функціонування системи.

Комбінація цих компонентів створює потужну і надійну основу для виконання бізнес-логіки і обробки даних. Ця архітектура дозволяє системі масштабуватися і залишатися продуктивною навіть при високому навантаженні, що є критично важливим для сучасних веб-додатків.

### 3.1.1 Взаємодія елементів системи

Користувачі взаємодіють із системою через Telegram-бота, відправляючи запити, такі як створення, редагування або отримання підписок. Ці запити надходять на серверну частину, написану на Golang, яка обробляє їх відповідно до бізнес-логіки. Для підвищення продуктивності система використовує Redis для кешування часто запитуваних даних. Це зменшує навантаження на базу даних і прискорює доступ до інформації. Всі постійні дані зберігаються в

PostgreSQL, забезпечуючи надійне і стійке зберігання. Даний підхід дає наступні переваги:

- Швидкодія: Redis, як кеш-система, забезпечує швидкий доступ до даних, зберігаючи їх у пам'яті. Це дозволяє значно прискорити відповідь системи на запити користувачів.
- Масштабованість: Redis підтримує різні методи кешування, які дозволяють ефективно масштабувати систему під високі навантаження.
- Оптимізація навантаження на базу даних: Використання Redis для кешування даних дозволяє зменшити навантаження на основну базу даних, оскільки часто запитувані дані можуть бути отримані без необхідності звертатися до дискових операцій.
- Надійність та стійкість даних: PostgreSQL забезпечує надійне зберігання постійних даних, забезпечуючи інтегрітет та захист даних.
- Бізнес-логіка та взаємодія: Golang використовується для реалізації серверної частини, обробки бізнес-логіки та взаємодії з Redis і PostgreSQL, що забезпечує ефективну та зручну архітектуру програмної системи.

Така архітектура створює потужну та надійну інфраструктуру, відповідаючи вимогам сучасних інформаційних технологій.

## 3.2 Redis

### 3.2.1 Особливості та переваги

Redis – це високо продуктивна, низько затримкова, розподілена система зберігання даних у пам'яті (in-memory data store), яка використовується як база даних, кеш, брокер повідомлень та черга повідомлень. Redis підтримує різноманітні структури даних, включаючи строки, списки, множини, хеші,

зворотні множини з пріоритетами (sorted sets) та інші, що робить його надзвичайно гнучким інструментом для розробників[6].

Основна характеристика Redis полягає у зберіганні даних у оперативній пам'яті, що забезпечує високу швидкість доступу та обробки даних. Незважаючи на те, що дані зберігаються в пам'яті, Redis підтримує функції збереження даних на диск, що дозволяє забезпечити їх стійкість та збереження у випадку перезапуску системи. Реплікація даних у Redis дозволяє створювати резервні копії та забезпечувати високу доступність системи.

Підтримка публікації та підписки (pub/sub) робить Redis ефективним брокером повідомлень, що дозволяє клієнтам підписуватися на певні канали і отримувати повідомлення в реальному часі. Завдяки зберіганню даних у пам'яті та ефективним алгоритмам, Redis забезпечує дуже високу продуктивність для операцій зчитування та запису, що робить його популярним вибором для кешування даних, обробки сеансів користувачів, реалізації черг завдань та обміну повідомленнями між сервісами.

### 3.2.1 Налаштування в Kubernetes

Процес налаштування Redis в середовищі Kubernetes, відбувається за допомогою конфігураційного файлу для деплойменту. Конфігурація дозволяє налаштувати основні параметри для розгортання Redis, забезпечуючи високу доступність і продуктивність сервісу.

Процес налаштування почався з визначення базових метаданих для деплойменту, таких як назва та мітки. Ці метадані забезпечують унікальну ідентифікацію та полегшують управління ресурсами в кластері Kubernetes[14].

Далі було налаштовано кількість реплік, що визначає скільки екземплярів Redis буде розгорнуто. Це забезпечує масштабованість та високу доступність сервісу, дозволяючи розподіляти навантаження між кількома екземплярами.

Секція selector визначає критерії для вибору підходящих під для цього деплойменту, а секція template описує шаблон для створення нових підів. У шаблоні вказано мітки та анотації для подів, що дозволяє легко ідентифікувати та керувати ними.

Важливою частиною конфігурації є секція spec, де визначаються налаштування для контейнерів Redis. Це включає ім'я контейнера, політику безпеки, образ Docker, що буде використаний, політику завантаження образів та порти, на яких буде працювати Redis. Крім того, було налаштовано ресурси, які будуть виділені для контейнера, що допомагає забезпечити стабільну роботу сервісу.

Також була реалізована можливість підключення до зовнішніх секретів за допомогою imagePullSecrets, що забезпечує безпечний доступ до приватних образів Docker. Налаштування безпеки для подів (securityContext) та контейнерів забезпечує додатковий рівень захисту.

Для управління сховищами даних у Redis було використано секції volumeMounts та volumes, що дозволяє підключати зовнішні диски та налаштовувати персистентність даних. Використання налаштувань nodeSelector, affinity та tolerations дозволяє визначати політику розміщення подів на вузлах кластера, забезпечуючи оптимальне використання ресурсів та стійкість до збоїв.

Таким чином, налаштування Redis у середовищі Kubernetes було здійснено з урахуванням всіх необхідних аспектів для забезпечення стабільної, безпечної та ефективної роботи сервісу. Це дозволило інтегрувати Redis у існуючу інфраструктуру, забезпечуючи високий рівень доступності та продуктивності.

### 3.2.1 Практична інтеграція з застосунком

Списки у Redis є впорядкованими колекціями елементів, які зберігаються у визначеній послідовності. Вони дозволяють додавати елементи як на початок,

так і на кінець списку, що робить їх ідеальними для реалізації черг та стеків. Операції додавання та видалення елементів у списках виконуються за константний час O(1), що забезпечує швидкий доступ до даних і мінімальні затримки. Це особливо важливо для систем, які потребують обробки великої кількості запитів у реальному часі.

У своєму проєкті було викоритсано Redis для зберігання та обробки різних типів даних. Було створено структуру RedisCache, яка забезпечує зручний інтерфейс для взаємодії з Redis. Основні функції включають підключення до Redis, зберігання та отримання даних користувачів, а також управління підписками.

Для зберігання даних користувачів було використано хеші (hashes) у Redis. Це дозволяє ефективно зберігати та швидко доступатися до даних користувачів за унікальним ім'ям користувача. Дані користувачів серіалізуються у формат JSON і зберігаються у вигляді ключ-значення.

Для зберігання підписок було обрано списки (lists) у Redis. Списки дозволяють ефективно обробляти послідовності даних, підтримуючи операції додавання та видалення елементів з кінців списку. Це особливо корисно для реалізації черг. Дані підписок також серіалізуються у формат JSON перед зберіганням у Redis.

Крім списків та хешів, у проєкті також використовувалися строки (strings) для зберігання простих значень. Це дозволяє зберігати та отримувати дані за ключем, забезпечуючи простий і швидкий доступ до інформації.

Підключення до Redis здійснювалося за допомогою клієнта Redis, який забезпечує необхідні методи для взаємодії з базою даних. Перевірка підключення виконувалася за допомогою команди Ping. Це дозволяє переконатися, що підключення встановлено успішно і Redis готовий до обробки запитів.

Для обробки даних користувачів використовувалися функції збереження (SetUser) та отримання (GetUser) даних. Ці функції забезпечують серіалізацію

та десеріалізацію даних користувачів у формат JSON, що дозволяє зручно зберігати складні об'єкти у Redis.

Для роботи з підписками використовувалися функції додавання (SetSubscription) та отримання (GetSubscription) підписок. Ці функції дозволяють ефективно керувати чергами підписок, додаючи нові елементи у список та отримуючи елементи з нього.

Таким чином, використання різних структур даних у Redis дозволило забезпечити ефективне зберігання та обробку інформації у проєкті. Завдяки високій продуктивності та гнучкості Redis, вдалося значно підвищити швидкість доступу до даних та оптимізувати роботу системи.

## 3.3 База даних

### 3.1.1 PostgresSQL

База даних (БД) - це організована колекція даних, яка зберігається та упорядковується з метою ефективного доступу, оновлення та управління ними. Вона використовується для зберігання інформації, яка може бути оброблена, аналізована і використована для різних цілей. Бази даних грають ключову роль у сучасних інформаційних системах, забезпечуючи надійне зберігання і консистентний доступ до даних[7].

Я вибрав PostgreSQL для свого проєкту з кількох причин. По-перше, PostgreSQL є потужною об'єктно-реляційною системою керування базами даних, яка підтримує багато функцій стандартного SQL і включає ряд розширень, що полегшують розробку і оптимізацію додатків. Вона також підтримує роботу з JSON-даними і може використовуватися як NoSQL-база даних, що є важливим для проектів з динамічними схемами даних.

Друга причина - це гнучкість масштабування PostgreSQL. Система дозволяє використовувати горизонтальне масштабування, що дозволяє розширювати обсяг і оброблювати більше даних без значних затрат на технічне обслуговування.

Крім того, PostgreSQL має активну спільноту розробників і користувачів, яка забезпечує швидку підтримку і вирішення проблем, що виникають під час розробки і впровадження проектів.

Загалом, PostgreSQL вибраний через свою потужність, гнучкість і підтримку, що робить його ідеальним вибором для реалізації надійних і ефективних баз даних у проектах будь-якої складності.

### 3.1.2 Налаштування в Kubernetes

StatefulSet є контролером Kubernetes, який призначений для управління становими додатками, такими як бази даних. Він забезпечує гарантоване розміщення і унікальні імена для кожного поду в межах мережевої системи. Це дозволяє зберігати стан і забезпечує стійкість додатків під час масштабування та оновлення[2].

В метаданих StatefulSet визначається ім'я та мітки, які ідентифікують ресурс. Ці мітки можна використовувати для відбору подів, які відносяться до конкретного StatefulSet. Це важливо для ідентифікації та організації ресурсів в середовищі Kubernetes[4].

Кількість реплік StatefulSet визначається параметром replicas. Це кількість ідентичних копій, які створюються для керованого ресурсу. Від цього параметра залежить навантаження, яке може бути оброблене, а також відмовостійкість рішення.

У Kubernetes сервіси використовуються для надання стійкого доступу до додатків. Для StatefulSet задається сервіс, який дозволяє іншим додаткам звертатися до бази даних за допомогою DNS імені сервісу.

Шаблон поду визначає конфігурацію контейнера, який містить базу даних PostgreSQL. Включає параметри, такі як образ Docker, політику підтягування образу, налаштування пристанцій, обмеження ресурсів, анотації і мітки, які використовуються для ідентифікації і організації ресурсів.

Контекст безпеки визначає політики безпеки, які використовуються для контейнера. Це може включати встановлення прав доступу до файлів, обмеження привілеїв і налаштування SELinux або AppArmor.

Проби живості (liveness probes) та готовності (readiness probes) використовуються для перевірки стану контейнера. Для PostgreSQL це може включати виконання SQL-запитів для перевірки доступності бази даних.

Точки монтування визначаються для збереження даних PostgreSQL, що зберігаються під час перезапуску контейнера.

### 3.3.3 Дизайн бази даних

У цьому розділі буде розглянуто деталізований опис маленької бази даних для зберігання інформації про користувачів та їх підписки.

Таблиця users призначена для зберігання основної інформації про користувачів. Вона має просту структуру і включає такі поля:

- username: Ім'я користувача, що може досягати до 28 символів у довжину.
- status: Статус користувача, який може бути до 20 символів у довжину.
- chatID: Унікальний ідентифікатор чату, який є ключем таблиці.

Ця таблиця дозволяє зберігати ідентифікатори користувачів та їх поточний статус у системі. Вона проста у використанні та ефективна для зберігання основної інформації.

Таблиця subscriptions призначена для зберігання інформації про підписки користувачів. Вона включає наступні поля:

- id: Унікальний ідентифікатор підписки, який автоматично збільшується для кожного нового запису.
- chatID: Унікальний ідентифікатор чату користувача, який використовується для ідентифікації підписок.
- subscription: Назва підписки, що може досягати до 255 символів у довжину.
- type: Тип підписки, який може досягати до 20 символів у довжину.
- updatetime: Час останнього оновлення підписки, представлений у форматі цілочисельного значення.

Додатково, у таблиці subscriptions встановлено унікальний ключ (unique constraint) на комбінацію полів chatID, subscription і type, що забезпечує унікальність записів підписок для кожного користувача.

### 3.3.4 Практична інтеграція з застосунком

Для зручності інтеграції з базою даних PostgreSQL було створено клас Postgres. Цей клас містить конфігурацію підключення та методи для підключення та виконання запитів до бази даних[16].

Конфігурація підключення містить дані про сервер бази даних, такі як ім'я користувача, пароль, хост, порт та назву бази даних. Перш ніж розпочати взаємодію з базою даних, було застосовано міграції. Це дозволяє автоматично створювати необхідні таблиці та структури даних. Підключаємося до бази даних та ініціалізуємо об'єкт, який дозволяє виконувати SQL запити до бази даних[25].

Як результат, цей підхід дозволяє ефективно та безпечно інтегрувати базу даних PostgreSQL з застосунком на мові програмування Go. Застосування міграцій перед підключенням забезпечує автоматичне створення необхідних структур даних, що робить розробку більш зручною та ефективною. Такий

підхід дозволяє швидко створити надійну систему зберігання даних, яка легко масштабується та супроводжується.

### 3.3.5 SQLC

SQLC - це інструмент для автоматизації створення SQL запитів у Go. Він дозволяє визначати SQL запити у вигляді SQL файлів та автоматично генерує відповідний безпечний код на Go для їх виконання. Це значно спрощує розробку, зменшує шанси на помилки та полегшує тестування[17].

Особливості SQLC:

- Автоматична генерація коду: SQLC зчитує SQL файли з визначених каталогів та створює об'єктно-орієнтований код на Go для кожного SQL запиту.
- Підтримка PostgreSQL: з різними типами баз даних, включаючи PostgreSQL, що робить його ідеальним вибором для проектів, що використовують цю СУБД.
- Конфігурація через файл sqlc.yaml: Для налаштування SQLC використовується конфігураційний файл sqlc.yaml, де визначаються шляхи до SQL файлів, налаштування генерації коду та інші параметри.
- Підтримка міграцій: SQLC може використовуватися для автоматичної генерації міграційних скриптів та керування схемою бази даних.

# ВИСНОВОК ДО РОЗДІЛУ 3

Даний розділ розглядає налаштування та дизайн баз даних у середовищі Kubernetes, з фокусом на Redis, PostgreSQL та використання SQLC для автоматичної генерації SQL запитів.

У першій частині було розглянуто загальну архітектуру системи, підхід та переваги використаних компонентів.

У другій частині розглянуто Redis, його особливості та переваги, включаючи високу швидкодію, підтримку різних типів даних та підтримку транзакцій. Налаштування Redis в Kubernetes охоплює використання StatefulSet для забезпечення стабільності, PersistentVolumeClaim для зберігання даних та ConfigMap для конфігурації. Практична інтеграція Redis з застосунком полягає у кешуванні запитів до PostgreSQL, що сприяє зниженню навантаження на базу даних та покращенню продуктивності.

У третій частині описано PostgreSQL, який використовується для зберігання даних про користувачів та підписки. Налаштування PostgreSQL в Kubernetes включає використання StatefulSet, PersistentVolumeClaim, ConfigMap та Service для забезпечення доступу. Дизайн бази даних PostgreSQL містить таблиці users та subscriptions, з використанням індексів та обмежень для оптимізації запитів.

SQLC використовується для автоматичної генерації безпечних SQL запитів та відповідного коду на Go, що спрощує розробку та зменшує час на тестування. SQLC дозволяє визначати SQL запити у вигляді SQL файлів та генерує необхідний код для виконання цих запитів.

Було додано чарти redis postgres , а також створено модулі database, cache та sqlc для подальшої реалізації бізнес-логіки.

# РОЗДІЛ 4. РОЗРОБКА ДОДАТКУ

## 4.1 Організація розробки

### 4.1.1 Система контролю версій

Система контролю версій (VCS) або програмне забезпечення для контролю версій автоматизує процес контролю версій. Він відстежує зміни у файлі або наборі файлів з часом, тому вам не потрібно керувати версіями файлів вручну або за допомогою спеціальних сценаріїв автоматизації.

Серед багатьох існуючих VCS, найбільш популярною та широко використовуваною є Git. Її популярність зумовлена кількома ключовими факторами, які роблять Git вибором номер один для розробників і команд у всьому світі.

Основна перевага Git полягає в можливості відстеження змін у коді та документах. Кожна зміна зберігається у вигляді окремого коміту, що дозволяє повертатися до попередніх версій файлів, аналізувати історію змін та визначати, коли і чому були внесені ті чи інші зміни. Це особливо корисно в процесі написання дипломної роботи, коли потрібно часто редагувати текст, додавати нові розділи або виправляти помилки[10].

Ще одна важлива перевага використання Git полягає в можливості створення гілок (branches) для розробки нових функцій або внесення змін без порушення основної гілки проекту. Це дозволяє експериментувати з новими ідеями, тестувати їх, а потім, якщо все працює правильно, об’єднувати їх з основною гілкою. У контексті дипломної роботи це означає, що можна працювати над окремими розділами або частинами проекту незалежно один від одного, а потім об’єднувати їх у фінальний документ.

Git також забезпечує безпеку та резервне копіювання. Всі зміни зберігаються у віддаленому репозиторії, що гарантує збереження роботи навіть у разі втрати доступу до локального комп’ютера. Це особливо важливо для довготривалих проектів, таких як написання дипломної роботи, де втрати даних можуть бути критичними[13].

Таким чином, використання системи контролю версій Git під час розробки дипломної роботи дозволяє ефективно відстежувати зміни, забезпечувати співпрацю між учасниками проекту, експериментувати з новими ідеями, гарантувати безпеку даних та зберігати всі версії проекту. Це робить процес розробки більш організованим, надійним та зручним.

### 4.1.2 Github

Існує кілька популярних платформ для хостингу Git-репозиторіїв, таких як GitHub, GitLab, Bitbucket, та SourceForge. Кожна з них має свої особливості, переваги та недоліки, які слід враховувати при виборі інструмента для управління версіями та співпраці над проектами.

Серед цих платформ GitHub виділяється завдяки своїм численним перевагам: він пропонує зручний і інтуїтивно зрозумілий інтерфейс, який спрощує управління проектами та полегшує роботу з репозиторіями. Крім того, потужні інструменти для рев'ю коду та обговорення змін, такі як pull requests, роблять процес співпраці ефективнішим.

Ще однією важливою перевагою GitHub є його інтеграція з безліччю сторонніх сервісів і інструментів, що дозволяє розробникам легко налаштовувати свої робочі процеси. GitHub Actions, наприклад, надає потужні можливості для налаштування CI/CD процесів, що дозволяє автоматизувати тести, збірку та розгортання додатків. Крім того, GitHub пропонує безкоштовний хостинг для відкритих репозиторіїв, що робить його доступним для широкого кола розробників та проектів.

Реєстрація на GitHub та створення репозиторію є основними кроками для початку роботи з системою контролю версій і управління кодом:

- Спочатку необхідно зареєструвати обліковий запис на GitHub.
- Наступним кроком є створення нового репозиторію. Для цього на головній сторінці облікового запису GitHub потрібно натиснути кнопку "New" або перейти до розділу "Repositories" і вибрати "New repository". У формі створення репозиторію необхідно вказати назву проекту, додатково можна додати опис та вибрати, чи буде репозиторій публічним або приватним.
- Потім потрібно додати файли проекту до репозиторію за допомогою команд git add для індексації файлів та git commit для збереження змін. Наприклад, команда git add . додає всі файли в поточній директорії, а команда git commit -m "Initial commit" зберігає зміни з повідомленням "Initial commit"[9].
- Нарешті, щоб завантажити локальні зміни на GitHub, використовується команда git push origin main (або master, залежно від налаштувань репозиторію). Це завантажує всі збережені зміни до віддаленого репозиторію на GitHub, роблячи їх доступними для інших користувачів і розробників(рис. 4.1).

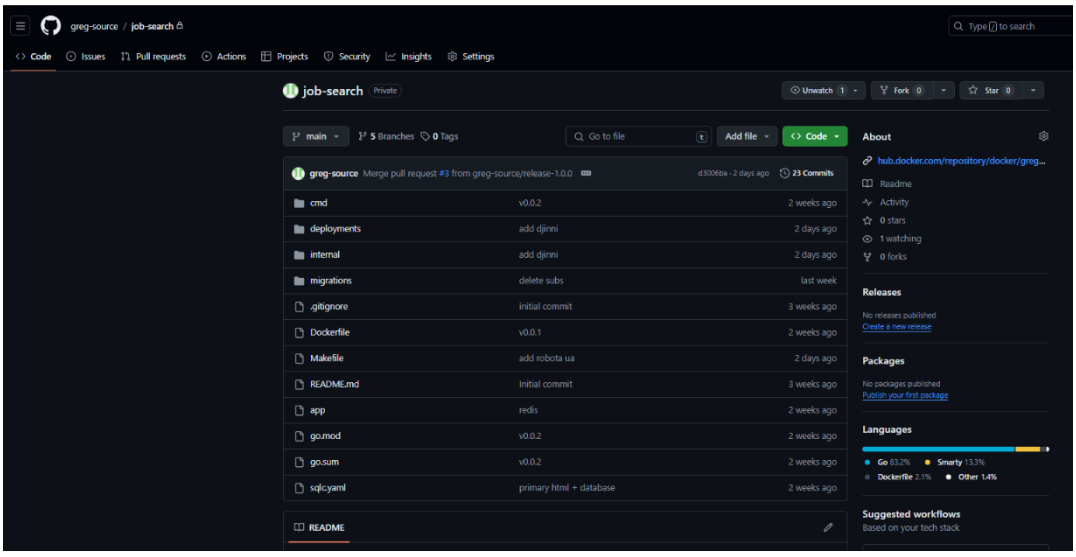



Рисунок 4.1 – публічний репозиторій проєкту

### 4.1.3 Методологія розробки

У розробці програмного забезпечення існує кілька методів, які дозволяють ефективно організовувати процес роботи над проектом, забезпечувати якість коду і зручність співпраці між розробниками. Деякі з найбільш поширених методів включають Feature Branch Workflow, Gitflow Workflow, Forking Workflow, а також методи Continuous Integration (CI) та Continuous Deployment (CD)[20].

Для даного проєкту було вибравно Gitflow Workflow. Це було обумовлено кількома важливими причинами, що зробили цей підхід найкращим вибором для ефективного управління процесом розробки:

По-перше, структура Gitflow Workflow забезпечує чітке розмежування між різними етапами розробки, що є особливо корисним при одноосібній розробці. По-друге, Gitflow Workflow пропонує зручний механізм управління новими функціями та виправленнями через створення окремих гілок для кожної функції (feature branches) та виправлення помилок (hotfix branches). Це дозволяє працювати над кожною новою функцією або виправленням окремо, зменшуючи ризик конфліктів при злитті коду і забезпечуючи можливість ретельного тестування перед інтеграцією в основний код.

По-друге, додаткові гілки для підготовки до релізу (release branches) дозволяють здійснювати фінальне тестування та стабілізацію коду перед випуском. Це гарантує, що всі релізи будуть ретельно перевірені і стабільні, що є критично важливим для забезпечення якості та надійності програмного забезпечення.

Таким чином, вибір Gitflow Workflow для даного проекту був обумовлений його здатністю забезпечити структурованість, ефективність та надійність процесу розробки, що є особливо важливим для підтримання високої якості коду та стабільності релізів.

## 4.2 Архітектура

### 4.2.1 Clean Architecture

Чиста архітектура, запропонована Робертом Мартіном[5], є концепцією побудови програмних систем з чіткими межами та залежностями між різними рівнями системи. Основна ідея полягає в поділі системи на кілька шарів, де кожен шар відповідає за свою конкретну функціональність і залежить тільки від шарів, які знаходяться під ним. Зовнішні залежності, такі як фреймворки та бази даних, знаходяться у зовнішніх шарах. Залежності завжди спрямовані всередину, що означає, що внутрішні шари не знають про зовнішні.

У чистій архітектурі є кілька основних компонентів: сутності, юзкейси, інтерфейсні адаптери та фреймворки. Сутності є високорівневими бізнес-моделями, які містять основні правила та логіку бізнесу і незалежні від інших частин системи. Юзкейси містять конкретні бізнес-логіку та правила, визначають, як система повинна поводитися в різних сценаріях, і можуть використовувати ентиті для виконання своєї роботи. Інтерфейсні адаптери відповідають за перетворення даних між різними частинами системи і можуть включати веб-контролери, в'ю моделі та інші компоненти, що забезпечують взаємодію з зовнішнім світом. Фреймворки та драйвери є зовнішніми інструментами та бібліотеками, які використовуються системою, але не впливають на внутрішню логіку. У наступному розділ буде розгялнуто, як саме було реалізовано ці принципи.

### 4.2.2 Структура проєкту

Проєкт складається з наступних частин:

• app: ця папка містить основну логіку програми. Вона може включати різні модулі та компоненти, що забезпечують функціональність додатка. Відповідає

за шар use case'ів (interactors), де знаходиться бізнес-логіка додатку. Це дозволяє зосередити всю основну логіку в одному місці.

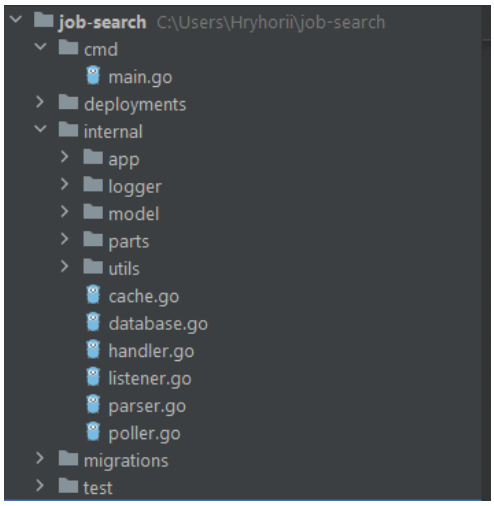



Рисунок 4.2 – загальна структура проєкту

• logger: папка містить реалізації логування для додатку. Логування є частиною інфраструктури, що забезпечує підтримку основної логіки додатка, але не змішується з нею.

• model: Папка містить моделі даних або об'єкти, які використовуються у програмі, наприклад, структури для бази даних. Відповідає шару entities, де знаходяться основні бізнес-об'єкти та логіка їх взаємодії.

• parts: папка містить різні частини або модулі, які складають програму. Можна трактувати як інші складові інфраструктури або допоміжні модулі, що взаємодіють з основною логікою.

• utils: папка містить утиліти або допоміжні функції, які можуть бути використані у різних місцях програми. Утиліти є частиною інфраструктури або шару framework, що надають підтримку для основних компонентів.

• Файли: cache.go, database.go, handler.go, listener.go, parser.go, poller.go:

Інтерфейси відіграють ключову роль у розділенні різних рівнів системи та забезпеченні гнучкості. Ці файли можуть відповідати різним рівням архітектури. Наприклад, handler.go може бути частиною шару інтерфейсу

(interface adapters), що забезпечує взаємодію з користувачами або іншими системами. Є частиною рівня interface adapters.

• migrations: папка для зберігання файлів міграцій бази даних, які використовуються для зміни структури бази даних з часом. Відповідає шару інфраструктури, що забезпечує підтримку і керування базою даних.

test: папка для зберігання тестових файлів і тестових сценаріїв для проекту. Відповідає шару тестування, що забезпечує перевірку коректності всіх шарів та компонентів системи.

## 4.3 Розробка системи парсингу для пошуку вакансій

### 4.3.1 Патерн Фабрика (Factory Pattern)

Патерн Фабрика є одним з основних породжувальних патернів проектування. Цей патерн використовується для створення об'єктів без конкретизації класу цього об'єкта. Фабричний метод визначає метод, який повинен використовуватися для створення об'єктів.

Він дозволяє легко розширювати функціональність додатку, додаючи нові класи без необхідності змінювати існуючий код. Цей підхід особливо корисний у випадках, коли створення об'єктів здійснюється залежно від умов, таких як типи підписок, як у випадку з підписковими парсерами.

Таким чином, це спрощує процес створення об'єктів та зменшує залежність коду від конкретних класів, що покращує його модульність і можливість розширення.

### 4.3.2 Реалізація

Система повинна бути здатна працювати з декількома джерелами вакансій, такими як DOU, Robota.ua, Djinni та тестові джерела. Для реалізації цієї системи було використано мову програмування Go та підхід з використанням фабрики для створення об'єктів парсерів.

Основна ідея системи полягає у використанні паттерну фабрики для створення парсерів для різних джерел вакансій. Це дозволяє легко додавати нові джерела в майбутньому, розширюючи можливості системи без значних змін в існуючом

У реалізації патерну Фабрика для створення парсерів, фабрика (Factory) має метод GetParser, який на основі типу підписки (subscription.Type) створює відповідний парсер. Наприклад, для типу підписки model.DOU, метод поверне об'єкт типу dou.Parser.у коді.

### 4.3.3 Приклад реалізації інтерфейсу Parser

Структура Parser представляє парсер для вакансій з DOU.ua. Вона має одне поле query, яке використовується для пошуку вакансій на сайті. Розглянемо основні компоненти:

- Конструктор New: це є фабричною функцією, яка створює новий екземпляр Parser з заданим запитом query.
- Функція parseHtml: використовується для рекурсивного парсингу HTML-документа з вакансіями. Вона приймає два аргументи: n, що є початковим вузлом для парсингу, і vacancies, що є контейнером для зберігання знайдених вакансій.
- Метод Parse: реалізує інтерфейс core.Parser. Він використовує функцію utils.GetHTML для отримання HTML-контенту сторінки з вакансіями за

певним запитом query. Після цього він парсить HTML-контент за допомогою функції parseHtml і повертає структуру model.Vacancies, яка містить зібрані вакансії.

- Метод parseHtml: перевіряє кожен елемент html.Node і, якщо він є тегом <a> з певними умовами (class="vt"), додає вакансію до списку vacancies. Вона також рекурсивно траверсує дочірні вузли для обробки всіх елементів документа.

```
//go:generate moq -fmt goimports -out ./parts/mocks/parser.go -pkg mock . Parser
type Parser interface {
    Parse(ctx context.Context) (*model.Vacancies, error)
}
```

Рисунок 4.3 – інтерфейс Parser

- Метод Parse: функція спочатку отримує HTML-контент сторінки з вакансіями за допомогою функції utils.GetHTML. Після цього він парсить отриманий HTML-контент за допомогою стандартного пакету golang.org/x/net/html, викликаючи функцію parseHtml для обробки кожного елемента документа. Він створює новий об'єкт model.Vacancies та заповнює його вакансіями, знайденими під час парсингу.

## 4.4 Обробка оновлень в Telegram Bot

### 4.4.1 Паттерн Chain of Responsibility

Паттерн "Ланцюжок обов'язків" є важливим інструментом в об'єктно-орієнтованому програмуванні. Він дозволяє створювати логічні ланцюжки обробників для обробки запитів або подій. Кожен обробник в цьому ланцюжку може спробувати обробити запит самостійно або передати його наступному

обробнику у ланцюжку. Це забезпечує гнучкість і розділення обов'язків, що робить паттерн особливо корисним у розробці програмного забезпечення.

Основна ідея полягає в тому, що є в наявності декілька обробників, які послідовно спробоватимуть обробити запит. Кожен обробник може вирішити, чи він може обробити запит самостійно, або передати його наступному обробнику у ланцюжку. Це дає можливість динамічно змінювати порядок обробки або додавати нові обробники без зміни клієнтського коду.

Основні переваги паттерну включають зменшення залежності між клієнтом і обробниками, спрощення коду завдяки розділенню обов'язків між обробниками, а також можливість конфігурування ланцюжка за допомогою додавання, видалення або зміни порядку обробників.

### 4.4.2 Реалізація

Структура Listener з служить об'єктом, який приймає оновлення від користувачів через Telegram Bot API. Вона використовується для обробки цих оновлень і передачі до відповідного обробника.

Розглянемо основні компоненти:

- Конструктор New: є фабричною функцією, яка створює новий екземпляр Listener з переданими залежностями: ботом Telegram, кешем та базою даних.
- Метод Listen: слідкує за оновленнями бота та передає їх відповідному обробнику. Він витягує ім'я користувача з кожного оновлення, завантажує користувача з кешу, створює контекст обробника, викликає обробник та обробляє помилки. Якщо обробник повертає помилку, метод відправляє повідомлення про помилку користувачеві та записує оновленого користувача до кешу.
- getHandler: Метод getHandler створює та повертає ланцюжок обов'язків обробників для різних типів оновлень бота.

Також Структура Listener має поля database, cache і botAPI, які використовуються для зберігання залежностей: бази даних, кешу та об'єкту бота Telegram.

### 4.4.3 Приклад реалізації інтерфейсу Handler

У нашому прикладі буде розглянуто клас List, який є частиною ланцюжка обробників запитів у Telegram-боті. Цей клас відповідає за обробку команди на виведення списку підписок користувача.

Клас List має наступні поля:

- cache: об'єкт кешу, який використовується для збереження даних у кеші.
- database: об'єкт бази даних, через який відбувається доступ до даних про підписки користувачів.
- handler: наступний обробник у ланцюжку, який буде виконувати обробку, якщо поточний обробник не зміг обробити запит.

```
type Handler interface {
    Execute(ctx context.Context, handlerContext *HandlerContext) error
    SetNext(handler Handler)
}
```

Рисунок 4.4 – інтерфейс Handler

- Метод SetNext встановлює наступний обробник у ланцюжку.
- Метод Execute виконує обробку запиту. Він перевіряє, чи статус користувача є LIST_SUBSCRIPTIONS і виводить список підписок з бази даних. Якщо статус не співпадає, запит передається наступному обробнику у ланцюжку. Якщо в ланцюжку немає наступного обробника, генерується помилка.

## 4.5 Надсилання сповіщень в Telegram Bot

### 4.5.1 Паттерн Observer(Спостерігач)

Паттерн Observer є одним із популярних поведінкових паттернів проектування. Він використовується для реалізації механізму підписки, де об'єкт (спостерігач або підписник) надсилається на зміни у іншому об'єкті (спостережуваний об'єкт або суб'єкт). Цей паттерн дозволяє забезпечити слабку залежність між спостерігачами і суб'єктом, що забезпечує гнучкість і розширюваність системи.

### 4.5.2 Реалізація

Клас Listener використовується для слухання подій і взаємодії з користувачами через Telegram-бота. Він відповідає за інтеграцію Telegram-бота з базою даних та кешем, а також взаємодіє з підсистемою підписки (Poller).

Клас Listener має наступні поля:

- Конструктор NewListener: створює новий екземпляр класу Listener, ініціалізує його залежності (Telegram-бот, кеш, база даних) і створює новий Poller.
- Метод Listen: Під час виклику методу Listen, викликається метод launch, який завантажує підписки з бази даних і передає їх у Poller, а таклж запускаються асинхронні горутини для додавання та видалення підписок через Poller.
- Метод launch: отримує всі підписки з бази даних. Додає кожну підписку у Poller для моніторингу оновлень вакансій.
- Метод addIncomingSubscriptions: асинхронно отримує нові підписки з кеша. Додає кожну нову підписку у Poller для моніторингу.

- Метод removeIncomingSubscriptions: асинхронно отримує підписки для видалення з кеша. Видаляє кожну з них з Poller.

### 4.5.3 Приклад реалізації інтерфейсу Poller

```
type Poller interface {
    AddSubscription(ctx context.Context, subscription model.Subscription)
    RemoveSubscription(ctx context.Context, subscription model.Subscription)
    StartPolling(ctx context.Context)
}
```

Рисунок 4.5 – інтерфейс Poller

У нашому прикладі буде розглянуто клас Poller, який відповідає за асинхронне отримання та обробку оновлень в Redis cache. Цей клас є важливою частиною архітектури Telegram-бота, яка дозволяє ефективно отримувати та обробляти оновлення в реальному часі.

Клас Poller має наступні поля:

- bot: об'єкт Telegram-бота, через який відбувається взаємодія з Telegram API.
- parserFactory: фабрика парсерів, яка створює парсери для обробки конкретних типів підписок.
- checkers: мапа, яка зберігає об'єкти Checker, які відповідають за перевірку оновлень для кожної підписки.
- mu: м'ютекс для безпечного доступу до мапи checkers з різних горутин.

Клас має механізм додавання та видалення підписок, а також основний метод StartPolling, який запускає цикл перевірки оновлень для кожної зареєстрованої підписки. Кожен об'єкт Checker відповідає за перевірку оновлень для конкретної підписки та викликає метод CheckVacancies, коли настав час оновлення підписки.

# ВИСНОВОК ДО РОЗДІЛУ 4

Процес розробки додатку розпочався з організації розробки, де було визначено використання системи контролю версій, зокрема Git, та платформи Github для зберігання і співпраці над кодом. Впровадження ефективної методології розробки, такої як Agile, сприяло гнучкому та ітеративному підходу до розробки програмного забезпечення.

Наступним важливим кроком було визначення архітектури додатку. Використання Clean Architecture забезпечило модульність, гнучкість та тестованість коду. Детально розглянуто структуру проєкту, що дозволило забезпечити чіткий розподіл відповідальностей між компонентами системи.

Особлива увага була приділена розробці системи парсингу для пошуку вакансій. Використання патерну "Фабрика" (Factory Pattern) дозволило створити гнучку та розширювану систему для парсингу даних. Було детально описано реалізацію системи, включаючи приклад реалізації інтерфейсу Parser, що демонструє практичне застосування патерну[22].

Для обробки оновлень в Telegram Bot було використано патерн "Ланцюжок відповідальностей" (Chain of Responsibility). Це забезпечило можливість гнучкого та динамічного оброблення різних типів повідомлень, що надходять до бота. Детально розглянуто реалізацію цього патерну, а також наведено приклад реалізації інтерфейсу Handler.

Надсилання сповіщень в Telegram Bot було реалізовано за допомогою патерну "Спостерігач" (Observer). Це дозволило створити ефективну систему для надсилання сповіщень користувачам про нові вакансії. Було детально описано процес реалізації цього патерну, включаючи приклад реалізації інтерфейсу Poller.

# РОЗДІЛ 5. ТЕСТУВАННЯ

## 5.1 Мануальне тестування

Телеграм-бот можна знайти за посиланням https://t.me/Job_Search_Notifier_bot. Можемо побачити початкову сторінку на рис. 5.1.

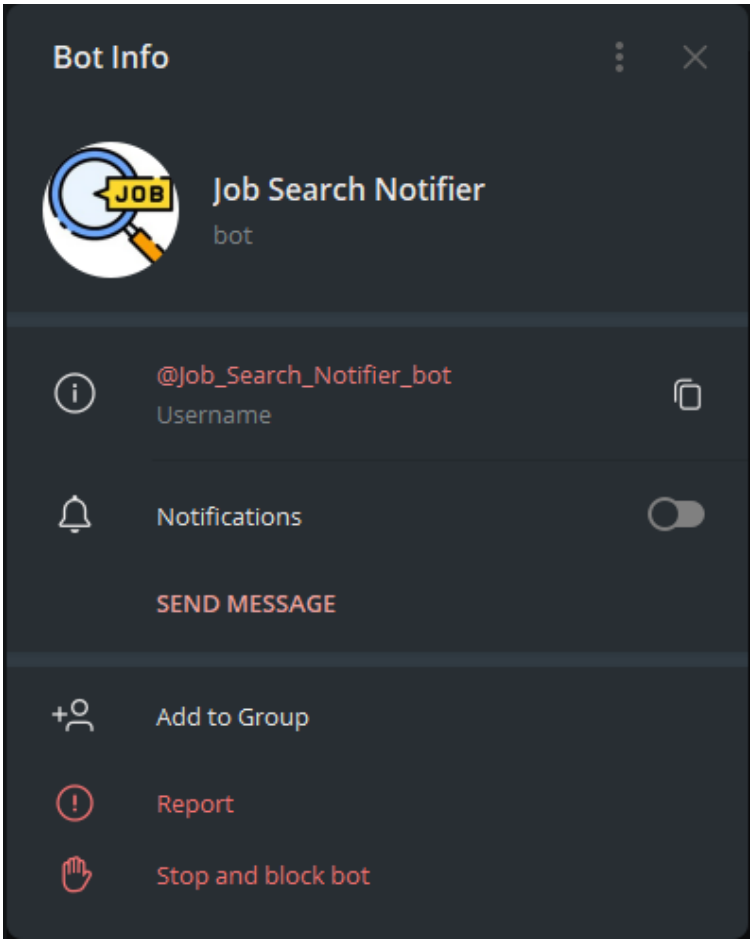



Рисунок 5.1 – початкове меню

Далі натискаємо start, бачимо привітання з використанням юзернейму в телеграм на рис 5.2:

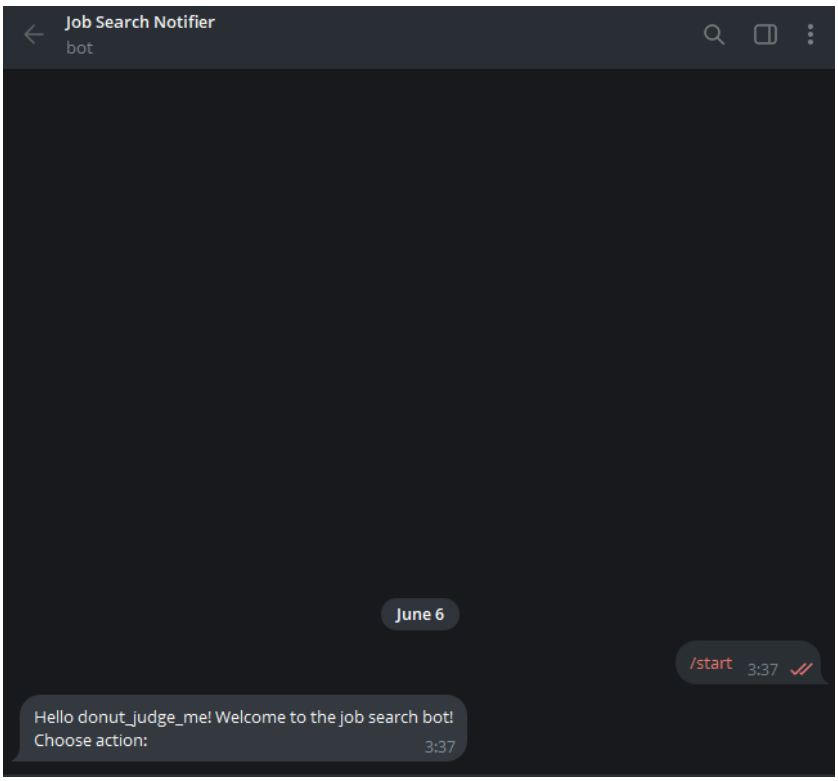



Рисунок 5.2 – початкове привітання

Далі бачимо декілька кнопок для роботи з функціоналом: List, Add та Delete(рис. 5.3).

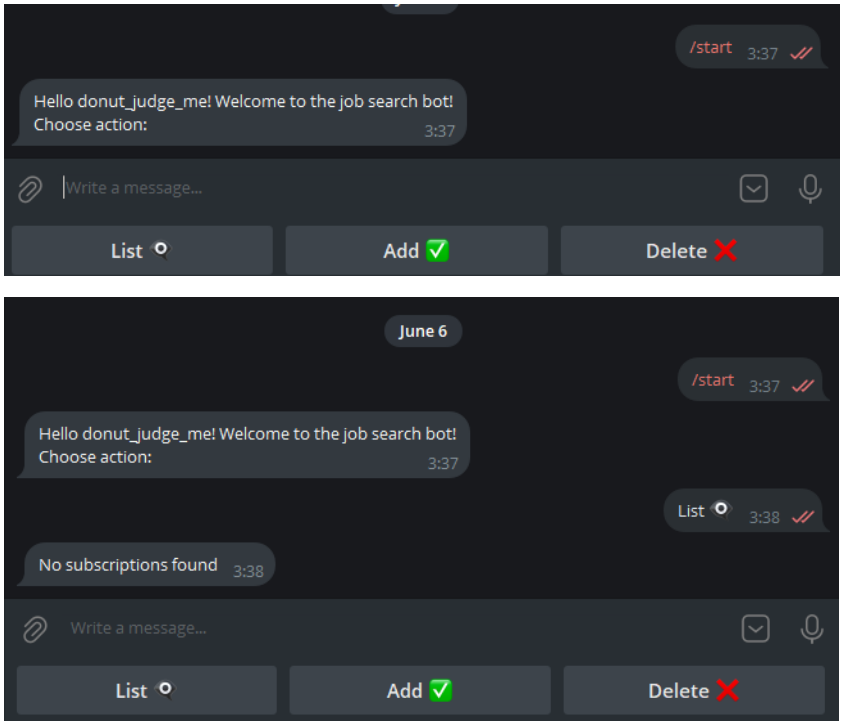



Рисунок 5.3 – початкові кнопки

Натискаємо клавішу Add, та бачимо список доступних підписок: наразі є підтримка dou, robota.ua та djinni.(рис 5.4).

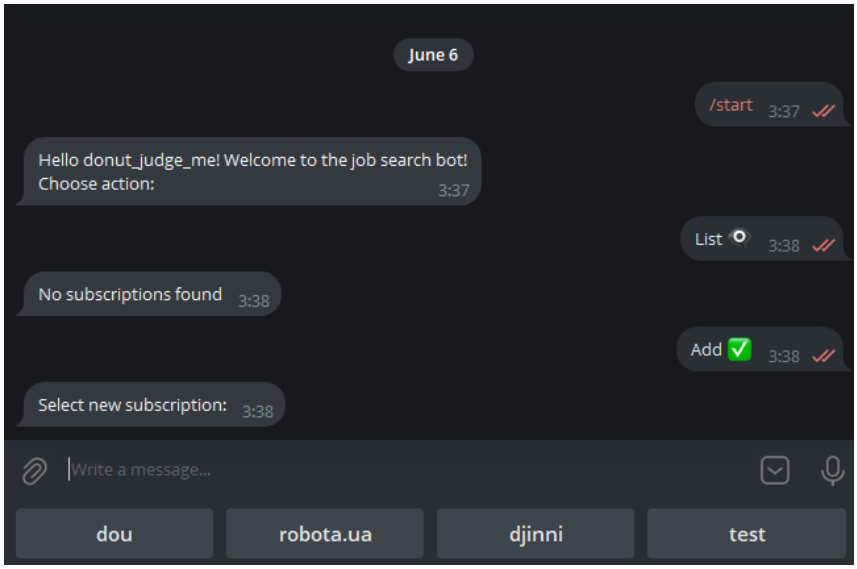



Рисунок 5.4 – додати підписку

Далі вибираємо підписку, та вводимо моніторинговий запит(наприклад senior golang).(рис 5.5).

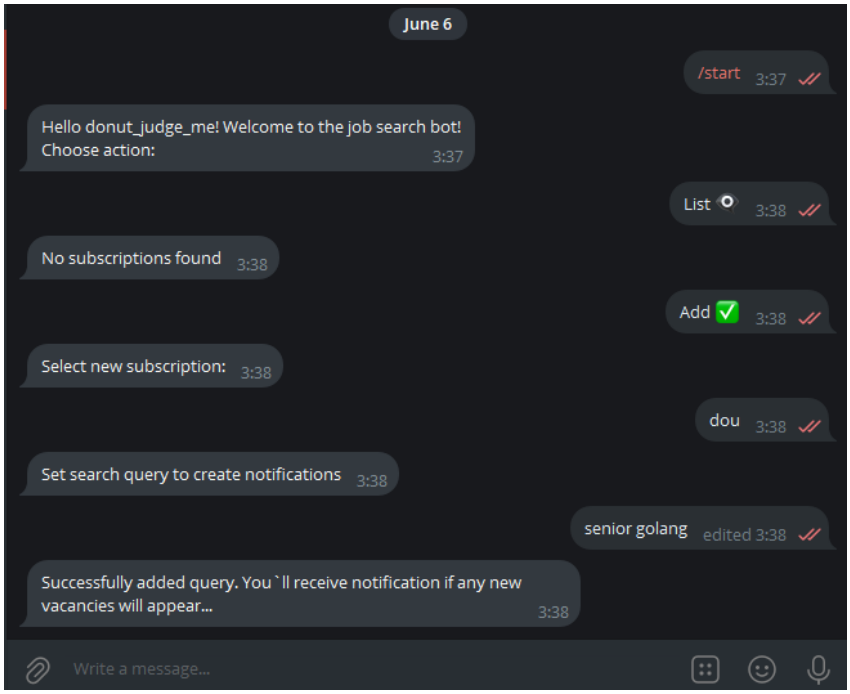



Рисунок 5.5 – додати підписку dou

Далі натискаємо кнопку List, та бачимо, що було додана нова підписка(рис. 5.6).

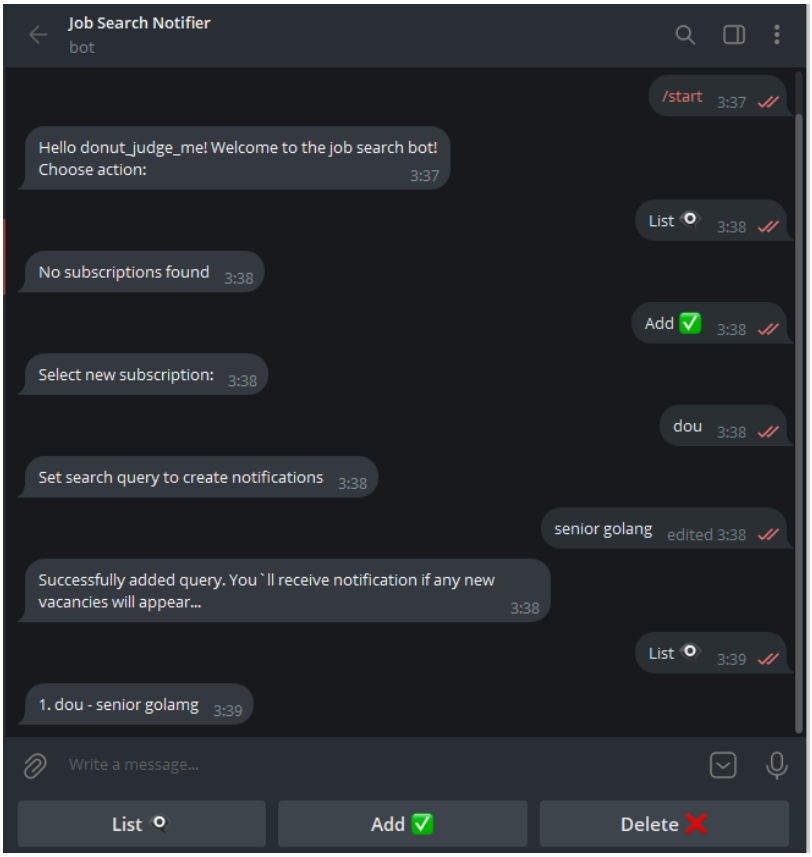



Рисунок 5.6 – показати підписки

Далі додаємо декілька підписок: одну для dou, другу для robota.ua. Вони були успішно додані. Тепер натискаємо кнопку Delete. Ми отримаємо інтерактивне меню, в якому можемо видалити будь-яку підписку з нещодавно доданих(рис. 5.7).

Після натискання кнопки dou – senior golang – можемо побачити, мо підписка успішно видаляється(рис. 5.8, 5.9).

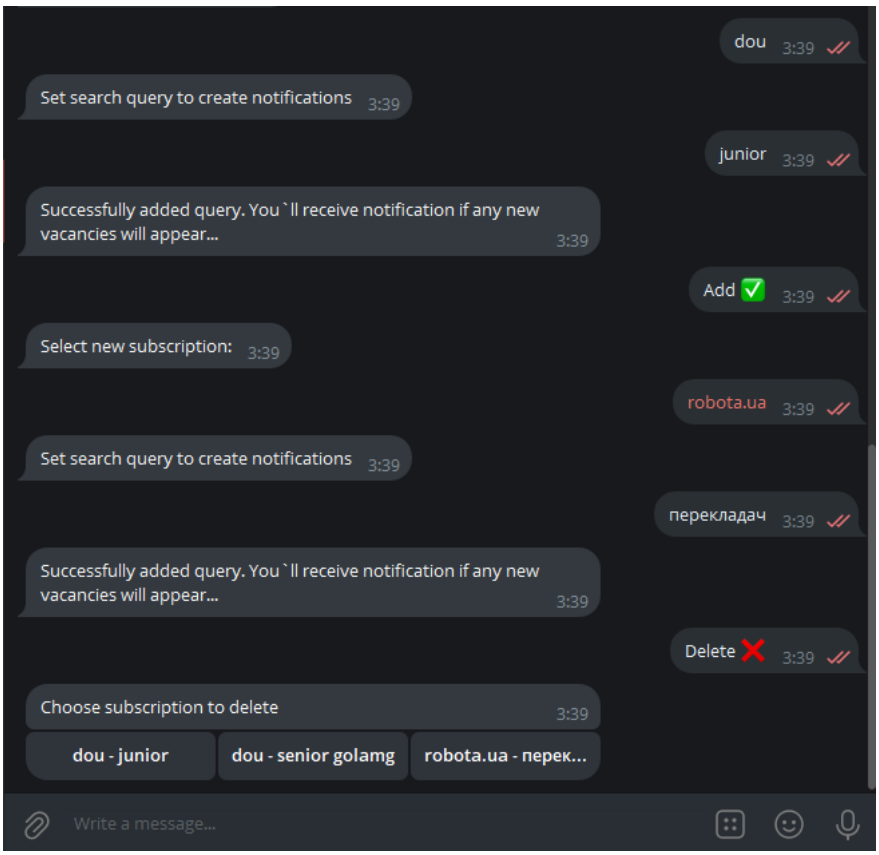



Рисунок 5.7 – видалити підписку

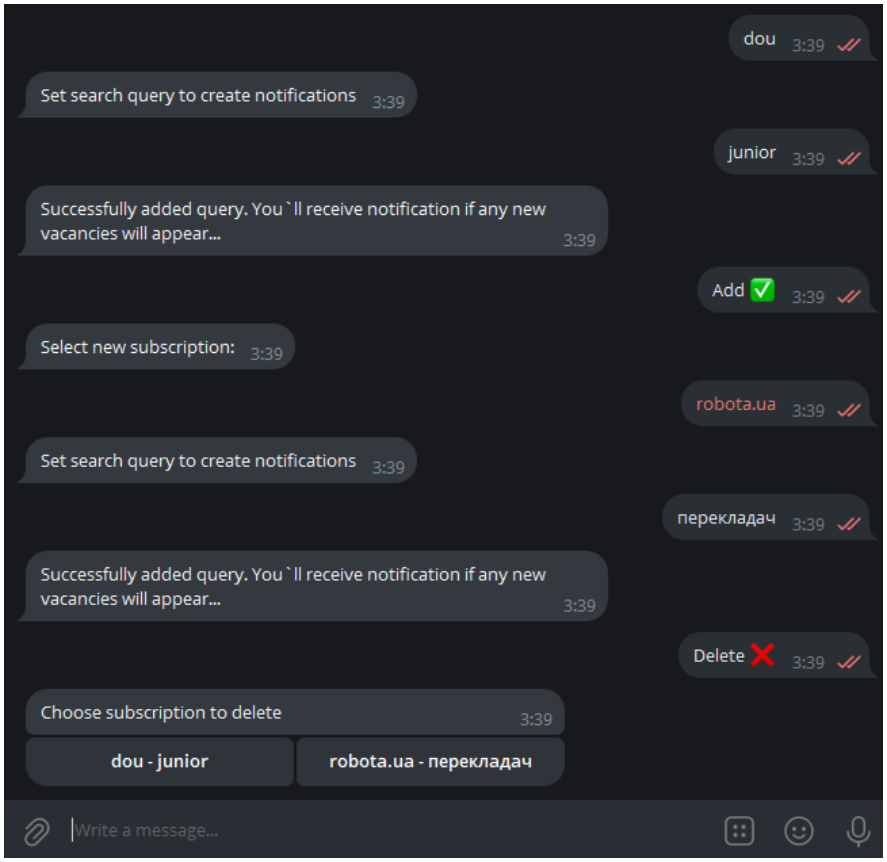



Рисунок 5.8 – результат видалення підписки

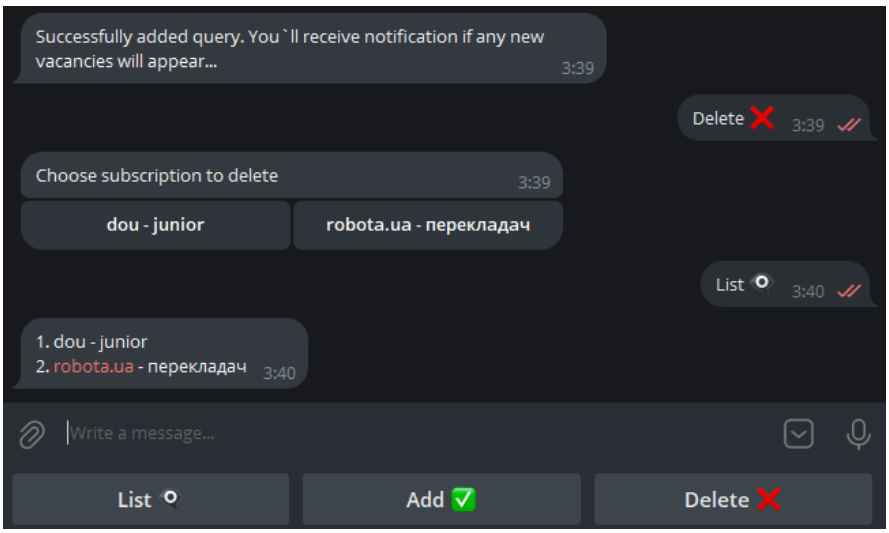



Рисунок 5.9 – верифікація факту видалення підписки

Після налаштування підписок, як тільки на сайті з вакансіями з'явиться нова вакансія, в наш телеграм бот прийде посилання з назвою вакансії, іменем підписки, та посиланням на нову вакансію(рис 5.10).


Рисунок 5.10 – приклад надісланої підписки

## 5.2 Автоматичне тестування

### 5.2.1 Тестовий вебсервер

Для автоматичного тестування було написано невеличкий сервер, який імітував реальний сайт з вакансіями, а також парсер, підлаштований конкретно під цей сервер.

Основний пакет програми конфігурує простий HTTP-сервер, який відповідає на запити даними у форматі JSON. Він ініціалізує попередньо визначені дані, які використовуються для відповідей на HTTP-запити. Сервер періодично оновлює ці дані, додаючи нові випадково згенеровані записи кожні 30 секунд, щоб імітувати динамічне оновлення інформації. Після останнього оновлення програма залишає сервер активним, не вносячи нових змін протягом тривалого періоду часу.

Оскільки тести запускалися в локальному середовищі, для того, щоб кластер міг отримати доступ, було свтворено віртуальний сервер TP-LINK(рис 5.11). Завдяки цьому сервер був публічно доступний через IP-address.

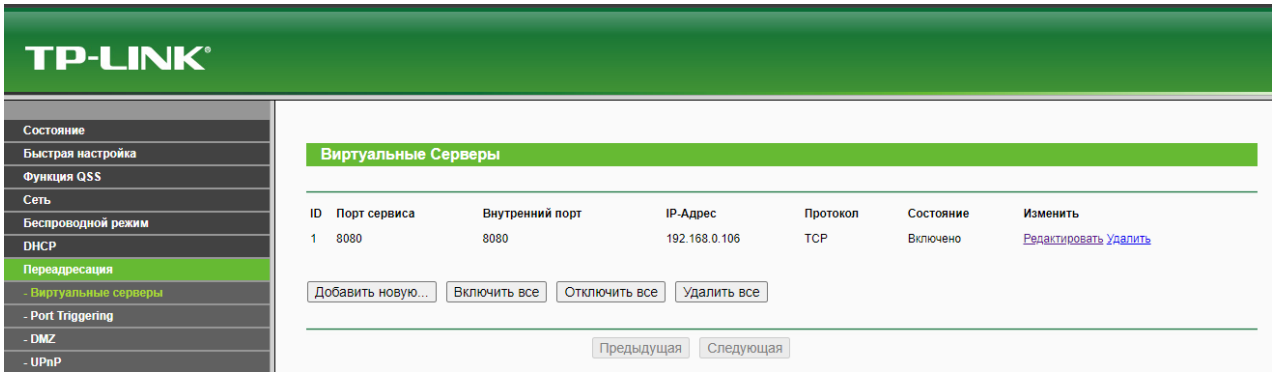


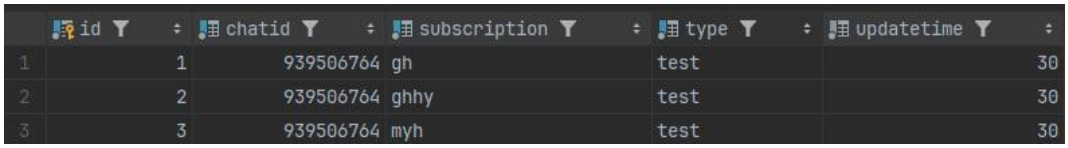

Рисунок 5.11 – створений віртуальний сервер

### 5.2.2 Імітація великої кількості користувачів

Оскільки тести запускалися в локальному середовищі, для того, щоб кластер міг отримати доступ, було свтворено віртуальний сервер TP-LINK(рис 5.11). Завдяки цьому сервер був публічно доступний через мій IP-address. У цьому розділі було досліджено вплив великого навантаження користувачів на веб-сервер за допомогою симуляційних тестів. Для адекватного відтворення реальних умов було використано різні кількості користувачів, що одночасно взаємодіють з сервером: 3, 10 і 20.

Наш веб-сервер налаштований для обробки запитів на адресу /, яка повертає дані у форматі JSON. Початкові дані включають список з трьох записів інформації про вакансії.

- Тестування з 3 користувачами



| | id | chatid | subscription | type | updatetime |
|---|---|---|---|---|---|
| 1 | 1 | 939506764 | gh | test | 30 |
| 2 | 2 | 939506764 | ghhy | test | 30 |
| 3 | 3 | 939506764 | myh | test | 30 |

Рисунок 5.12 – скріншот бази даних, 3 користувачів

Після запуску сервера три користувачі одночасно надсилають запити до сервера, які викликають оновлення його даних кожні 30 секунд. Кожні 30 секунд сервер додаватиме деяку випадково згенеровану вакансію до списку, що повертається відповіддю.

- Тестування з 10 користувачами

| | id | chatid | subscription | type | updatetime |
|---|---|---|---|---|---|
| 1 | 1 | 939506764 | gh | test | 30 |
| 2 | 2 | 939506764 | ghhy | test | 30 |
| 3 | 3 | 939506764 | myh | test | 30 |
| 4 | 4 | 939506764 | t | test | 30 |
| 5 | 5 | 939506764 | tt | test | 30 |
| 6 | 6 | 939506764 | htj | test | 30 |
| 7 | 7 | 939506764 | hgjt | test | 30 |
| 8 | 8 | 939506764 | tyj | test | 30 |
| 9 | 9 | 939506764 | jhtjy | test | 30 |
| 10 | 10 | 939506764 | tyjuj | test | 30 |

Рисунок 5.13 – скріншот бази даних, 10 користувачів

На наступному етапі, 10 користувачів одночасно надсилають запити до сервера. Це створює більше навантаження на сервер, оскільки він тепер обробляє більшу кількість запитів від користувачів, що спостерігають за вакансіями.

- Тестування з 20 користувачами

| | id | chatid | subscription | type | updatetime |
|---|---|---|---|---|---|
| 1 | 1 | 939506764 | gh | test | 30 |
| 2 | 2 | 939506764 | ghhy | test | 30 |
| 3 | 3 | 939506764 | myh | test | 30 |
| 4 | 4 | 939506764 | t | test | 30 |
| 5 | 5 | 939506764 | tt | test | 30 |
| 6 | 6 | 939506764 | htj | test | 30 |
| 7 | 7 | 939506764 | hgjt | test | 30 |
| 8 | 8 | 939506764 | tyj | test | 30 |
| 9 | 9 | 939506764 | jhtjy | test | 30 |
| 10 | 10 | 939506764 | tyjuj | test | 30 |
| 11 | 11 | 939506764 | fjydfdhgj | test | 34 |
| 12 | 12 | 939506764 | dgfhtgjk | test | 453 |
| 13 | 13 | 939506764 | sdfghjk | test | 546 |
| 14 | 14 | 939506764 | wqersdftgyh | test | 546 |
| 15 | 15 | 939506764 | dsfghj | test | 56 |
| 16 | 16 | 939506764 | dsertfyg | test | 46 |
| 17 | 17 | 939506764 | dsafegrt | test | 4 |
| 18 | 76 | 939506764 | dfsegrt | test | 36 |
| 19 | 63 | 939506764 | wfertyu | test | 4765 |
| 20 | 346 | 939506764 | fedrgtyu | test | 4 |
| 21 | 3646 | 939506764 | fwergthy | test | 456 |
| 22 | 456 | 939506764 | fergtyu | test | 45 |

Рисунок 5.14 – скріншот бази даних, 20 користувачів

На останньому етапі, 20 користувачів одночасно надсилають запити до сервера. Це створює ще більше навантаження на сервер, перевіряючи його реакцію на значну кількість одночасних запитів.

Після проведення тестів можемо підтвердити, що сервер успішно справлявся з усіма навантаженнями. Використання бази даних Redis та додатку в нормі, а також контроль ресурсів (ЦПУ та пам'яті) в нормі. Час відповіді сервера залишався прийнятним, навіть під великим навантаженням.

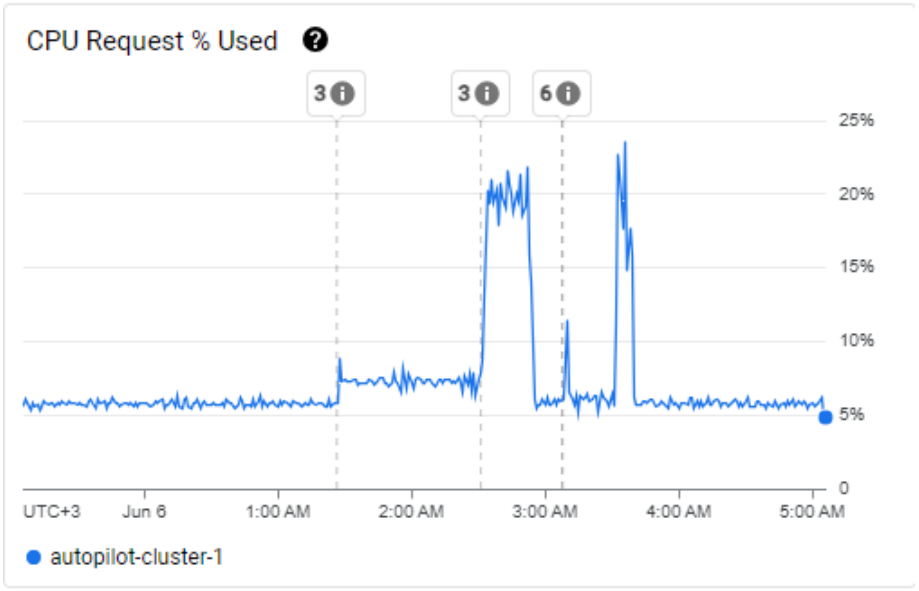



Рисунок 5.15 – показники CPU

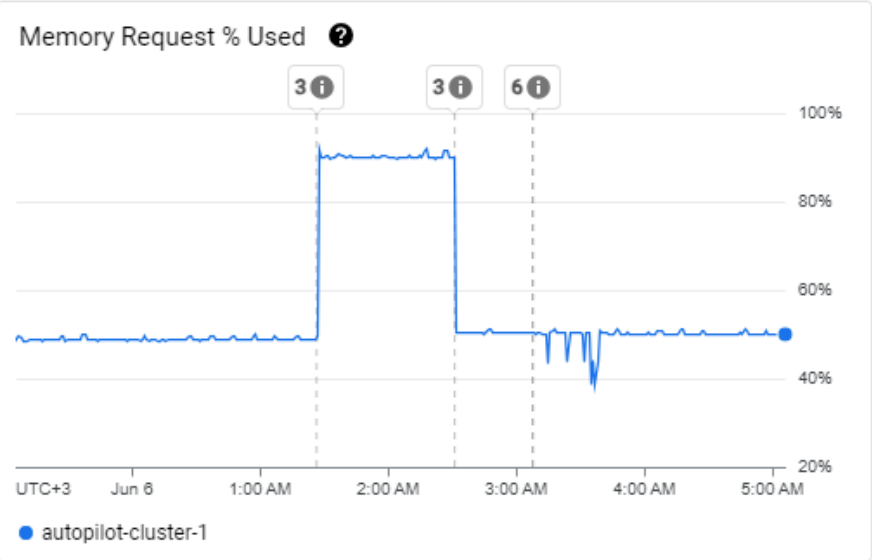



Рисунок 5.16 – показники Memory

Цей експеримент підтверджує, що наш веб-сервер здатний ефективно обробляти велику кількість одночасних запитів, а системні ресурси і база даних використовуються ефективно і з обережністю.

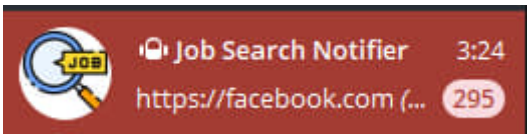



Рисунок 5.17 – результат надісланих сповіщень

Результати тестування свідчать про успішність реалізації веб-сервера з можливістю динамічного оновлення даних під великим навантаженням користувачів. Всі параметри, включаючи швидкість відповіді, використання ресурсів і стійкість до високого навантаження, відповідали очікуванням і відповідали стандартам ефективності та надійності.

# ВИСНОВОК ДО РОЗДІЛУ 5

Дослідження реалізації телеграм-бота для моніторингу вакансій підтвердило його успішну функціональність та надійність під час взаємодії з користувачем. Бот здатен автоматично надсилати сповіщення про нові вакансії у відповідь на підписки, створені користувачами. Основні функції бота включають можливість додавання, перегляду та видалення підписок на конкретні пошукові запити через інтерактивне меню.

Основна функціональність бота:

- Додавання підписок на вакансії з різних сайтів з працевлаштування.
- Перегляд підписаних запитів.
- Видалення підписок за допомогою інтерактивного меню.

Для перевірки функціональності та надійності бота було проведено мануальне та автоматичне тестування. Мануальне тестування показало, що усі основні функції бота працюють як очікувалося. Автоматичне тестування включало імітацію великої кількості користувачів, що одночасно взаємодіюють з ботом, і підтвердило його здатність ефективно обробляти велику кількість запитів.

Результати тестування:

- Усі функції бота були успішно протестовані.
- Веб-сервер бота демонструє стабільну реакцію на збільшене навантаження.
- Використання бази даних Redis та системних ресурсів (ЦПУ та пам'яті) перебуває в межах норми під час роботи з ботом.

Це дослідження підтверджує, що реалізація телеграм-бота для моніторингу вакансій є ефективною та надійною. Він здатний надавати користувачам актуальну інформацію про нові вакансії та взаємодіяти з ними у зручному форматі через платформу Telegram.

# СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ


1. Контейнеризація додатків [Електронний ресурс] – Режим доступу до ресурсу: https://docker.com/what-container.
2. Огляд Kubernetes [Електронний ресурс] // Kubernetes Documentation – Режим доступу до ресурсу: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/.
3. Jones M. D. Docker Deep Dive / M. D. Jones. – California: Independently Published, 2020. – С. 110-145.
4. Smith J. P. Kubernetes Up and Running / J. P. Smith, B. Burns, J. Beda. – Sebastopol: O'Reilly Media, 2017. – С. 25-50.
5. Martin R. Clean Architecture: A Craftsman's Guide to Software Structure and Design / R. Martin. – New York: Prentice Hall, 2017.
6. Особливості та переваги Redis [Електронний ресурс] // Redis Documentation – Режим доступу до ресурсу: https://redis.io/docs/about/.
7. PostgreSQL: The World's Most Advanced Open Source Relational Database [Електронний ресурс] – Режим доступу до ресурсу: https://www.postgresql.org/about/.
8. Віртуалізація та її недоліки [Електронний ресурс] // VMware – Режим доступу до ресурсу: https://vmware.com/virtualization/virtualization-and-its-benefits.html.
9. Налаштування Dockerhub [Електронний ресурс] – Режим доступу до ресурсу: https://hub.docker.com/.
10. Mitchell J. GitHub Essentials / J. Mitchell. – Birmingham: Packt Publishing, 2015. – С. 15-40.
11. Технологія контейнеризації [Електронний ресурс] // Red Hat – Режим доступу до ресурсу: https://redhat.com/en/topics/containers/what-is-containerization.

12. Kubernetes Monitoring [Електронний ресурс] – Режим доступу до ресурсу: https://prometheus.io/docs/introduction/overview/.
13. Git: Система контролю версій [Електронний ресурс] // Git Documentation – Режим доступу до ресурсу: https://git-scm.com/doc.
14. Sivakumar S. The Kubernetes Book / S. Sivakumar. – Independently Published, 2019.
15. Helm: Kubernetes Package Manager [Електронний ресурс] – Режим доступу до ресурсу: https://helm.sh/docs/.
16. Проєктування баз даних [Електронний ресурс] // DB Design – Режим доступу до ресурсу: https://dbdesign.com/database-design/.
17. Особливості SQLC [Електронний ресурс] – Режим доступу до ресурсу: https://sqlc.dev/.
18. Налаштування інфраструктури в GKE [Електронний ресурс] – Режим доступу до ресурсу: https://cloud.google.com/kubernetes-engine/docs/.
19. Методологія розробки додатків [Електронний ресурс] // Agile Alliance – Режим доступу до ресурсу: https://agilealliance.org/.
20. Автоматичне тестування додатків [Електронний ресурс] – Режим доступу до ресурсу: https://testautomation.com/.
21. Приклад реалізації Factory Pattern [Електронний ресурс] – Режим доступу до ресурсу: https://refactoring.guru/design-patterns/factory-method.
22. Інтеграція Redis з додатками [Електронний ресурс] – Режим доступу до ресурсу: https://redis.com/integration/.
23. Контейнеризація Go додатків [Електронний ресурс] – Режим доступу до ресурсу: https://golang-docker.com/.
24. Google Kubernetes Engine: Best Practices [Електронний ресурс] – Режим доступу до ресурсу: https://cloud.google.com/kubernetes-engine/docs/best-practices.
25. Налаштування PostgresSQL в Kubernetes [Електронний ресурс] – Режим доступу до ресурсу: https://postgresql-kubernetes.com/.

26. Переваги використання Helm [Електронний ресурс] – Режим доступу до ресурсу: https://helm.sh/docs/intro/using_helm/.
27. Системи парсингу вакансій [Електронний ресурс] // Job Parsing Systems – Режим доступу до ресурсу: https://jobparsing.com/.
28. Моніторинг інфраструктури [Електронний ресурс] – Режим доступу до ресурсу: https://monitoring.com/.
29. Мікросервіси з Docker і Kubernetes [Електронний ресурс] – Режим доступу до ресурсу: https://microservices.io/patterns/microservices.html.
30. DevOps Практики [Електронний ресурс] // DevOps Agile Skills Association – Режим доступу до ресурсу: https://dasa.io/.
31. Docker: Налаштування CI/CD [Електронний ресурс] – Режим доступу до ресурсу: https://docs.docker.com/ci-cd/.

# ДОДАТОК 1

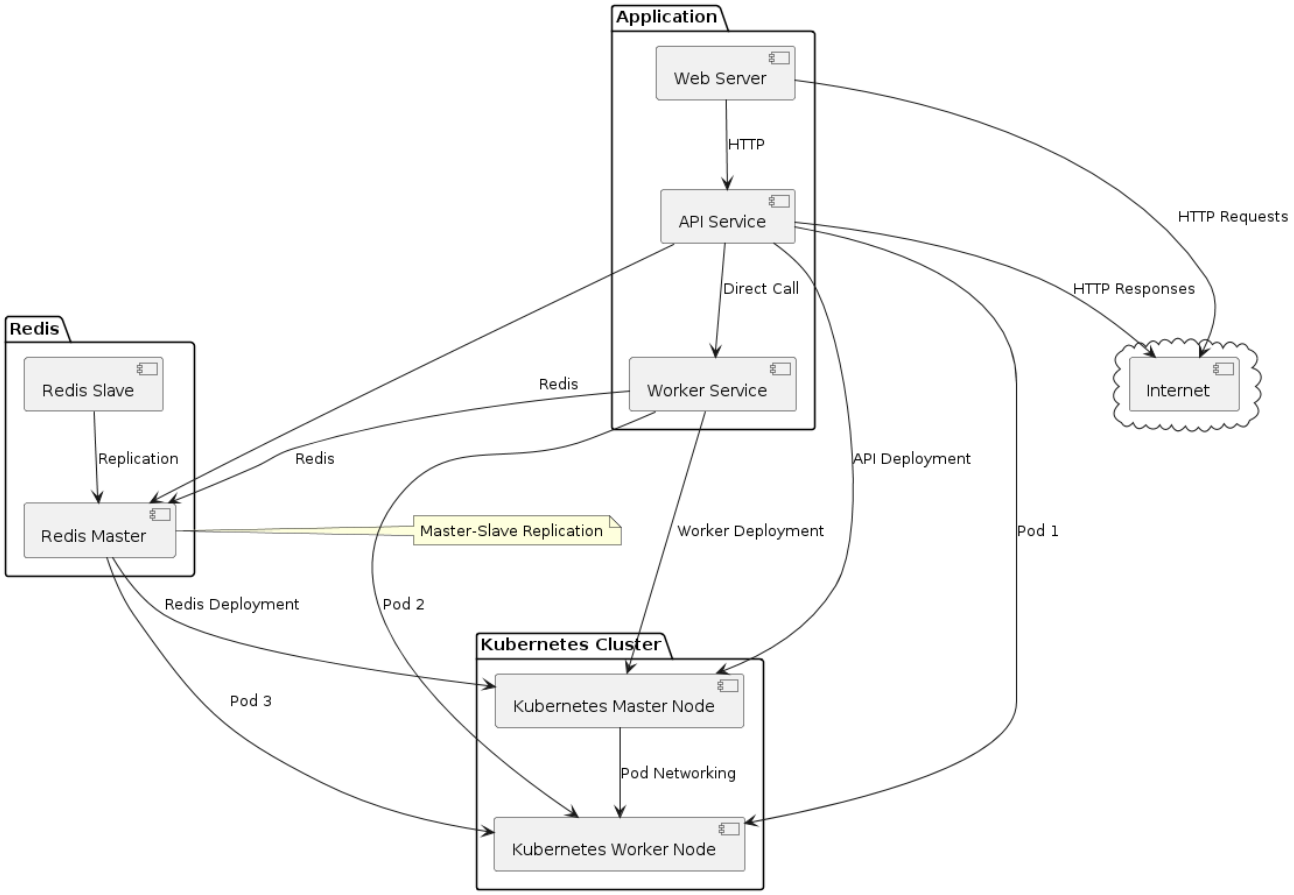
Телеграм-бот для пошуку роботи

## Структурна схема системи

ІАЛЦ.467200.002 Д1

Аркушів 1

**Київ 2024 р**




| | | | | | |
|---|---|---|---|---|---|
| | | | | | ІАЛЦ.467200.004 Д1 |
| | | № докум. | Підпис | Дата | |
| Розробив | | Грициняк Г.С. | | | *Телеграм-бот для пошуку роботи* Структурна схема системи |
| Перевірив | | Нечай Д.О. | | | |
| | | | | | |
| Н. Контр. | | Ковальчук О. М | | | |
| Затвердив | | | | | |

| Літ. | Аркуш | Аркушів |
|---|---|---|
| | 1 | 1 |

КПІ ім. Ігоря Сікорського, ФІОТ, ІО-06

# ДОДАТОК 2

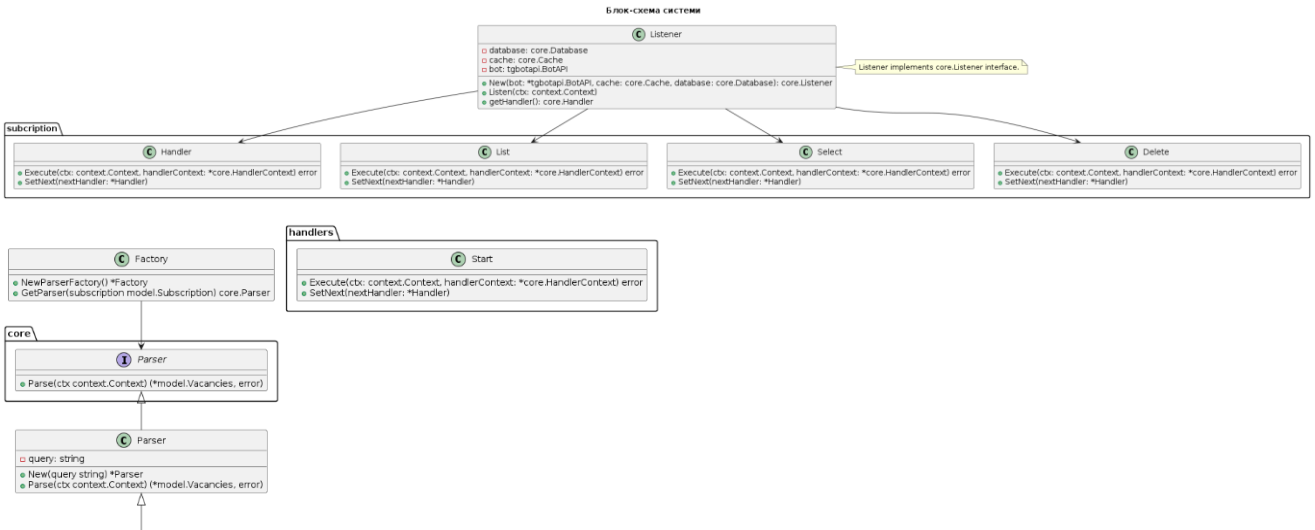
Телеграм-бот для пошуку роботи

**Функціональна схема (діаграма класів)**

ІАЛЦ.467200.002 Д2

Аркушів 1

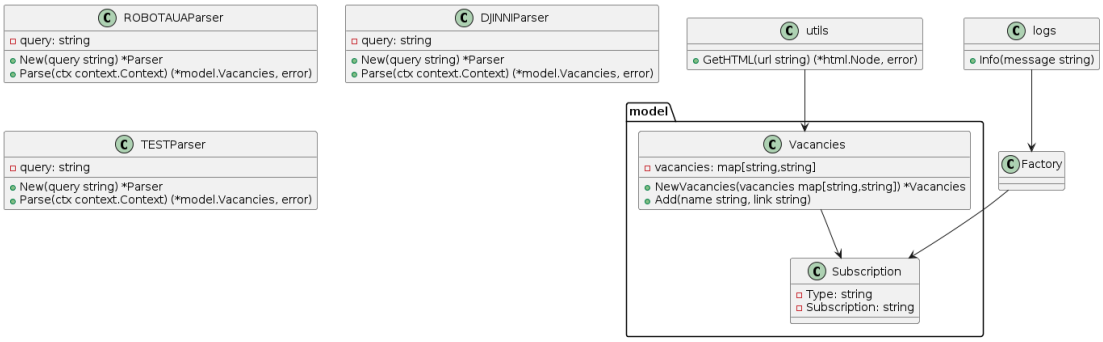
**Київ 2024 р**

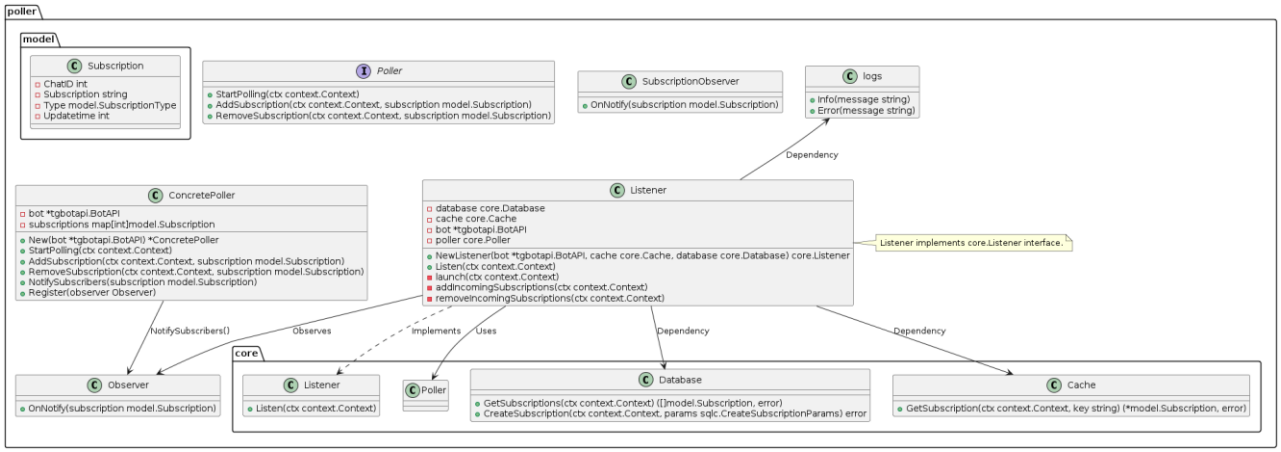



| | | | | |
|---|---|---|---|---|
| | | № докум. | Підпис | Дата |
| | | | | |

ІАЛЦ.467200.005 Д2

| | | | | | |
|---|---|---|---|---|---|
| Розробив | Грициняк Г.С. | | | *Телеграм-бот для пошуку роботи* Функціональна схема | Літ. / Аркуш / Аркушів: 1 / 1 |
| Перевірив | Нечай Д.О. | | | | |
| | | | | | КПІ ім. Ігоря Сікорського, ФІОТ, ІО-06 |
| Н. Контр. | Ковальчук О. М | | | | |
| Затвердив | | | | | |


ROBOTAUAParser
query: string
New(query string) *Parser
Parse(ctx context.Context) (*model.Vacancies, error)
DJINNIParser
query: string
New(query string) *Parser
Parse(ctx context.Context) (*model.Vacancies, error)
utils
GetHTML(url string) (*html.Node, error)
logs
Info(message string)
TESTParser
query: string
New(query string) *Parser
Parse(ctx context.Context) (*model.Vacancies, error)
model
Vacancies
vacancies: map[string,string]
NewVacancies(vacancies map[string,string]) *Vacancies
Add(name string, link string)
Factory
Subscription
Type: string
Subscription: string


poller
model
Subscription
ChatID int
Subscription string
Type model.SubscriptionType
Updatetime int
Poller
StartPolling(ctx context.Context)
AddSubscription(ctx context.Context, subscription model.Subscription)
RemoveSubscription(ctx context.Context, subscription model.Subscription)
SubscriptionObserver
OnNotify(subscription model.Subscription)
logs
Info(message string)
Error(message string)
Dependency
ConcretePoller
bot *tgbotapi.BotAPI
subscriptions map[int]model.Subscription
New(bot *tgbotapi.BotAPI) *ConcretePoller
StartPolling(ctx context.Context)
AddSubscription(ctx context.Context, subscription model.Subscription)
RemoveSubscription(ctx context.Context, subscription model.Subscription)
NotifySubscribers(subscription model.Subscription)
Register(observer Observer)
Listener
database core.Database
cache core.Cache
bot *tgbotapi.BotAPI
poller core.Poller
NewListener(bot *tgbotapi.BotAPI, cache core.Cache, database core.Database) core.Listener
Listen(ctx context.Context)
launch(ctx context.Context)
addIncomingSubscriptions(ctx context.Context)
removeIncomingSubscriptions(ctx context.Context)
Listener implements core.Listener interface.
NotifySubscribers()
Observes
Implements
Uses
Dependency
Dependency
core
Observer
OnNotify(subscription model.Subscription)
Listener
Listen(ctx context.Context)
Poller
Database
GetSubscriptions(ctx context.Context) ([]model.Subscription, error)
CreateSubscription(ctx context.Context, params sqlc.CreateSubscriptionParams) error
Cache
GetSubscription(ctx context.Context, key string) (*model.Subscription, error)

# ДОДАТОК 3

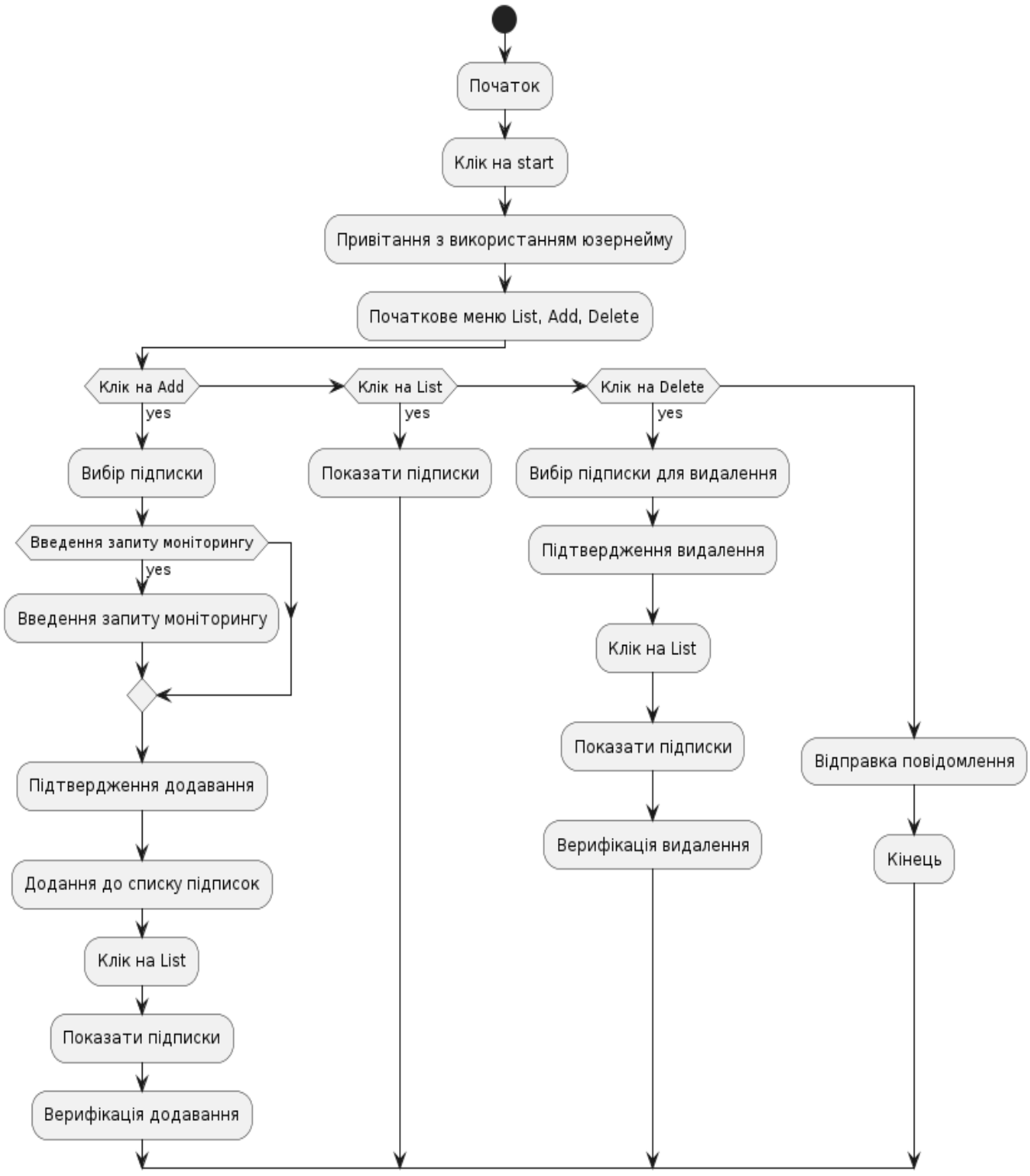
Телеграм-бот для пошуку роботи

## Алгоритм дій програмного забезпечення

ІАЛЦ.467200.002 Д3

Аркушів 1

**Київ 2024 р**




| | | | | | | |
|---|---|---|---|---|---|---|
| | | | | | ІАЛЦ.467200.006 Д3 | |
| | | № докум. | Підпис | Дата | | |
| Розробив | | Гриciняк Г.С. | | | *Телеграм-бот для пошуку роботи* Алгоритм дій програмного забезпечення | Літ. / Аркуш 1 / Аркушів 1 |
| Перевірив | | Нечай Д.О. | | | | |
| Н. Контр. | | Ковальчук О. М | | | | КПІ ім. Ігоря Сікорського, ФІОТ, ІО-06 |
| Затвердив | | | | | | |

# ДОДАТОК 4

Телеграм-бот для пошуку роботи

Текст програмного коду

ІАЛЦ.467200.002 Д4

Аркушів 41

**Київ 2024 р**

```
package main

import (
	"context"
	"github.com/greg-source/job-search/internal/app"
	logs "github.com/greg-source/job-search/internal/logger"
	"github.com/greg-source/job-search/internal/utils"
	"go.uber.org/zap"
	"os"
	"os/signal"
)

func main() {
	config, err := utils.LoadConfig()
	if err != nil {
		logs.Fatal("failed to load configuration", zap.Error(err))
	}

	err = app.New(&config).Run(context.Background())
	if err != nil {
		logs.Fatal("failed to run app", zap.Error(err))
	}

	systemQuit := make(chan os.Signal, 1)
	signal.Notify(systemQuit, os.Interrupt)

	<-systemQuit
}
package app


import (
	"context"
	tgbotapi "github.com/go-telegram-bot-api/telegram-bot-api/v5"
	"github.com/greg-source/job-search/internal/model"
	"github.com/greg-source/job-search/internal/parts/cache"
	"github.com/greg-source/job-search/internal/parts/database"
	"github.com/greg-source/job-search/internal/parts/notification/listener"
	"github.com/greg-source/job-search/internal/parts/notification/poller"
)


type App struct {
	config *model.Config
}


func New(config *model.Config) *App {
	return &App{config: config}
}
```

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | | ІАЛЦ.467200.007 Д4 | | |
| | | № докум. | Підпис | Дата | | | |
| Розробив | | Грициняк Г.С. | | | *Телеграм-бот для пошуку роботи* Текст програмного коду | Літ. | Аркуш 1 / Аркушів 41 |
| Перевірив | | Нечай Д.О. | | | | | |
| Н. Контр. | | Ковальчук О. М | | | | КПІ ім. Ігоря Сікорського, ФІОТ, ІО-06 | |
| Затвердив | | | | | | | |

```
func (c *App) Run(ctx context.Context) error {
	pos := database.New(&c.config.PostgresConfig)

	err := pos.Connect(ctx)
	if err != nil {
		return err
	}

	client := cache.New(&c.config.RedisConfig)

	err = client.Connect(ctx)
	if err != nil {
		return err
	}

	bot, err := tgbotapi.NewBotAPI(c.config.Token)
	if err != nil {
		return err
	}

	lis := listener.New(bot, client, pos)
	notifier := poller.NewListener(bot, client, pos)

	go lis.Listen(ctx)
	go notifier.Listen(ctx)

	return nil
}
package logs

import (
	"go.uber.org/zap"
	"go.uber.org/zap/zapcore"
)

var zapLog *zap.Logger

func init() {
	var err error

	config := zap.NewProductionConfig()
	enccoderConfig := zap.NewProductionEncoderConfig()
```

```
	zapcore.TimeEncoderOfLayout("Jan _2 15:04:05.000000000")

	enccoderConfig.StacktraceKey = "" // to hide stacktrace info
	config.EncoderConfig = enccoderConfig

	zapLog, err = config.Build(zap.AddCallerSkip(1))
	if err != nil {
		panic(err)
	}
}

func Info(message string, fields ...zap.Field) {
	zapLog.Info(message, fields...)
}

func Debug(message string, fields ...zap.Field) {
	zapLog.Debug(message, fields...)
}

func Error(err error) {
	zapLog.Error("", zap.Error(err))
}

func Fatal(message string, fields ...zap.Field) {
	zapLog.Fatal(message, fields...)
}package model

import "errors"

var (
	ErrInternal = errors.New("internal error")
) package model

// Config stores all configuration of the application.
// The values are read by viper from a config file or environment variable.
type Config struct {
	AppConfig
	PostgresConfig
	RedisConfig
}

type AppConfig struct {
	Token string `env:"TOKEN"`
```

```
}

type PostgresConfig struct {
	Host     string `env:"POSTGRES_HOST"`
	Port     int    `env:"POSTGRES_PORT"`
	User     string `env:"POSTGRES_USER"`
	Password string `env:"POSTGRES_PASSWORD"`
	DbName   string `env:"POSTGRES_DBNAME"`
}

type RedisConfig struct {
	Host string `env:"REDIS_HOST"`
	Port int    `env:"REDIS_PORT"`
}package model

import (
	"fmt"
	tgbotapi "github.com/go-telegram-bot-api/telegram-bot-api/v5"
)

type UserStatus string

const Unauthorized UserStatus = "UNAUTHORIZED"
const Authorized UserStatus = "AUTHORIZED"
const Select UserStatus = "SELECT"
const LIST_SUBSCRIPTIONS UserStatus = "List 👁"
const ADD_SUBSCRIPTIONS UserStatus = "Add ✅"
const DELETE_SUBSCRIPTIONS UserStatus = "Delete ❌"
const Type UserStatus = "Type"
const Query UserStatus = "QUERY"

type User struct {
	Username string     `json:"username"`
	Status   UserStatus `json:"status"`
	ChatID   int        `json:"chatid"`
}

type SubscriptionType string

const DOU SubscriptionType = "dou"
const ROBOTA_UA SubscriptionType = "robota.ua"
const DJINNI SubscriptionType = "djinni"
const TEST SubscriptionType = "test"
```

```
var DOMAINS = []string{string(DOU), string(ROBOTA_UA), string(DJINNI), string(TEST)}
var     CHOICES     =     []string{string(LIST_SUBSCRIPTIONS),     string(ADD_SUBSCRIPTIONS),
string(DELETE_SUBSCRIPTIONS)}

type Subscription struct {
       ChatID       int              `json:"chatid"`
       Subscription string           `json:"subscription"`
       Type         SubscriptionType `json:"type"`
       Updatetime   int              `json:"updatetime"`
}

const StatusStart UserStatus = "Start"

const USERS string = "users"
const SUBSCRIPTIONS string = "subscriptions"
const DEL_SUBSCRIPTIONS string = "delete_subscriptions"

type Notification struct {
       Title  string
       Link   string
       ChatID int64
       Query  string
}

func (c *Notification) TgMessage() tgbotapi.MessageConfig {
       descr := fmt.Sprintf("_%s_", fmt.Sprintf("(subscription: %s)", c.Query))
       title := fmt.Sprintf("**%s**", c.Title)

       msg   :=   tgbotapi.NewMessage(c.ChatID,   fmt.Sprintf("%s\n%s\n\n%s",   title,   descr,
c.Link))
       msg.ParseMode = "Markdown"

       return msg
}package model

import "sync"

type Vacancies struct {
       vacancies map[string]string
       mu        *sync.Mutex
}
```

```
func NewVacancies(vacancies map[string]string) *Vacancies {
	return &Vacancies{vacancies: vacancies, mu: new(sync.Mutex)}
}

func (c *Vacancies) Add(title string, link string) {
	c.mu.Lock()
	defer c.mu.Unlock()

	c.vacancies[title] = link
}

func (c *Vacancies) GetAll() map[string]string {
	c.mu.Lock()
	defer c.mu.Unlock()

	return c.vacancies
}package cache

import (
	"context"
	"encoding/json"
	"fmt"
	"github.com/greg-source/job-search/internal/model"
	"github.com/redis/go-redis/v9"
)

type RedisCache struct {
	config *model.RedisConfig
	redis.Client
}

func New(config *model.RedisConfig) *RedisCache {
	return &RedisCache{config: config}
}

func (c *RedisCache) Connect(ctx context.Context) error {
	client := redis.NewClient(&redis.Options{
		Addr: fmt.Sprintf("%s:%d", c.config.Host, c.config.Port),
	})

	status := client.Ping(ctx)
	if status.Err() != nil {
		return status.Err()
```

```
	}

	c.Client = *client

	return nil
}

func (c *RedisCache) SetUser(ctx context.Context, user *model.User) error {
	jsonData, err := json.Marshal(*user)
	if err != nil {
		return err
	}

	return c.setMap(ctx, model.USERS, user.Username, string(jsonData))
}

func (c *RedisCache) GetUser(ctx context.Context, username string) (*model.User, error) {
	res, err := c.getMap(ctx, model.USERS, username)
	if err != nil {
		return nil, err
	}

	ress := res.([]interface{})
	if len(ress) == 0 || ress[0] == nil {
		return nil, model.ErrInternal
	}

	jsonData := ress[0].(string)

	var user model.User

	err = json.Unmarshal([]byte(jsonData), &user)
	if err != nil {
		return nil, err
	}

	return &user, nil
}

func (c *RedisCache) SetSubscription(ctx context.Context, key string, user
*model.Subscription) error {
	jsonData, err := json.Marshal(*user)
	if err != nil {
```

```
		return err
	}

	return c.pushList(ctx, key, jsonData)
}

func (c *RedisCache) GetSubscription(ctx context.Context, key string) (*model.Subscription,
error) {
	res, err := c.popList(ctx, key)
	if err != nil {
		return nil, err
	}

	ress := res.([]string)

	var subscription model.Subscription

	err = json.Unmarshal([]byte(ress[1]), &subscription)
	if err != nil {
		return nil, err
	}

	return &subscription, nil
}

func (c *RedisCache) Set(ctx context.Context, key string, value string) error {
	return c.Client.Set(ctx, key, value, 0).Err()
}

func (c *RedisCache) Get(ctx context.Context, key string) (string, error) {
	res := c.Client.Get(ctx, key)
	if res.Err() != nil {
		return "", res.Err()
	}

	return res.Val(), nil
}

func (c *RedisCache) setMap(ctx context.Context, key string, value ...interface{}) error {
	return c.Client.HMSet(ctx, key, value).Err()
}
```

```
func (c *RedisCache) getMap(ctx context.Context, mp string, key string) (interface{}, error)
{
	return c.Client.HMGet(ctx, mp, key).Result()
}

func (c *RedisCache) pushList(ctx context.Context, key string, value ...interface{}) error {
	return c.Client.LPush(ctx, key, value...).Err()
}

func (c *RedisCache) popList(ctx context.Context, key string) (interface{}, error) {
	return c.Client.BLPop(ctx, 0, key).Result()
}

func (c *RedisCache) getAllList(ctx context.Context, key string) (interface{}, error) {
	return c.Client.LRange(ctx, model.SUBSCRIPTIONS, 0, -1).Result()
}package database

import (
	"context"
	"database/sql"
	"fmt"
	"github.com/greg-source/job-search/internal/model"
	"github.com/greg-source/job-search/internal/parts/sqlc"
	"github.com/greg-source/job-search/internal/utils"
	_ "github.com/lib/pq"
)

type Postgres struct {
	config *model.PostgresConfig
	sqlc.Querier
}

func New(config *model.PostgresConfig) *Postgres {
	return &Postgres{config: config}
}

func (c *Postgres) Connect(ctx context.Context) error {
	dbUrlTemplate := "postgresql://%s:%s@%s:%d/%s?sslmode=disable"
	dbUrl := fmt.Sprintf(dbUrlTemplate, c.config.User, c.config.Password, c.config.Host,
c.config.Port, c.config.DbName)

	connection, err := sql.Open("postgres", dbUrl)
	if err != nil {
```

```
		return err
	}

	err = connection.Ping()
	if err != nil {
		return err
	}

	err = utils.ApplyMigrations(connection)
	if err != nil {
		return err
	}

	c.Querier = sqlc.New(connection)

	return nil
}package listener

import (
	"context"
	tgbotapi "github.com/go-telegram-bot-api/telegram-bot-api/v5"
	core "github.com/greg-source/job-search/internal"
	logs "github.com/greg-source/job-search/internal/logger"
	"github.com/greg-source/job-search/internal/model"
	"github.com/greg-source/job-search/internal/parts/notification/listener/handlers"
	"github.com/greg-source/job-
search/internal/parts/notification/listener/handlers/subcription"
)

type Listener struct {
	database core.Database
	cache    core.Cache
	bot      *tgbotapi.BotAPI
}

func New(bot *tgbotapi.BotAPI, cache core.Cache, database core.Database) core.Listener {
	return &Listener{bot: bot, cache: cache, database: database}
}

func (c *Listener) Listen(ctx context.Context) {
	u := tgbotapi.NewUpdate(0)
	u.Timeout = 60
```

```
	updates := c.bot.GetUpdatesChan(u)
	handler := c.getHandler()

	for update := range updates {
		var username string
		if update.Message != nil {
			username = update.Message.Chat.UserName
		} else if update.CallbackQuery != nil {
			username = update.CallbackQuery.From.UserName
		}

		user, err := c.cache.GetUser(ctx, username)

		handlerContext := &core.HandlerContext{
			Update: &update,
			User:   user,
		}

		err = handler.Execute(ctx, handlerContext)

		if err != nil {
			handlerContext.Message   =   tgbotapi.NewMessage(update.Message.Chat.ID,
model.ErrInternal.Error())

			continue
		}

		err = c.cache.SetUser(ctx, handlerContext.User)

		if err != nil {
			logs.Error(err)
		}

		_, err = c.bot.Send(handlerContext.Message)
		if err != nil {
			logs.Error(err)
		}
	}
}

func (c *Listener) getHandler() core.Handler {
	listHandler := subcription.NewList(c.cache, c.database)
```

```
	selectHandler := subcription.NewSelect(c.cache)
	selectHandler.SetNext(listHandler)

	deleteHandler := subcription.NewDelete(c.cache, c.database)
	deleteHandler.SetNext(selectHandler)

	startHandler := handlers.NewStart(c.database, c.cache)
	startHandler.SetNext(deleteHandler)

	return startHandler
}package handlers

import (
	"context"
	"database/sql"
	"errors"
	"fmt"
	tgbotapi "github.com/go-telegram-bot-api/telegram-bot-api/v5"
	core "github.com/greg-source/job-search/internal"
	logs "github.com/greg-source/job-search/internal/logger"
	"github.com/greg-source/job-search/internal/model"
	"github.com/greg-source/job-search/internal/parts/sqlc"
)

type Start struct {
	database core.Database
	cache    core.Cache

	handler core.Handler
}

func NewStart(database core.Database, cache core.Cache) *Start {
	return &Start{database: database, cache: cache}
}

func (c *Start) SetNext(handler core.Handler) {
	c.handler = handler
}

func (c *Start) Execute(ctx context.Context, handlerContext *core.HandlerContext) error {
	update := handlerContext.Update

	message := GetMessage(update)
```

```
	handlerContext.Message = message

	if handlerContext.User == nil {
		chatID := update.Message.Chat.ID
		username := update.Message.Chat.UserName

		// Try to get from DB
		dbUser, err := c.database.GetUser(context.Background(), username)
		if errors.Is(err, sql.ErrNoRows) {
			// If not exist - create...
			message.Text = fmt.Sprintf("Hello %s! Welcome to the job search bot!\nChoose action:", username)

			err := c.database.CreateUser(context.Background(), sqlc.CreateUserParams{
				Username: username,
				Status:   string(model.StatusStart),
				Chatid:   chatID,
			})

			if err != nil {
				logs.Error(err)

				return model.ErrInternal
			}

			handlerContext.User = &model.User{
				Username: dbUser.Username,
				Status:   model.UserStatus(dbUser.Status),
				ChatID:   int(dbUser.Chatid),
			}

			return nil

		} else if err != nil {
			logs.Error(err)

			return model.ErrInternal
		}

		handlerContext.User = &model.User{
			Username: dbUser.Username,
```

```
                Status:   model.UserStatus(dbUser.Status),
                ChatID:   int(dbUser.Chatid),
            }
        }

        if handlerContext.User.Status == model.StatusStart && update.CallbackQuery == nil {
            switch update.Message.Text {
            case    string(model.ADD_SUBSCRIPTIONS),    string(model.LIST_SUBSCRIPTIONS),
string(model.DELETE_SUBSCRIPTIONS):
                handlerContext.User.Status = model.UserStatus(update.Message.Text)

            default:
                message.Text = "Choose action:"

                return nil
            }
        }

        if c.handler == nil {
            return model.ErrInternal
        }

        return c.handler.Execute(ctx, handlerContext)
}

func GetMainKeyboard(options ...string) *tgbotapi.ReplyKeyboardMarkup {
        buttons := make([]tgbotapi.KeyboardButton, len(options))

        for i, option := range options {
            buttons[i] = tgbotapi.NewKeyboardButton(option)
        }

        row1 := tgbotapi.NewKeyboardButtonRow(buttons...)
        keyboard := tgbotapi.NewReplyKeyboard(row1)

        keyboard.OneTimeKeyboard = true

        return &keyboard
}

func GetInlineKeyboard(options ...string) *tgbotapi.InlineKeyboardMarkup {
        buttons := make([]tgbotapi.InlineKeyboardButton, len(options))
```

```
	for i, option := range options {
		buttons[i] = tgbotapi.NewInlineKeyboardButtonData(option, option)
	}

	row := tgbotapi.NewInlineKeyboardRow(buttons...)
	keyboard := tgbotapi.NewInlineKeyboardMarkup(row)

	return &keyboard
}

func GetMessage(update *tgbotapi.Update) *tgbotapi.MessageConfig {
	message := new(tgbotapi.MessageConfig)

	message.ReplyMarkup = GetMainKeyboard(model.CHOICES...)

	if update.CallbackQuery != nil {
		message.ChatID = update.CallbackQuery.From.ID
		update.Message = update.CallbackQuery.Message
	} else {
		message.ChatID = update.Message.Chat.ID
	}

	return message
}package subcription

import (
	"context"
	"errors"
	core "github.com/greg-source/job-search/internal"
	logs "github.com/greg-source/job-search/internal/logger"
	"github.com/greg-source/job-search/internal/model"
	"github.com/greg-source/job-search/internal/parts/notification/listener/handlers"
	"strings"
)

type Select struct {
	cache core.Cache

	handler core.Handler
}

func NewSelect(cache core.Cache) *Select {
	return &Select{cache: cache}
```

```
}

func (c *Select) SetNext(handler core.Handler) {
	c.handler = handler
}

func (c *Select) Execute(ctx context.Context, handlerContext *core.HandlerContext) error {
	update := handlerContext.Update
	message := handlers.GetMessage(update)

	handlerContext.Message = message

	switch handlerContext.User.Status {
	case model.ADD_SUBSCRIPTIONS:
		message.Text = "Select new subscription:"
		message.ReplyMarkup = handlers.GetMainKeyboard(model.DOMAINS...)

		handlerContext.User.Status = model.Type

		return nil

	case model.Query:
		query := update.Message.Text
		typ, err := c.cache.Get(ctx, handlerContext.User.Username)
		if err != nil {
			logs.Error(err)
			return err
		}

		err = c.cache.SetSubscription(ctx, model.SUBSCRIPTIONS, &model.Subscription{
			ChatID:       handlerContext.User.ChatID,
			Subscription: strings.ToLower(query),
			Type:         model.SubscriptionType(typ),
			Updatetime:   30,
		})
		if err != nil {
			logs.Error(err)
			return err
		}

		message.Text = "Successfully added query. You`ll receive notification if any
new vacancies will appear..."
		handlerContext.User.Status = model.StatusStart
```

```
		return nil

	case model.Type:
		switch update.Message.Text {
		case	string(model.DOU),	string(model.ROBOTA_UA),	string(model.DJINNI),
string(model.TEST):
			err	:=	c.cache.Set(ctx,	handlerContext.User.Username,
update.Message.Text)
			if err != nil {
				logs.Error(err)

				return err
			}

			handlerContext.User.Status = model.Query
			message.Text = "Set search query to create notifications"
			message.ReplyMarkup = nil

			if err != nil {
				logs.Error(err)

				return err
			}

		default:
			message.Text = "Select new subscription:"
			message.ReplyMarkup = handlers.GetMainKeyboard(model.DOMAINS...)
		}

		return nil
	}

	if c.handler == nil {
		logs.Error(errors.New("no handlers found"))

		return model.ErrInternal
	}

	return c.handler.Execute(ctx, handlerContext)
}package subcription

import (
```

```
	"context"
	"errors"
	"fmt"
	tgbotapi "github.com/go-telegram-bot-api/telegram-bot-api/v5"
	core "github.com/greg-source/job-search/internal"
	logs "github.com/greg-source/job-search/internal/logger"
	"github.com/greg-source/job-search/internal/model"
	"github.com/greg-source/job-search/internal/parts/notification/listener/handlers"
	"github.com/greg-source/job-search/internal/parts/sqlc"
	"strings"
)

type Delete struct {
	cache    core.Cache
	database core.Database

	handler core.Handler
}

func NewDelete(cache core.Cache, database core.Database) *Delete {
	return &Delete{cache: cache, database: database}
}

func (c *Delete) SetNext(handler core.Handler) {
	c.handler = handler
}

func (c *Delete) Execute(ctx context.Context, handlerContext *core.HandlerContext) error {
	update := handlerContext.Update
	user := handlerContext.User

	if update.CallbackQuery != nil {
		// Handle callback query
		callback := update.CallbackQuery

		// Edit the inline button message
		message := tgbotapi.NewEditMessageText(
			callback.Message.Chat.ID,
			callback.Message.MessageID,
			callback.Message.Text,
		)
```

```
		subscriptions,	err	:=	c.database.GetSubscriptionsByChatID(ctx,
int64(user.ChatID))
		if err != nil {
			logs.Error(err)

			return model.ErrInternal
		}

		for _, subscription := range subscriptions {
			data := strings.Split(callback.Data, " - ")
			if len(data) != 2 {
				logs.Info("invalid data")

				return model.ErrInternal
			}

			typ := data[0]
			query := data[1]
			if subscription.Subscription == query && subscription.Type == typ {
				err		:=		c.database.DeleteSubscription(ctx,
sqlc.DeleteSubscriptionParams{
					Type:		typ,
					Subscription: query,
					Chatid:		int64(user.ChatID),
				})
				if err != nil {
					logs.Error(err)

					return model.ErrInternal
				}

				err	=	c.cache.SetSubscription(ctx,	model.DEL_SUBSCRIPTIONS,
&model.Subscription{
					ChatID:		handlerContext.User.ChatID,
					Subscription: strings.ToLower(query),
					Type:		model.SubscriptionType(typ),
					Updatetime:	30,
				})

				if err != nil {
					logs.Error(err)

					return model.ErrInternal
```

```
                        }

                        var newKyboard tgbotapi.InlineKeyboardMarkup
                        newKyboard.InlineKeyboard                                    =
make([][]tgbotapi.InlineKeyboardButton, 1)
                        inlineKeyboard := callback.Message.ReplyMarkup

                        if len(inlineKeyboard.InlineKeyboard[0]) > 1 {
                                for _, a := range inlineKeyboard.InlineKeyboard[0] {
                                        data := strings.Split(a.Text, " - ")
                                        if len(data) != 2 {
                                                logs.Info("invalid data")

                                                return model.ErrInternal
                                        }

                                        typ := data[0]
                                        query := data[1]

                                        if    subscription.Subscription    !=    query    ||
subscription.Type != typ {
                                                newKyboard.InlineKeyboard[0]              =
append(newKyboard.InlineKeyboard[0], a)
                                        }
                                }

                                if err != nil {
                                        logs.Error(err)

                                        return model.ErrInternal
                                }

                                message.ReplyMarkup = &newKyboard
                                handlerContext.Message = message
                        } else {
                                handlerContext.Message                               =
tgbotapi.NewDeleteMessage(callback.Message.Chat.ID,
                                        callback.Message.MessageID)
                        }
                }
        }

        //buttons := make([]string, len(subscriptions))
```

```
		//
		//if len(subscriptions) == 0 {
		//	message.Text = "No subscriptions found"
		//
		//	return nil
		//} else {
		//	for i, subscription := range subscriptions {
		//		message.Text = "Choose subscription to delete"
		//		buttons[i] = fmt.Sprintf("%d. %s - %s\n", i+1, subscription.Type,
subscription.Subscription)
		//	}
		//}
		//
		//// Create new inline keyboard with updated button
		//inlineKeyboard := handlers.GetInlineKeyboard(buttons...)
		//message.ReplyMarkup = inlineKeyboard

		return nil
	}

	if user.Status == model.DELETE_SUBSCRIPTIONS {
		message := handlers.GetMessage(update)

		handlerContext.Message = message

		defer func() {
			user.Status = model.StatusStart
		}()

		subscriptions,	err	:=	c.database.GetSubscriptionsByChatID(ctx,
int64(user.ChatID))
		if err != nil {
			logs.Error(err)

			return model.ErrInternal
		}

		buttons := make([]string, len(subscriptions))

		if len(subscriptions) == 0 {
			message.Text = "No subscriptions found"

			return nil
```

```
		} else {
			for i, subscription := range subscriptions {
				message.Text = "Choose subscription to delete"
				buttons[i]   =   fmt.Sprintf("%s   -   %s",   subscription.Type,
subscription.Subscription)
			}
		}

		message.ReplyMarkup = handlers.GetInlineKeyboard(buttons...)

		return nil
	}

	if c.handler == nil {
		logs.Error(errors.New("no handlers found"))

		return model.ErrInternal
	}

	return c.handler.Execute(ctx, handlerContext)
}package subcription

import (
	"context"
	"errors"
	"fmt"
	core "github.com/greg-source/job-search/internal"
	logs "github.com/greg-source/job-search/internal/logger"
	"github.com/greg-source/job-search/internal/model"
	"github.com/greg-source/job-search/internal/parts/notification/listener/handlers"
)

type List struct {
	cache    core.Cache
	database core.Database

	handler core.Handler
}

func NewList(cache core.Cache, database core.Database) *List {
	return &List{cache: cache, database: database}
}
```

```
func (c *List) SetNext(handler core.Handler) {
	c.handler = handler
}

func (c *List) Execute(ctx context.Context, handlerContext *core.HandlerContext) error {
	update := handlerContext.Update
	message := handlers.GetMessage(update)

	handlerContext.Message = message

	user := handlerContext.User

	if user.Status == model.LIST_SUBSCRIPTIONS {
		defer func() {
			user.Status = model.StatusStart
		}()

		subscriptions, err := c.database.GetSubscriptionsByChatID(ctx,
int64(user.ChatID))
		if err != nil {
			logs.Error(err)

			return model.ErrInternal
		}

		if len(subscriptions) == 0 {
			message.Text = "No subscriptions found"

			return nil
		} else {
			for i, subscription := range subscriptions {
				message.Text += fmt.Sprintf("%d. %s - %s\n", i+1,
subscription.Type, subscription.Subscription)
			}
		}

		return nil
	}

	if c.handler == nil {
		logs.Error(errors.New("no handlers found"))

		return model.ErrInternal
```

```
	}

	return c.handler.Execute(ctx, handlerContext)
}package poller

import (
	"context"
	core "github.com/greg-source/job-search/internal"
	logs "github.com/greg-source/job-search/internal/logger"
	"github.com/greg-source/job-search/internal/model"
)

type Checker struct {
	old          *model.Vacancies
	parser       core.Parser
	subscription model.Subscription
}

func NewChecker(parser core.Parser, subscription model.Subscription) *Checker {
	return &Checker{parser: parser, subscription: subscription}
}

func (c *Checker) CheckVacancies(ctx context.Context, send chan<- model.Notification) {
	if c.old == nil {
		vacancies, err := c.parser.Parse(ctx)
		if err != nil {
			logs.Error(err)
		}

		c.old = model.NewVacancies(vacancies.GetAll())
	} else {
		parse, err := c.parser.Parse(ctx)
		if err != nil {
			logs.Error(err)
		}

		vacancies := parse.GetAll()

		for key, vacancy := range vacancies {
			if _, ok := c.old.GetAll()[key]; !ok {
				c.old.Add(key, vacancy)
				send <- model.Notification{
					Title:  key,
```

```
                                Link:   vacancy,
                                ChatID: int64(c.subscription.ChatID),
                                Query:  c.subscription.Subscription,
                            }
                        }
                }
        }
}package poller

import (
        "context"
        tgbotapi "github.com/go-telegram-bot-api/telegram-bot-api/v5"
        core "github.com/greg-source/job-search/internal"
        logs "github.com/greg-source/job-search/internal/logger"
        "github.com/greg-source/job-search/internal/model"
        "github.com/greg-source/job-search/internal/parts/sqlc"
)

type Listener struct {
        database core.Database
        cache    core.Cache
        bot      *tgbotapi.BotAPI

        poller core.Poller
}

func NewListener(bot *tgbotapi.BotAPI, cache core.Cache, database core.Database) core.Listener
{
        return &Listener{database: database, cache: cache, bot: bot, poller: NewPoller(bot)}
}

func (c *Listener) Listen(ctx context.Context) {
        c.launch(ctx)
        logs.Info("Listener retrieved subscription and started")

        go c.addIncomingSubscriptions(ctx)
        go c.removeIncomingSubscriptions(ctx)

        go c.poller.StartPolling(ctx)
}

func (c *Listener) launch(ctx context.Context) {
        // Boot up
```

```
	subscriptions, err := c.database.GetSubscriptions(ctx)
	if err != nil {
		logs.Error(err)

		return
	}

	for _, subscription := range subscriptions {
		c.poller.AddSubscription(ctx, model.Subscription{
			ChatID:       int(subscription.Chatid),
			Subscription: subscription.Subscription,
			Type:         model.SubscriptionType(subscription.Type),
			Updatetime:   int(subscription.Updatetime),
		})
	}
}

func (c *Listener) addIncomingSubscriptions(ctx context.Context) {
	for {
		subscription, err := c.cache.GetSubscription(ctx, model.SUBSCRIPTIONS)
		if err != nil {
			logs.Error(err)

			continue
		}

		c.poller.AddSubscription(ctx, *subscription)

		err = c.database.CreateSubscription(ctx, sqlc.CreateSubscriptionParams{
			Chatid:       int64(subscription.ChatID),
			Subscription: subscription.Subscription,
			Type:         string(subscription.Type),
			Updatetime:   30,
		})
	}
}

func (c *Listener) removeIncomingSubscriptions(ctx context.Context) {
	for {
		subscription, err := c.cache.GetSubscription(ctx, model.DEL_SUBSCRIPTIONS)
		if err != nil {
			logs.Error(err)
```

```
                        continue
                }

                c.poller.RemoveSubscription(ctx, *subscription)
        }
}package poller

import (
        "context"
        tgbotapi "github.com/go-telegram-bot-api/telegram-bot-api/v5"
        core "github.com/greg-source/job-search/internal"
        logs "github.com/greg-source/job-search/internal/logger"
        "github.com/greg-source/job-search/internal/model"
        "github.com/greg-source/job-search/internal/parts/parser"
        "sync"
        "time"
)

type Poller struct {
        bot *tgbotapi.BotAPI

        parserFactory core.ParserFactory
        checkers      map[model.Subscription]*Checker
        mu            *sync.Mutex
}

func NewPoller(bot *tgbotapi.BotAPI) core.Poller {
        return &Poller{
                bot:           bot,
                checkers:      make(map[model.Subscription]*Checker),
                parserFactory: parser.NewParserFactory(),
                mu:            new(sync.Mutex)}
}

func (c *Poller) AddSubscription(ctx context.Context, subscription model.Subscription) {
        c.mu.Lock()
        defer c.mu.Unlock()

        parser := c.parserFactory.GetParser(subscription)
        if parser == nil {
                return
        }
```

```
	c.checkers[subscription] = NewChecker(parser, subscription)
	c.checkers[subscription].CheckVacancies(ctx, make(chan model.Notification))
}

func (c *Poller) RemoveSubscription(ctx context.Context, subscription model.Subscription) {
	c.mu.Lock()
	defer c.mu.Unlock()

	delete(c.checkers, subscription)
}

func (c *Poller) StartPolling(ctx context.Context) {
	for {
		c.mu.Lock()

		var wg sync.WaitGroup
		notifications := make(chan model.Notification, 20)

		for _, subscription := range c.checkers {
			wg.Add(1)

			go func(notifications chan<- model.Notification) {
				defer wg.Done()

				time.Sleep(time.Duration(subscription.subscription.Updatetime)  *
time.Second)
				subscription.CheckVacancies(ctx, notifications)

			}(notifications)
		}

		c.mu.Unlock()

		go func() {
			wg.Wait()
			close(notifications)
		}()

		for notif := range notifications {
			_, err := c.bot.Send(notif.TgMessage())
			if err != nil {
				logs.Error(err)
			}
```

```
		}
	}
}package parser

import (
	core "github.com/greg-source/job-search/internal"
	logs "github.com/greg-source/job-search/internal/logger"
	"github.com/greg-source/job-search/internal/model"
	"github.com/greg-source/job-search/internal/parts/parser/djinni"
	"github.com/greg-source/job-search/internal/parts/parser/dou"
	"github.com/greg-source/job-search/internal/parts/parser/robotaua"
	"github.com/greg-source/job-search/internal/parts/parser/test"
)

type Factory struct {
}

func NewParserFactory() *Factory {
	return &Factory{}
}

func (c *Factory) GetParser(subscription model.Subscription) core.Parser {
	switch subscription.Type {
	case model.DOU:
		return dou.New(subscription.Subscription)
	case model.ROBOTA_UA:
		return robotaua.New(subscription.Subscription)
	case model.DJINNI:
		return djinni.New(subscription.Subscription)
	case model.TEST:
		return test.New(subscription.Subscription)
	default:
		logs.Info("invalid subscription")
	}

	return nil
}package djinni

import (
	"context"
	"fmt"
	"github.com/greg-source/job-search/internal/model"
	"github.com/greg-source/job-search/internal/utils"
```

```
	"golang.org/x/net/html"
	"strings"
)

const djinniLink = "https://djinni.co/jobs/?keywords=%s"

type Parser struct {
	query string
}

func New(query string) *Parser {
	return &Parser{query: query}
}

func parseHtml(n *html.Node, vacancies *model.Vacancies) {
	switch n.Data {
	case "a":
		// check if FirstChild node of the h2 element is a text
		if n.FirstChild != nil && n.FirstChild.Type == html.TextNode {
			if len(n.Attr) == 2 && n.Attr[0].Key == "class" {
				st := strings.ReplaceAll(n.FirstChild.Data, " ", "")
				vacancies.Add(strings.ReplaceAll(st,          "\n",          ""),
fmt.Sprintf("https://djinni.co%s", n.Attr[1].Val))
			}
		}
	}

	// Traverse child nodes
	for c := n.FirstChild; c != nil; c = c.NextSibling {
		parseHtml(c, vacancies)
	}
}

func (c *Parser) Parse(ctx context.Context) (*model.Vacancies, error) {
	html, err := utils.GetHTML(fmt.Sprintf(djinniLink, c.query))
	if err != nil {
		return nil, err
	}

	vacancies := model.NewVacancies(make(map[string]string, 20))

	parseHtml(html, vacancies)
```

```
	return vacancies, nil
}package dou

import (
	"context"
	"fmt"
	"github.com/greg-source/job-search/internal/model"
	"github.com/greg-source/job-search/internal/utils"
	"golang.org/x/net/html"
)

const douLink = "https://jobs.dou.ua/vacancies/?search=%s"

type Parser struct {
	query string
}

func New(query string) *Parser {
	return &Parser{query: query}
}

func parseHtml(n *html.Node, vacancies *model.Vacancies) {
	switch n.Data {
	case "a":
		// check if FirstChild node of the h2 element is a text
		if n.FirstChild != nil && n.FirstChild.Type == html.TextNode {
			if len(n.Attr) == 2 && n.Attr[0].Key == "class" && n.Attr[0].Val == "vt"
{
				vacancies.Add(n.FirstChild.Data, n.Attr[1].Val)
			}
		}
	}

	// Traverse child nodes
	for c := n.FirstChild; c != nil; c = c.NextSibling {
		parseHtml(c, vacancies)
	}
}

func (c *Parser) Parse(ctx context.Context) (*model.Vacancies, error) {
	html, err := utils.GetHTML(fmt.Sprintf(douLink, c.query))
	if err != nil {
		return nil, err
```

```
	}

	vacancies := model.NewVacancies(make(map[string]string, 20))

	parseHtml(html, vacancies)

	return vacancies, nil
}package robotaua

import (
	"context"
	"encoding/json"
	"fmt"
	"github.com/greg-source/job-search/internal/model"
	"io"
	"net/http"
)

const robotauaLink = "https://api.rabota.ua/vacancy/search?keyWords=golang"

type Parser struct {
	query string
}

func New(query string) *Parser {
	return &Parser{query: query}
}

type Document struct {
	Name    string `json:"name"`
	Company int    `json:"notebookId"`
	Vacancy int    `json:"id"`
}

var Response struct {
	Documents []Document `json:"documents"`
}

func (c *Parser) Parse(ctx context.Context) (*model.Vacancies, error) {
	resp, err := http.Get(robotauaLink)
	if err != nil {
		return nil, err
	}
```

```
	defer resp.Body.Close()

	all, err := io.ReadAll(resp.Body)
	if err != nil {
		return nil, err
	}

	err = json.Unmarshal(all, &Response)
	if err != nil {
		return nil, err
	}

	vacancies := model.NewVacancies(make(map[string]string, 20))

	for _, doc := range Response.Documents {
		vacancies.Add(doc.Name,
fmt.Sprintf("https://robota.ua/ru/company%d/vacancy%d", doc.Company, doc.Vacancy))
	}

	return vacancies, nil
}package utils

import (
	"github.com/caarlos0/env/v11"
	"github.com/greg-source/job-search/internal/model"
)

// LoadConfig reads configuration from environment variables or file.
func LoadConfig() (config model.Config, err error) {
	if err := env.Parse(&config); err != nil {
		return model.Config{}, err
	}

	return config, nil
}package utils

import (
	"database/sql"
	logs "github.com/greg-source/job-search/internal/logger"
	"go.uber.org/zap"
	"os"
	"path/filepath"
	"strings"
```

```
)

var migrationDir = "/migrations"

func ApplyMigrations(db *sql.DB) error {
	rootDir, err := filepath.Abs(".")
	if err != nil {
		return err
	}
	migrationDir = rootDir + migrationDir
	files, err := os.ReadDir(migrationDir)
	if err != nil {
		return err
	}
	for _, file := range files {
		if !file.IsDir() && strings.HasSuffix(file.Name(), ".sql") {
			migrationPath := filepath.Join(migrationDir, file.Name())
			migrationSQL, err := os.ReadFile(migrationPath)
			if err != nil {
				return err
			}

			_, err = db.Exec(string(migrationSQL))
			if err != nil {
				return err
			}
			logs.Info("Applied migration: %s", zap.String("key", file.Name()))
		}
	}
	return nil
}package utils

import (
	"golang.org/x/net/html"
	"net/http"
)

func GetHTML(url string) (*html.Node, error) {
	// Make the GET request
	resp, err := http.Get(url)
	if err != nil {
		return nil, err
	}
```

```
       defer resp.Body.Close()

       return html.Parse(resp.Body)
}
FROM debian:bullseye-slim

# Install dependencies and gcsfuse
RUN apt-get update && apt-get install -y --no-install-recommends \
    ca-certificates \
    curl \
    gnupg \
    fuse \
    rsync \
    && apt-get clean \
    && rm -rf /var/lib/apt/lists/*

RUN export GCSFUSE_REPO=gcsfuse-bullseye \
    &&   echo   "deb   http://packages.cloud.google.com/apt   $GCSFUSE_REPO   main"   |   tee
/etc/apt/sources.list.d/gcsfuse.list \
    && curl https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add - \
    && apt-get update \
    && apt-get install -y gcsfuse \
    && rm -rf /var/lib/apt/lists/*

ENTRYPOINT ["sh", "-c"]
CMD ["sleep 3600"]apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ include "app.fullname" . }}
  labels:
    {{- include "app.labels" . | nindent 4 }}
spec:
  {{- if not .Values.autoscaling.enabled }}
  replicas: {{ .Values.replicaCount }}
  {{- end }}
  selector:
    matchLabels:
      {{- include "app.selectorLabels" . | nindent 6 }}
  template:
    metadata:
      {{- with .Values.podAnnotations }}
      annotations:
        {{- toYaml . | nindent 8 }}
```

```
    {{- end }}
    labels:
      {{- include "app.labels" . | nindent 8 }}
      {{- with .Values.podLabels }}
      {{- toYaml . | nindent 8 }}
      {{- end }}
  spec:
    {{- with .Values.imagePullSecrets }}
    imagePullSecrets:
      {{- toYaml . | nindent 8 }}
    {{- end }}
    serviceAccountName: {{ include "app.serviceAccountName" . }}
    securityContext:
      {{- toYaml .Values.podSecurityContext | nindent 8 }}
    containers:
      - name: {{ .Chart.Name }}
        env:
          - name: TOKEN
            value: {{ .Values.global.app.TOKEN }}
          - name: POSTGRES_USER
            value: {{ .Values.global.postgres.USER }}
          - name: POSTGRES_PASSWORD
            value: {{ .Values.global.postgres.PASSWORD }}
          - name: POSTGRES_DB
            value: {{ .Values.global.postgres.DB }}
          - name: POSTGRES_HOST
            value: {{ .Values.global.postgres.HOST }}
          - name: POSTGRES_PORT
            value: "{{ .Values.global.postgres.PORT }}"
          - name: REDIS_HOST
            value: {{ .Values.global.redis.HOST }}
          - name: REDIS_PORT
            value: "{{ .Values.global.redis.PORT }}"
        securityContext:
          {{- toYaml .Values.securityContext | nindent 12 }}
        image:   "{{   .Values.image.repository   }}:{{   .Values.image.tag   |   default
.Chart.AppVersion }}"
        imagePullPolicy: {{ .Values.image.pullPolicy }}
        ports:
          - name: http
            containerPort: {{ .Values.service.port }}
            protocol: TCP
        resources:
```

```
            {{- toYaml .Values.resources | nindent 12 }}
          {{- with .Values.volumeMounts }}
          volumeMounts:
            {{- toYaml . | nindent 12 }}
          {{- end }}
      {{- with .Values.volumes }}
      volumes:
        {{- toYaml . | nindent 8 }}
      {{- end }}
      {{- with .Values.nodeSelector }}
      nodeSelector:
        {{- toYaml . | nindent 8 }}
      {{- end }}
      {{- with .Values.affinity }}
      affinity:
        {{- toYaml . | nindent 8 }}
      {{- end }}
      {{- with .Values.tolerations }}
      tolerations:
        {{- toYaml . | nindent 8 }}
      {{- end }}apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: {{ include "chadbot-psql.fullname" . }}
  labels:
    {{- include "chadbot-psql.labels" . | nindent 4 }}
spec:
  replicas: {{ .Values.replicaCount }}
  serviceName: {{ include "chadbot-psql.fullname" . }}
  selector:
    matchLabels:
      {{- include "chadbot-psql.selectorLabels" . | nindent 6 }}
  template:
    metadata:
      {{- with .Values.podAnnotations }}
      annotations:
        {{- toYaml . | nindent 8 }}
      {{- end }}
      labels:
        {{- include "chadbot-psql.selectorLabels" . | nindent 8 }}
    spec:
      {{- with .Values.imagePullSecrets }}
      imagePullSecrets:
```

```
        {{- toYaml . | nindent 8 }}
      {{- end }}
      serviceAccountName: {{ include "chadbot-psql.serviceAccountName" . }}
      securityContext:
        {{- toYaml .Values.podSecurityContext | nindent 8 }}
      containers:
        - name: {{ .Chart.Name }}
          image:   "{{   .Values.image.repository   }}:{{   .Values.image.tag   |   default
.Chart.AppVersion }}"
          imagePullPolicy: {{ .Values.image.pullPolicy }}
          ports:
            - name: http
              containerPort: {{ .Values.service.port }}
              protocol: TCP
          livenessProbe:
            exec:
              command:
                - /bin/sh
                - -c
                - exec psql -U ${POSTGRES_USER} -d $POSTGRES_DB -c SELECT 1
            failureThreshold: 6
            initialDelaySeconds: 5
            periodSeconds: 10
            successThreshold: 1
            timeoutSeconds: 5
          readinessProbe:
            exec:
              command:
                - /bin/sh
                - -c
                - exec psql -U ${POSTGRES_USER} -d $POSTGRES_DB -c SELECT 1
            failureThreshold: 6
            initialDelaySeconds: 30
            periodSeconds: 10
            successThreshold: 1
            timeoutSeconds: 51
          resources:
            {{- toYaml .Values.resources | nindent 12 }}
          envFrom:
            - configMapRef:
                name: {{ include "chadbot-psql.fullname" . }}
          volumeMounts:
            - mountPath: ./data/postgres:/var/lib/postgresql/data
```

```
              name: postgredb
      volumes:
        - name: postgredb
          persistentVolumeClaim:
            claimName: {{ include "chadbot-psql.fullname" . }}—

Create the first table to store username and status
CREATE TABLE IF NOT EXISTS users (
                                    username VARCHAR(28) NOT NULL,
    status VARCHAR(20) NOT NULL,
    chatID BIGINT PRIMARY KEY
    );

-- Create the second table to store username, subscription, type, updatetime, and query
CREATE TABLE IF NOT EXISTS subscriptions (
                                            id SERIAL PRIMARY KEY,
                                            chatID BIGINT NOT NULL,
                                            subscription VARCHAR(255) NOT NULL,
    type VARCHAR(20) NOT NULL,
    updatetime INT NOT NULL,
    CONSTRAINT unique_fields_constraint UNIQUE (chatID, subscription, type)
    );
```