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)
“__” _____2021 р.

Дипломний проєкт

на здобуття ступеня бакалавра

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

спеціальності 123 “Комп’ютерна інженерія”

на тему: Система автоматизації процесу збору даних

Виконав (-ла): студент (-ка) 4 курсу, групи ІВ-72
(шифр групи)

Киба Олег Дмитрович _____
(прізвище, ім’я, по батькові) (підпис)

Керівник ст. викладач Алещенко О. В. _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант (нормоконтроль) проф. д.т.н. Сімоненко В.П. _____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент _____ _____
(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 – 2021 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)
“__” _____2021 р.

ЗАВДАННЯ
на бакалаврський дипломний проєкт студента

Киби Олега Дмитровича

1. Тема проєкту *Система автоматизації процесу збору даних*
керівник проєкту Алещенко Олексій Вадимович, ст. викладач,
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від 11.05 2021 року № 1139-с
2. Термін здачі студентом закінченого проєкту *2 червня 2021 р.*
3. Вихідні дані до проєкту *прикладний програмний інтерфейс.*
4. Зміст розрахунково-пояснювальної записки (перелік питань, які розробляються)
Опис предметної області, аналіз наявних програмних рішень, розробка архітектури системи для автоматизації процесу збору даних, тестування та

впровадження сервісу

5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень) діаграма класів, діаграма розгортання застосунку, блок-схеми алгоритмів роботи програми

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

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормконтроль	Сімоненко В. П.		

7. Дата видачі завдання 15.12.2020

Календарний план

№ п/п	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту	Примітки
1.	*Затвердження теми проєкту*	*10.12.2020-15.12.2020*	
2.	*Вивчення та аналіз завдання*	*15.12.2021-15.03.2021*	
3.	*Розробка архітектури та загальної структури системи*	*15.03.2021-25.03.2021*	
4.	*Розробка структур окремих підсистем*	*25.03.2021-5.04.2021*	
5.	*Програмна реалізація системи*	*5.04.2021-15.04.2021*	
6.	*Оформлення пояснювальної записки*	*15.04.2021-20.05.2021*	
7.	*Захист програмного продукту*	*25.04.2021*	
8.	*Передзахист*	*25.05.2021*	
9.	*Захист*	*16.06.2021*	

Студент-дипломник _____
(підпис)

Керівник проєкту _____
(підпис)

Анотація

В даному бакалаврському дипломному проекті проведено аналіз існуючих рішень систем автоматизації процесу збору даних. Зроблено огляд предметної області та встановлено передумови розробки сервісу.

Спроектовано і розроблено систему автоматизації процесу збору даних. Застосунок дозволяє користувачу створювати сучасні конвеєри збору даних, запускати задачі та слідкувати за їх станом виконання через веб сервер.

Всі компоненти сервісу реалізовані мовою програмування Python з використанням технологій Flask, Celery, Apache Airflow.

Annotation

In this bachelor's thesis project the analysis of existing decisions of systems of automation of the process of data collection is carried out. An overview of the subject area and the prerequisites for the development of the service.

A system for automating the data collection process has been designed and developed. The application allows the user to create modern data collection pipelines, run tasks and monitor their progress through a web server.

All components of the service are implemented in the Python programming language using Flask, Celery, Apache Airflow technologies.

ОПИС АЛЬБОМУ

№ з/п	Формат	Позначення	Найменування	Кількість листів	Додаток
	А4		Завдання на дипломний проєкт	2	
	А4	ІАЛЦ.467200.001 ВП	Відомість проєкту	2	
	А4	ІАЛЦ.467200.002 ТЗ	Технічне завдання	4	
	А4	ІАЛЦ.467200.003 ПЗ	Пояснювальна записка	67	
	А4	ІАЛЦ.467200.006 Д3	Функціональна схема	1	
	А4	ІАЛЦ.467200.005 Д2	Структурна схема	1	
	А4	ІАЛЦ.467200.004 Д1	Принципова схема	1	
	А4	ІАЛЦ.467200.007 Д4	Лістинг програми	10	

Зм.	Арк.	№ докум.	Підпис	Дата	*ІАЛЦ.467200.001 ОА*		
Розроб.		Киба О. Д.			*Система автоматизації процесу збору даних* Опис альбому	Лит.	Арк. 1 / Аркушів 1
Перев.		Алещенко О.В.					
Реценз.						НТУУ «КПІ», ФІОТ, ІВ-72	
Н. Контр.		Сімоненко В. П.					
Затверд.		Стіренко С. Г.					

ТЕХНІЧНЕ ЗАВДАННЯ

до дипломної роботи
освітньо-кваліфікаційного рівня бакалавр

на тему: «Система автоматизації процесу збору даних»

Київ – 2021

ЗМІСТ

1. НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання розповсюджується на розробку програмного забезпечення для потреб бізнесу. Область застосування: використання для управління процесом збору даних.

2. ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки є завдання на виконання роботи кваліфікаційно-освітнього рівня «бакалавр комп'ютерної інженерії», затверджене кафедрою обчислювальної техніки Національного технічного Університету України «Київський Політехнічний інститут ім. Ігоря Сікорського».

3. МЕТА І ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою даного проєкту є розробка системи для автоматизації процесу збору даних та тестування коректності роботи сервісу.

4. ДЖЕРЕЛА РОЗРОБКИ

Джерелом розробки є науково-технічна література з теорії і практики розробки вебсервісів, проєктування програмного забезпечення, наукові статті, публікації в Інтернеті з даних питань.

5. ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до продукту, що розробляється

1. Можливість виконання завдань збору даних “на вимогу”
2. Можливість інтеграції з сучасним інструментами побудови складних конвеєрів обробки даних
3. Надання користувачу інструментів відслідковування стану завдань

4. Можливість горизонтального масштабування
5. Виконання процесу збору даних у фоновому режимі

5.2. Вимоги до програмного забезпечення

1. Встановлений Python версії 3.6 та вище
2. Операційна система на базі ядра Linux

5.3. Вимоги до апаратної частини

1. Комп’ютер на базі процесору Intel Core i5 і вище
2. Оперативної пам’яті не менше 4 Гбайт
3. Вільного місця на жорсткому диску не менше 5 Гбайт
4. Стабільне підключення до Інтернету

6. ЕТАПИ РОЗРОБКИ

	Дата
Затвердження теми роботи	15.12.2020
Аналіз завдання та огляд існуючих рішень	15.03.2021-5.04.2021
Розробка архітектури та загальної структури системи	5.04.2021-10.04.2021
Програмна реалізація системи	10.04.2021-1.05.2021
Доопрацювання і налагодження системи	1.05.2021-6.05.2021
Оформлення пояснювальної записки	2.05.2021-20.05.2021
Передзахист та проходження нормативного контролю	25.05.2021
Захист дипломного проєкту	16.06.2021

Пояснювальна записка
до дипломного проєкту
на тему: «Система автоматизації процесу збору даних»

Київ – 2021

ЗМІСТ

Перелік скорочень

SQL	мова структурованих запитів (англ. Structured Query Language)
AMQP	розширений протокол черги повідомлень (англ. Advanced Message Queuing Protocol)
UI	інтерфейс користувача (англ. User Interface)
DAG	спрямований ациклічний граф (англ. Directed Acyclic Graph)
REST	передача репрезентативного стану (англ. Representational State Transfer)
API	прикладний програмний інтерфейс (англ. Application Programming Interface)
ATF	фреймворк асинхроних завдань (англ. Asynchronous Task Framework)
СУБД	система управління базами даних
RPC	виклик віддаленої процедури (англ. Remote Procedure Call)
ETL	витяг, перетворення та завантаження (англ. Extract, Transform, Load)
HTTP	протокол передачі гіпертекста (англ. HyperText Transfer Protocol)
JSON	запис об'єктів JavaScript (англ. JavaScript Object Notation)
BSON	бінарний запис об'єктів JavaScript (англ. Binary JavaScript Object Notation)
IoC	інверсія управління (англ. Inversion of Control)
DI	впровадження залежностей (англ. Dependency Injection)
ELK	Elasticsearch, Logstash, Kibana

ACID	атомарність, консистенція, ізоляція, довговічність (англ. Atomicity, Consistency, Isolation, Durability)
ORM	об’єктно-реляційне відображення (англ. Object-relational mapping)
DDD	предметно-орієнтоване проектування (англ. Domain Driven Design)
DIP	принцип інверсії залежностей (англ. Dependency Inversion Principle)
DTO	об'єкт передачі даних (англ. Data Transfer Object)
DAO	об’єкт доступу до даних (англ. Data Access Object)
UoW	одиниця роботи (англ. Unit of Work)
CSV	Значення розділені комою (англ. Comma-Separated Values)

ВСТУП

У сучасному світі розробникам програмного забезпечення все частіше доводиться розв’язувати задачі, пов’язані з обробкою та аналізом даних. Ще 10 років тому дані більшості проектів могли поміститися на жорсткому диску одного комп’ютера в реляційній СУБД. А задачі з вилучення та обробки даних зводилися до написання коректного SQL (Structured query language) запиту.

З приходом масової «діджиталізації», бурхливого розвитку IoT (Internet of Things) та штучного інтелекту, збільшенням кількості мобільних телефонів та гаджетів, обсяг генеруємих та використовуємих даних виріс в десятки тисяч разів. Аналізувати такий потік інформації вручну, а тим більше, витягувати корисні для бізнесу або науки дані, практично неможливо. Старі інструменти показали свою неефективність у роботі з великими об’ємами даних. З приходом великих даних з'явилися нові професії, такі як: Data Scientist / Analyst, Data Engineer. У цих спеціалізаціях з’явилися свої підходи та інструменти для вирішення задач обробки даних.

Об’єктом дипломної роботи є система автоматизації збору даних. Вибір об’єкту для створення зумовлений необхідністю організації екосистеми виконання задач збору, обробки та завантаження даних для потреб сучасного бізнесу.

Метою роботи є: огляд технологій, необхідних для створення ефективної системи автоматизації збору даних, аналіз переваг та недоліків аналогів, виконання розробки власного сервісу. Дипломна робота складається зі вступу, двох розділів та висновків до проекту. У першому розділі детально оглянуто приклади вже існуючих систем автоматизації збору даних, досвід інженерів багатьох сучасних компанії при рішені ETL задач. У другому розділі були розглянуті технології для імплементації системи та розроблена високорівнева архітектура сервісу. В третьому розділі був описаний процес розробки програмного забезпечення та обґрунтовані рішення прийняті під час нього.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1. Опис проблематики

Багато програмістів, які працюють в компаніях, що займаються аналізом даних, стикаються з проблемою їх збору та очисткою для аналізу. Інженери створюють власні “скріпти” для виконання цих цілей. Завдання даної роботи полягає в створенні системи автоматизації збору даних, яка б надала уніфікований інтерфейс для виклику процедур збору та обробки даних. Головною функціональністю цього сервісу має бути збір даних “на вимогу”. Система має виконувати типові кроки збору, обробки та завантаження. Існують і інші види систем збору даних, наприклад, пошукові роботи, але в цій роботі розглядається саме система, яка виконує збір даних з доступних для користувача джерел. Також важливою вимогою до сервісу є доступність для користувача, що є гарантією прийняття в обробку запиту на операцію за деякий час. Для виконання цієї вимоги система має бути спроектований таким чином, щоб мати можливість горизонтального масштабування при потребі.

1.2. Огляд існуючих рішень

Починаючи аналіз, необхідно розглянути досвід інших інженерів, які вирішували цю проблему. Таким чином відбудеться систематизування бази знань, яке дозволить знайти оптимальні рішення для конкретних типів задач обробки даних.

1.2.1 Celery

Завдання збору даних, які не потребують складних конвеєрів обробки, але які мають виконуватися у фоновому режимі, можуть бути автоматизовані, використовуючи черги задач. Яскравим представником даної системи є Celery. Celery - це черга асинхронних задач для мови програмування Python. Celery написаний на Python, але протокол може бути реалізований будь-якою мовою. Окрім Python, існують node-celery для Node.js, а також PHP-клієнт.

.Асинхронні черги завдань - це інструмент, що дозволяє запускати частини програмного забезпечення в окремому контексті виконання (машини, процеси, потоки, цикли подій і т.д.) [1]. Він часто використовується у вебархітектурі як спосіб делегування довготривалих завдань для зменшення часу відповіді на запити.

Найчастіше черги асинхронних задач використовують для досягнення прийнятної швидкості відповіді. За допомогою черг асинхронних задач відбувається делегація тривалих завдань процесора. Але найпопулярнішим варіантом використання є виконання зовнішніх викликів. Кожного разу, коли система залежить від зовнішніх служб, стає важко передбачити та контролювати, скільки часу потрібно для виконання. Буває, що задача ніколи не буде готова, оскільки зовнішня система може вийти з ладу або зламатися. Тому черги асинхронних задач підходять для збору даних із зовнішніх сервісів, які не є підконтрольними розробникам.

Ще одним цікавим способом використанням черг асинхронних задач є підготовка та збереження значень результатів. Згідно доповіді на конференції PyCon 2013, саме таким чином розробники Instagram формують головну сторінку свого додатку. Розглянемо цей приклад детальніше [2].

Головна сторінка Instagram складається з останніх десяти фотографій, які виклали люди, на яких підписаний користувач. У своїй доповіді Рік Бренсон розповідає про два підходи, завдяки яким можна отримати десять останніх фотографій: наївний (від якого відмовився в Instagram) та Fanout-on-Write.

При наївному способі потрібно виконати SQL запит (рис 1.1). У цьому випадку при роботі з великим обсягом даних не існує гарантії, що запит буде виконуватись достатньо швидко. Планувальник запитів збирає всі акаунти, на які підписаний користувач, збирає всі фотографії підписок, сортує їх по часу створення та обмежує вибірку до перших 10 фотографій.

```
SELECT * FROM photos
  WHERE author_id IN
    (SELECT target_id FROM following
       WHERE source_id = %(user_id)d)
  ORDER BY creation_time DESC
  LIMIT 10;
```

Рис. 1.1. — SQL запит, що виконувався в Instagram при наївному способі отримання головної сторінки.

Для того, щоб пришвидшити читання головної сторінки, розробники Instagram вирішили зберігати передоброблені списки ідентифікаторів медіа для кожного користувача в ключ-значення базі даних Redis. Ці списки оновлюються за допомогою черг асинхронних задач Celery, які при додаванні нової фотографії користувачем (ініціатор збору даних), обробляють списки медіа його підписників. Такий спосіб автоматизації обробки даних в Instagram назвали Fanout-on-Write.

Архітектура Celery (рис. 1.2) складається з трьох частин: посередник повідомлень, блок виконання завдання, сховище результатів завдань [3].

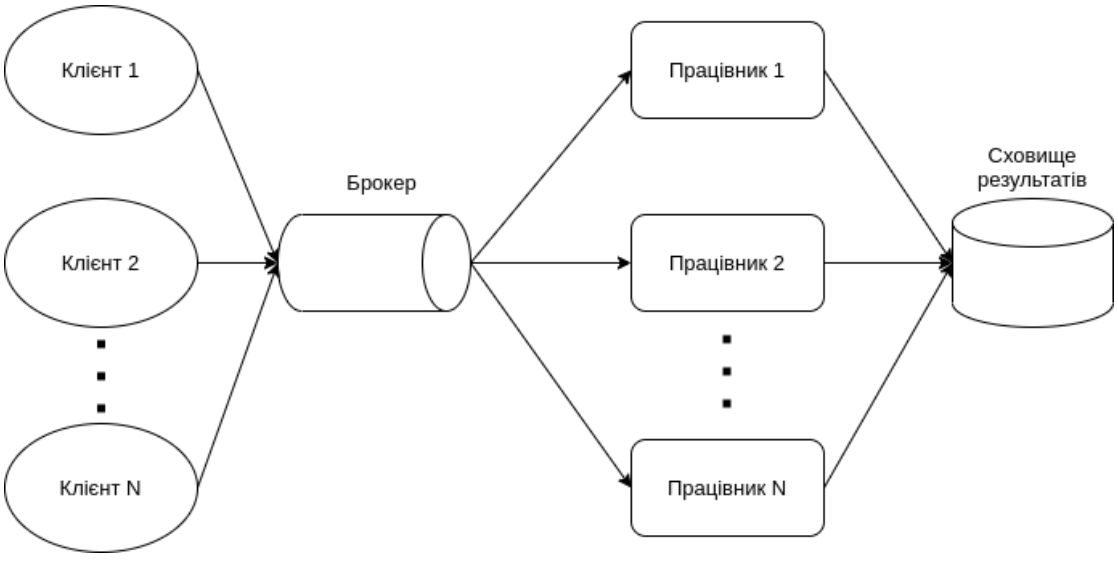


Рис. 1.2. Схема архітектури Celery.

Безпосередньо Celery не надає послуги обміну повідомленнями, але його можна легко інтегрувати з проміжним програмним забезпеченням для обміну повідомленнями, що надається третіми сторонами. Найпопулярнішими з них є RabbitMQ, Redis, Amazon SQS, але Celery також підтримує MongoDB, SQL бази даних, Apache Zookeeper.

Робітник - це блок виконання завдань, що надається Celery, і вони працюють одночасно у розподілених системних вузлах.

Сховище результатів використовується для зберігання результатів завдань, які виконують блоки виконання завдань. Celery підтримує зберігання результатів завдань різними способами, включаючи AMQP, Redis тощо.

Celery не має власного Web UI, але його можна підключити за допомогою плагіна Flower (рис 1.3) [4].

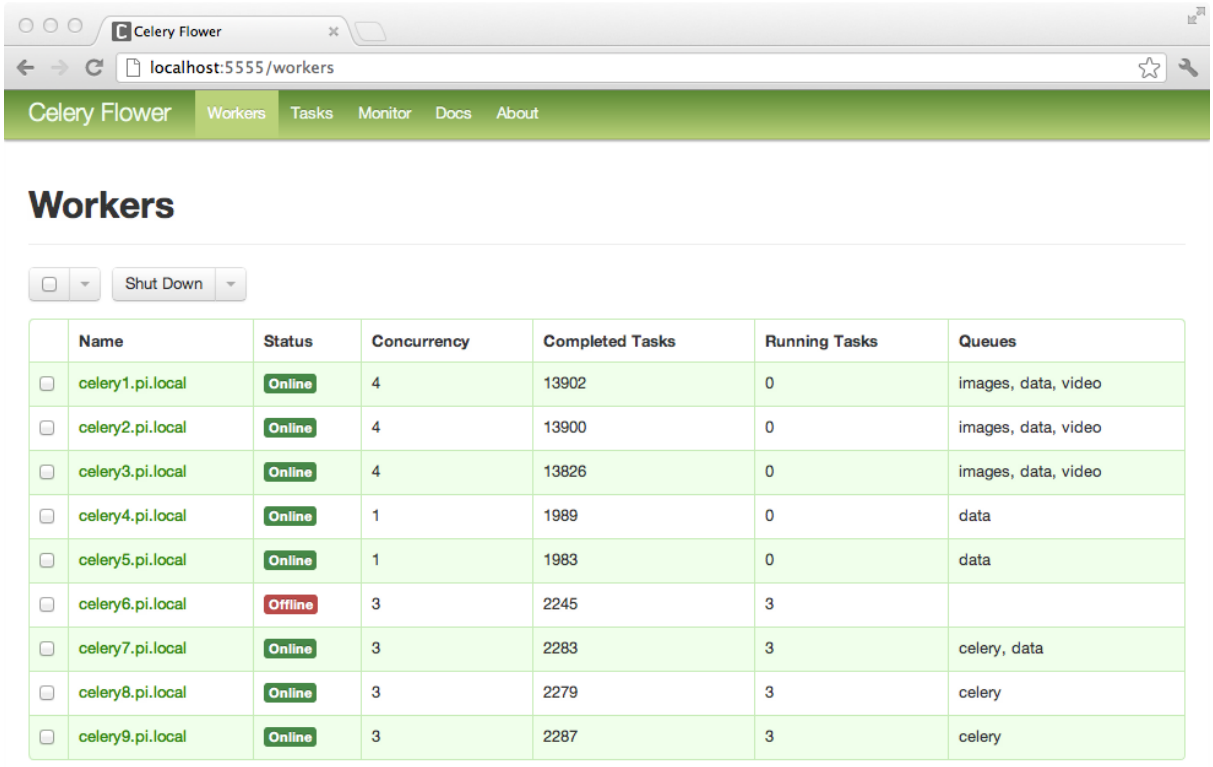


Рис. 1.3. Flower Web UI.

Головною причиною, чому в таких великих компаніях як Instagram використовують Celery, є популярність в спільноті розробників Python, легкість

підтримки та налаштування, простота використання. Цей інструмент має багато важливих для систем автоматизації функцій: відслідковування, але не збереження станів задач, повторне їх виконання при виявлені помилок, базові методи для групування та створення ланцюгів з них, виконання задач за розкладом.

1.2.2. Luigi

Luigi - це бібліотека Python з відкритим кодом, розроблене Spotify, який допомагає створювати складні конвеєри завдань. Бібліотека призначена для вирішення всіх конвеєрних проблем, пов’язаних з тривалими процесами. Вона виконує вирішення залежностей одних задач від інших, управляє робочим процесом, надає візуалізацію, обробку збоїв, інтеграцію командного рядка та багато іншого.

Структура конвеєру в Luigi (рис 1.4) нагадує структуру графу [5]. Він містить вузли, де інформація обробляється, і ребра, що з’єднують вузли, передаючи інформацію наступному вузлу.

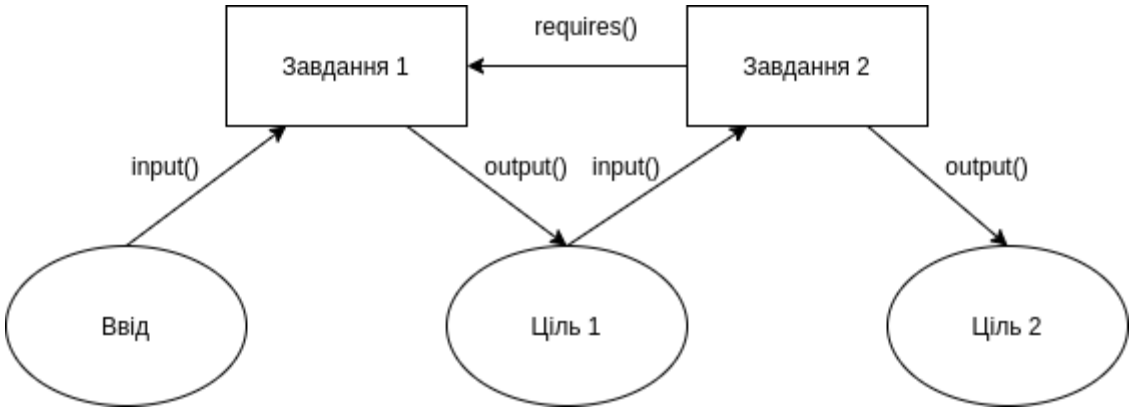


Рис. 1.4 Структура конвеєра в Luigi.

Luigi складається з двох програмних блоків - це завдання та цілі. Ціль - це файл, який зазвичай є результатом виконання завдання, в свою чергу. завдання - це одиниця виконання, яка робить задачі та споживає цілі, генеровані іншими

завданнями. Завдання виконує свою роботу і в результаті генерує ціль, друге завдання бере цільовий файл як вхід, виконує деякі операції та виводить другий цільовий файл і так далі.

Кожне завдання описується як клас в мові програмування Python, похідний від luigi.Task. У новому класі міститься принаймні один або всі з наступних методів:

- requires();
- run();
- output().

Метод **require**() - це перший метод, який виконується, якщо він присутній. Він містить усі екземпляри задач, які будуть виконані до поточного завдання. Найчастіше цей метод здійснює виклик іншого завдання, дозволяючи коду рухатись назад на початок конвеєра. Якщо цілі - це те, що з'єднує завдання з наступним, то require() - це те, що з'єднує поточне завдання з попереднім. Цілі переміщують код у кінець, require - на початок.

Лише коли метод require переконується, що Task може починати своє виконання, запускається другий метод **run**(). У цьому методі міститься код самого завдання. Це може бути що завгодно, виклик іншого методу, запуск скрипта тощо.

Метод **output**() повертає один або кілька об'єктів цілі. Хоча рекомендується, щоб будь-яке завдання повертало лише одну ціль у вихідний файл.

Програма починається з останнього завдання, вона перевіряє, чи можна її виконати, чи перед цим потрібно виконати іншу задачу. Якщо це так, він рухається вгору по конвеєру, поки не знайде вузол, який задовольняє всі вимоги, і лише тоді він починає запускати вже відвідані вузли. Спочатку ця методологія може здатися неінтуїтивною, але насправді вона допомагає уникнути запуску вже виконаних вузлів, виконаних після появи помилки. Такий механізм допомагає ефективно здійснювати повторне виконання ланцюгів завдань при

невдалому запуску до цього.

На відміну від Celery, даний інструмент надає вбудований UI (рис 1.5).

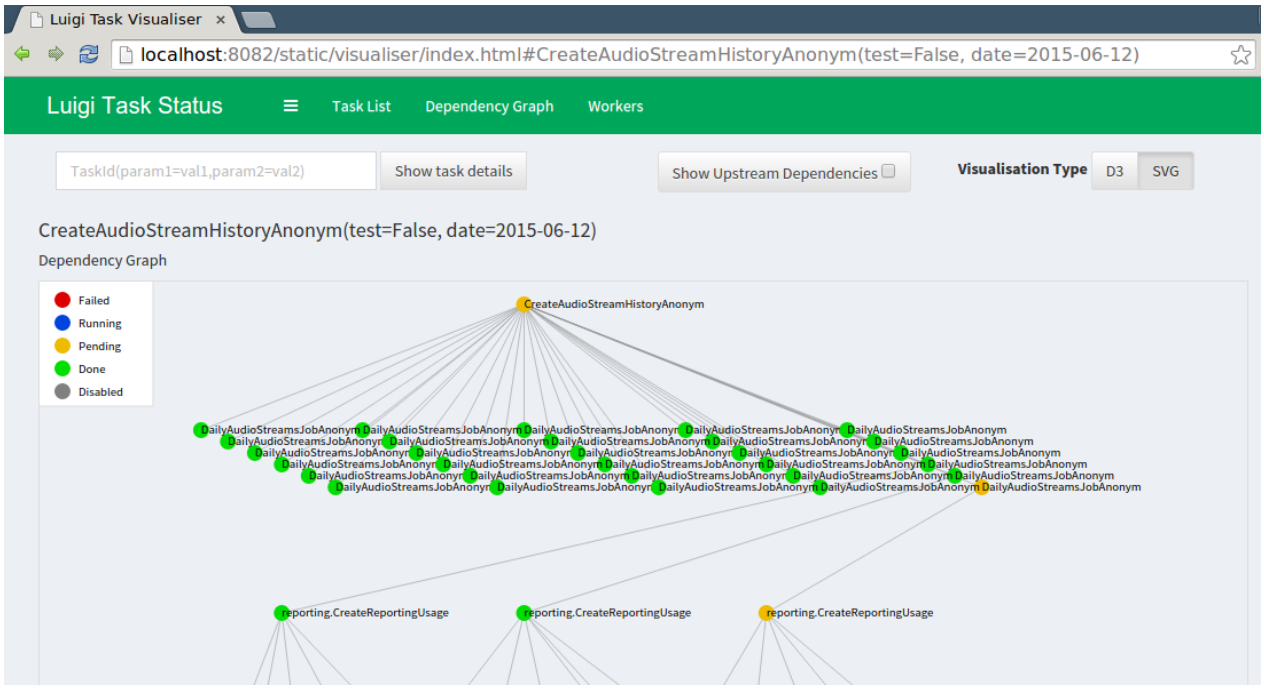


Рис. 1.5 Вбудований Luigi UI.

Luigi не має власного інструменту для організації періодичних завдань, тому прикладним розробникам доводиться вирішувати цю проблему самостійно, використовуючи cron.

Архітектура даної технології дозволяє ефективно будувати прості конвеєри без розгалужень та групувань. Ці операції можна зробити, використовуючи клас luigi.WrapperTask із бібліотеки, але тоді такий код буде важче підтримувати через його заплутаність та контр інтуітивність. Щодо паралелізму, то він повністю делегується бібліотеці. Єдине, що можна зробити - це вказати кількість робочих процесів. Через те, що цілями в такій системі є файли, її дуже важко горизонтально масштабувати, що є вагомим мінусом серед інших систем автоматизації процесів збору даних.

1.2.3. Apache Airflow

Airflow був розроблений інженерами компанії Airbnb в 2014 році, а згодом вихідний код фреймворку був викладений в мережу інтернет. У 2016 році він

долучився до програми інкубації фонду програм Apache [6].

У Airflow конвеєр даних визначається як набір завдань із спрямованими залежностями, в основному спрямований ациклічний графік (DAG). Кожен вузол на графі є завданням, а ребра визначають залежності серед завдань.

У теорії графів орієнтований ациклічний граф - це орграф, в якому відсутні спрямовані цикли, але можуть бути «паралельні» шляху, що виходять з одного вузла і різними шляхами приходять в кінцевий вузол [7]. Спрямований ациклічний граф є узагальненням дерева (точніше, їх об'єднання - ліси). Більшість менеджерів робочих процесів вимагають, щоб графи були ациклічні, що означає, що вони не можуть містити цикли.

У Airflow DAG - це сукупність усіх завдань, які створюють один конвеєр, організований таким чином, щоб відображати їх взаємозв'язки та залежності [8]. DAG складається з операторів та залежностей між ними. Оператори можуть виконувати будь-які завдання в будь-якій технології.

Орієнтовані ациклічні графи описуються мовою програмування Python. Airflow дозволяє описувати графи не тільки у декларативному, а й у імперативному стилі, що надає можливість динамічно генерувати DAG-и. Важливим є той факт, що це фреймворк з відкритим кодом, в якому наявна велика колекція операторів та хуків. Така гнучкість допомогла Airflow бути одним з найпопулярніших рішень для автоматизації процесів збору та обробки даних.

Кластер Airflow має ряд складових (рис. 1.6), які працюють разом: вебсервер, планувальник, сховище метаданих, виконавець та один або кілька працівників [9].

Вебсервер - це інтерфейс фреймворку, за допомогою якого можна отримати огляд загального стану різних спрямованих ациклічних графіків (DAG), а також візуалізувати різні компоненти і стани кожного DAG. Вебсервер також надає можливість керувати користувачами і ролями, запускати конвеєри та слідкувати

за станом виконання DAG-ів через REST API.

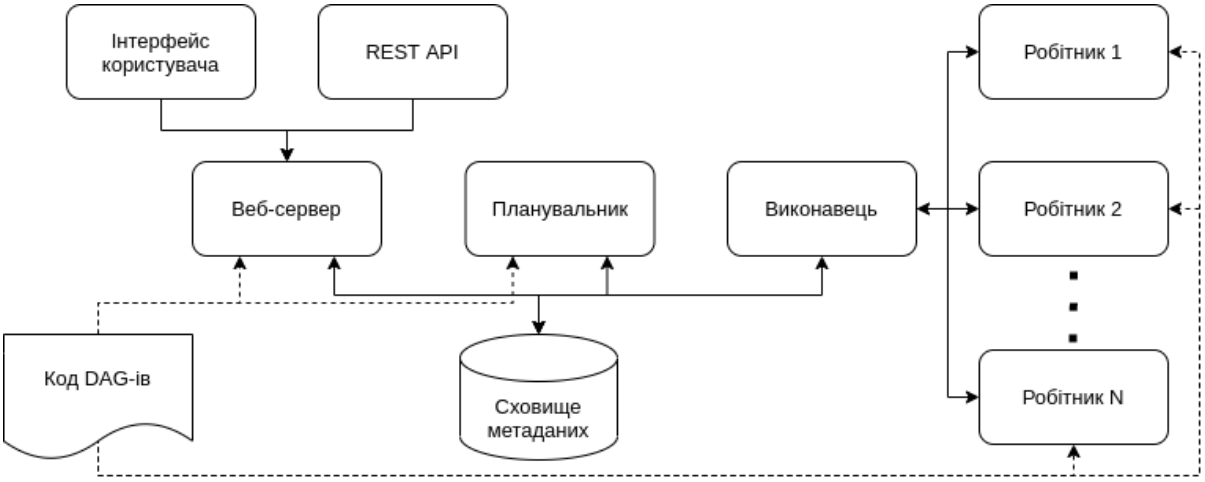


Рис. 1.6 Складові Apache Airflow.

Вебсервер Airflow має добре обладнаний вбудований користувальницький інтерфейс, (рис. 1.7) який забезпечує контроль над кожним конвеєром. Інтерфейс користувача також має розділ для візуалізації потоку виконання DAG, деревоподібний вигляд усіх останніх запусків. Можна також переглядати код DAG, час, за який виконується кожне завдання для кожного запуску, журнали, щоб допомогти налагодити запуски DAG.

Вебсервер Airflow також надає набір REST API, які можна використовувати для виконання різних завдань: таких як активація DAG, завдань або отримання статусу кожного екземпляра завдання. Інтерфейс вебсервера також пропонує опції для управління різними конфігураціями: такими як змінні, з'єднання та перегляд конфігурації Airflow за замовчуванням у самому інтерфейсі.

Планувальник - це найважливіша частина Airflow, яка організовує та слідкує за виконанням DAG-ів та їх задач, дбаючи про їх взаємозалежність, обмежуючи кількість запусків кожного DAG, щоб один DAG не перевантажував усю систему. Планувальник - це багатопотоковий процес Python, який перевіряє та аналізує весь код, що знаходиться в папці виділеній для описання графів. На основі конфігурації кожен DAG отримує ряд процесів або пулів, на яких він

може працювати.

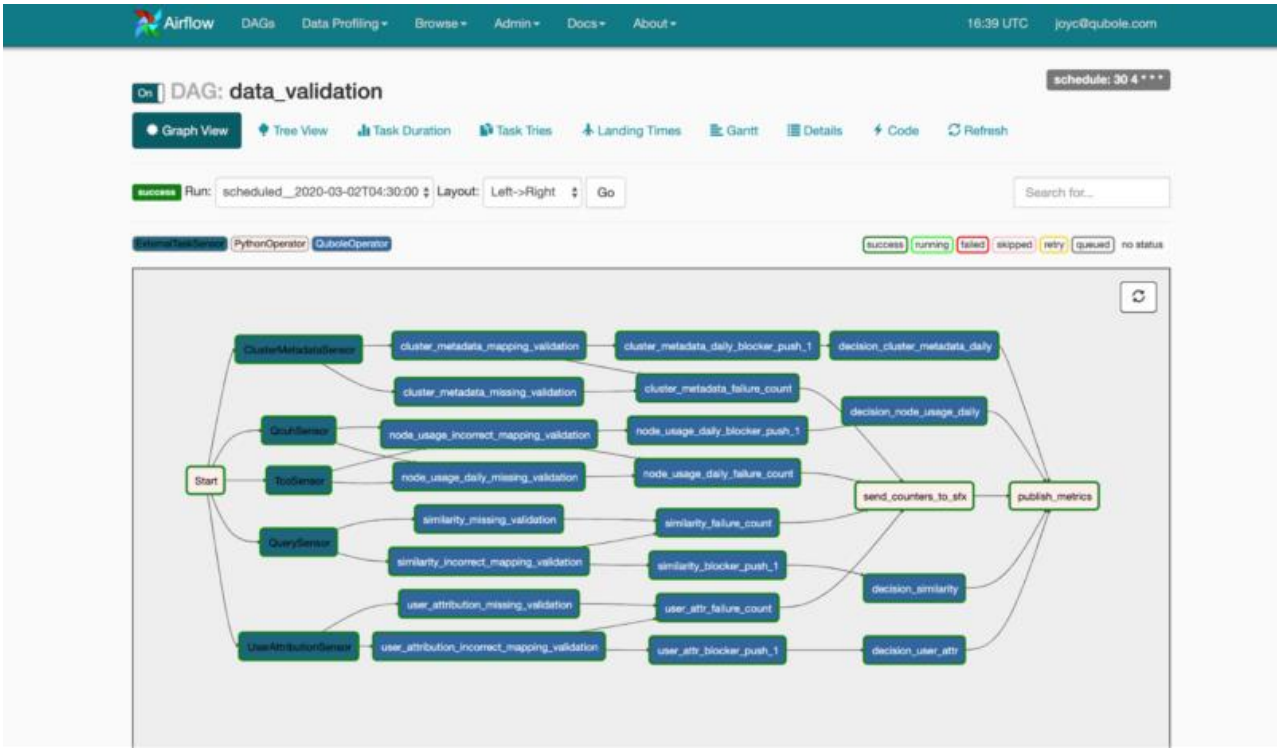


Рис. 1.7 Apache Airflow UI.

Планувальник вирішує, який DAG повинен запускатися на якому пулі, далі він делегує завдання виконавцю. Планувальник стежить за виконанням задачі та оновлює всі переходи стану в базі даних метаданих. За необхідності саме планувальник ініціює повторну спробу виконання.

Виконавець в Apache Airflow - це компонент, який виконує завдання. У Airflow наявні різні типи виконавців, і будь-який з них можна вибрати за допомогою файлу конфігурації на основі вимог до паралельної обробки. Існують такі види [10]:

- SequentialExecutor;
- LocalExecutor;
- CeleryExecutor;
- KubernetesExecutor;

SequentialExecutor може одночасно запускати лише один екземпляр

завдання. Такий тип виконавця підходить лише для налагодження або тестування конвеєрів. Це також єдиний виконавець, який можна використовувати SQLite як сховище метаданих, оскільки SQLite не підтримує кілька одночасних з'єднань**.**

LocalExecutor виконує завдання шляхом розподілення завдань між процесами локальної машини. В ідеалі кількість завдань LocalExecutor необмежена, якщо вказати паралелізм рівним 0. Але кількість завдань тоді обмежується конфігурацією машини, яка використовується для запуску завдань, і об’ємом доступної пам’яті та кількістю ядер центрального процесора. Можна стверджувати, що SequentialExecutor є LocalExecutor з обмеженим паралелізмом лише одного працівника. Цей тип виконавця підходить лише для систем з малим навантаженням, для більшої паралельності нам потрібно мати можливість розподіляти завдання між кількома виконавцями, що працюють на різних машинах.

CeleryExecutor заснований на черзі асинхронних задач для мови програмування Python - Celery, яка широко використовується для обробки асинхронних завдань. У випадку з CeleryExecutor планувальник додає всі завдання до черги, яку ми налаштовуємо. З черги працівник Celery бере завдання і запускає його. Після завершення виконання працівник повідомляє про стан завдання в базі даних. Планувальник опитує базу даних та дізнається стан задач, а потім запускає наступний набір завдань.

KubernetesExecutor надає спосіб запуску завдань Airflow, використовуючи оркестратор контейнерів Kubernetes. Kubernetes запускає новий Pod для кожного завдання. В термінах Kubernetes Pod - це група з одного або кількох контейнерів та ресурси, спільні для них [11]. Поки Kubernetes піклується про виконання завдань у Pod-ах, планувальник опитує статуси завдань у системи оркестрації контейнерів. За допомогою KubernetesExecutor кожне завдання обробляється в новому середовищі виконання всередині кластера Kubernetes, що покращує

ізоляцію завдань. З таким підходом стає легше керувати залежностями за допомогою Docker. Також Kubernetes надає зручні інструменти для горизонтального масштабування (наприклад, збільшення або зменшенн кількості працівників).

Airflow підтримує різноманітні бази даних для свого сховища метаданих. Ця база даних зберігає метадані про DAG-и, їх запуски та інші конфігурації Airflow: такі як користувачі, ролі та з'єднання. Вебсервер показує стани DAG-ів та їх запуски з бази даних. Планувальник також оновлює інформацію в цій базі метаданих. Airflow використовує SQLAlchemy як ORM (Object Relational Mapping) для підключення до сховища метаданих. Це означає, що будь-яку базу даних, що підтримує SQLAlchemy, можна використовувати для зберігання всіх метаданих Airflow.

Airflow відрізняється від інших систем автоматизації та обробки даних своєю функціональною повнотою: цей інструмент містить довгий і постійно зростаючий перелік операторів і хуків, він має легко масштабуєму архітектуру, простий та зрозумілий у використанні інтерфейс для управління робочими процесами.

1.2.4. Asynchronous Task Framework

ATF (Asynchronous Task Framework) був розроблений інженерами компанії Dropbox для виконання асинхронних задач збору та обробки даних. Перед розробниками постала задача створення системи, головна функціональність якої є виконання завдань “на замовлення”. ATF мав би використовуватися різними інженерними групами компанії. Треба було розробити таку архітектуру фреймворку, що дозволила б інтегрувати вже розроблене програмного забезпечення інших команд. На той час розробники ATF не змогли знайти проектів з відкритим кодом чи рішень, які б їх задовольняли, тому в Dropbox створили власну систему.

ATF повинен був не тільки ефективно працювати з самого початку, але й легко

масштабуватися. Цей проект задумувався як фундаментальний будівельний елемент інфраструктури Dropbox. З самого початку йому потрібно було б обробляти 10 000 асинхронних завдань в секунду і бути сконструйованим для майбутнього зростання. Було щонайменше два десятки інженерних команд, які хотіли б використовувати його для абсолютно різних частин кодової бази Dropbox. Зараз коли ATF розгорнуто, він виконує 9000 асинхронних завдань на секунду та використовується 28 інженерними групами.

Планування асинхронних завдань на вимогу є критично важливою функцією для Dropbox. Async Task Framework - це інфраструктурна система, яка підтримує цю можливість за допомогою архітектури на основі зворотного виклику. ATF дозволяє розробникам визначати зворотні виклики та планувати завдання, які виконуються щодо цих заздалегідь визначених зворотних викликів.

В ATF використовують такі сутності для описання завдань: лямбда, задача, колекція та пріоритет. Лямбда - зворотний виклик, що реалізує бізнес-логіку. Завдання - одиниця виконання лямбди. Кожна асинхронна задача, запланованe за допомогою ATF, є завданням. Колекція - підмножина завдань, що належать одній лямбда. Таким чином в ATF реалізовані конвеєри. Пріоритет - мітка, що визначає пріоритет виконання завдань.

Головними функції ATF є:

- розклад задач (завдання можуть бути заплановані для негайного виконання або відкладені відповідно до випадку використання);
- пріоритезація задач (у кожного завдання має бути пріоритет, завдання з вищим пріоритетом повинні виконуватися перед завданнями з нижчим пріоритетом);
- відображення стану задач (користувачі системи можуть відслідковувати статус запланованого завдання).

ATF має чіткі правила, за якими він працює. Система гарантує, що завдання

виконується принаймні один раз після планування. Неможливе одночасне виконання одного і того самого завдання на різних машинах або процесах. 95% завдань починають виконуватись протягом п’яти секунд від запланованого часу виконання. ATF має бути доступним для прийому запитів на планування завдань від будь-якого клієнта.

Для того, щоб ATF міг виконати усі гарантії, які він надає користувачу системи, фреймворк робить обмеження для бізнес-логіки, яку можна помістити у лямбду.

Лямбда-функції мають бути індемполентними. Це означає, що одне завдання може виконуватися кілька разів у системі ATF. Розробники повинні написати лямбда-функцію таким чином, щоб це не вплинуло на логіку та правильність виконання завдань.

Коректність роботи лямбда-функції має бути незалежною від конкретної машини. Розробники можуть припинити свою роботу через помилку або технічні несправності в будь-який момент часу. Власники лямбда повинні проектувати свої лямбди так, щоб повторні спроби запуску завдань на різних хостах не вплинули на правильність виконання бізнес- логіки.

ATF буде повторювати задачу, поки з ламбда-логіки системі не нададуть сигнал про завершення. Функція може позначити завдання як успішно виконане, припинене або перевірне. Дуже важливо, щоб власники лямбда розробляли свої функції з урахуванням цієї особливості, щоб уникнути нескінченних спроб запуска коду.

ATF складається з наступних компонентів (рис. 1.8):

- фронтенд-сервер;
- сховище завдань;
- споживач сховища;
- черги задач;

- контролер;
- виконавець;
- контролер серцебиття та стану.

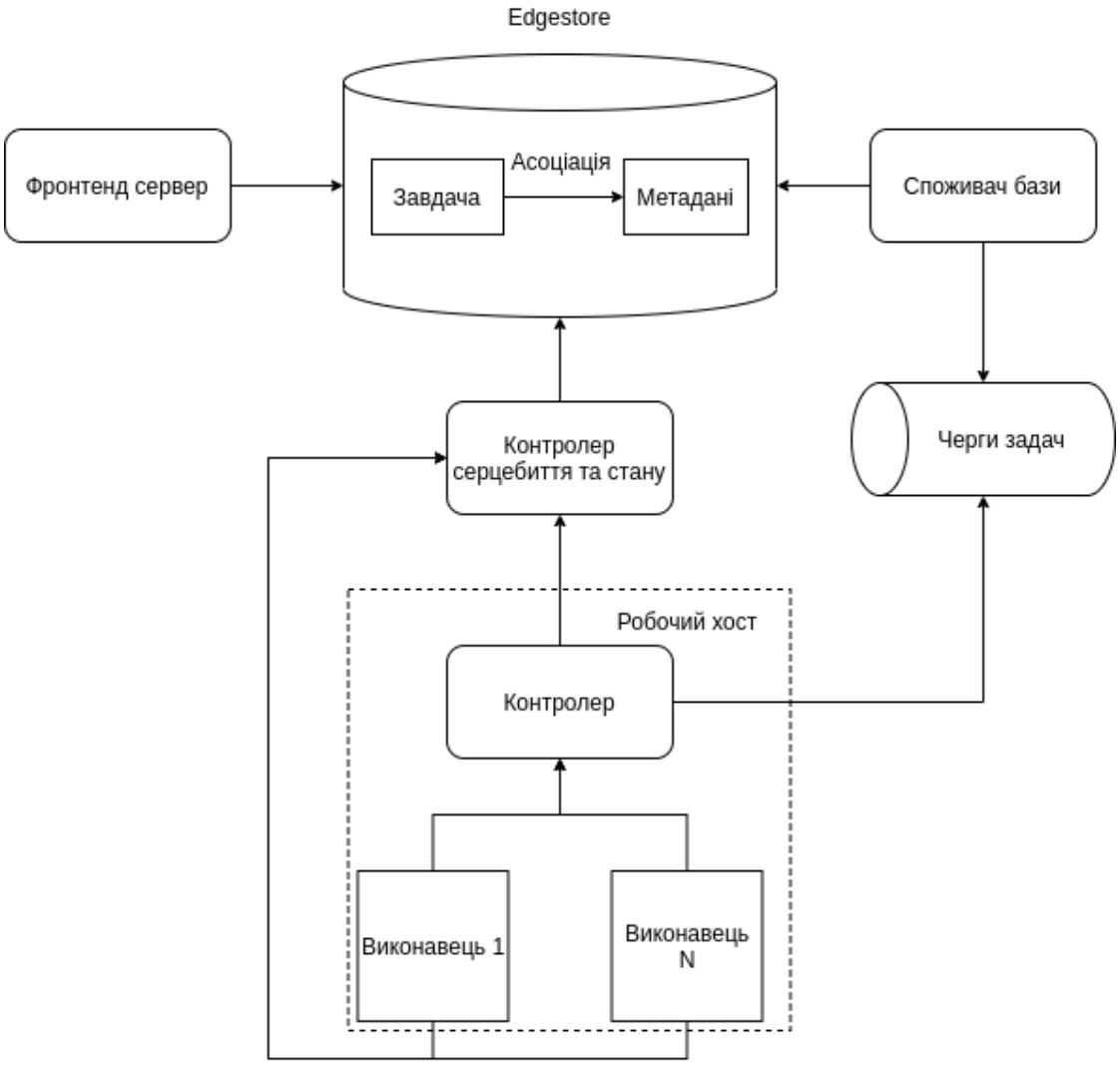


Рис. 1.8 Компоненти ATF.

Фронтенд-сервер - це сервіс, який зберігає задачі через інтерфейс RPC. Сервер ATF використовує протокол gRPC для спілкування з користувачем. gRPC - це сучасна високопродуктивна система віддаленого виклику процедур від компанії Google із відкритим кодом, яка може працювати в будь-якому середовищі [12]. Інтерфейс приймає запити RPC від клієнтів і планує завдання, взаємодіючи із сховищем задач ATF.

Сховищем завдань в ATF може бути будь-яка загальна база даних з можливістю індексованого запиту. У випадку ATF в Dropbox використовують власну внутрішню розробку, яка називається Edgestore (обгортка над MySQL СУБД). [13] Для розуміння роботи ATF буде досить ознайомитися з концепцією асоціацій в Edgestore.

Об'єктами в Edgestore можуть бути компонент або асоціація, кожна з яких може мати визначені користувачем атрибути (аналогічно стовпцям у традиційній таблиці бази даних). Асоціації описують, як різні компоненти співвідносяться між собою: наприклад, співвідношення між користувачами та командами можна описати таким чином: користувачі - UserEntity, команди - TeamEntity, а асоціація між ними - це UserTeamAssoc. При цьому слід розуміти, що Edgestore підтримує індексацію лише за атрибутами асоціацій.

На основі цього дизайну ми маємо два види об’єктів, пов’язаних з ATF, в Edgestore. Асоціація ATF зберігає інформацію про планування, час запуску завдання (вперше або для повторної спроби). ATF зберігає всю інформацію про завдання (стан, аргументи виконання і т.д.). Щоб отримати готові до виконання завдання, споживачу сховища достатньо зробити запит за встановленим інтервалом по часу виконання, що зберігається в асоціації задач.

Споживач сховища - це сервіс, який періодично опитує сховище завдань, щоб знайти задачі, готові до виконання, та передати їх в чергу. Це можуть бути нещодавно готові до виконання завдання або застарілі завдання, які готові до виконання знову. Також споживач опитує завдання, які некоректно завершилися під час попередніх спроб виконання. Таким чином працює механізм повторного виконання в ATF.

ATF використовує Simple Queue Service (SQS) від компанії Amazon для внутрішньої **черги завдань**. Ці черги виконують роль брокера повідомлень між споживачем магазину та контролерами. В ATF для кожної пари лямбда та пріоритет створюється власна черга. Таким чином, пріоритетність в цій системі

реалізується на рівні черг.

В ATF прийнято ділити обчислювальні потужності на робочі хости. Робочі хости - це фізичні хости, призначені для виконання завдання. Кожен робочий хост має один процес **контролера**, відповідальний за опитування завдань з черг у фоновому потоці, та направлення задач в локальні черги цього робочого хоста. Кожна машина обслуговує конкретний набір лямбд, тому контролер опитує лише обмежений набір необхідних черг. Саме на рівні контроллера відбувається пріоритезація, бо він вирішує, яке завдання надати виконавцю, коли той зробить запит до нього згідно алгоритму пріоритезації .

Виконавець - це процес із декількома потоками, відповідальний за фактичне виконання завдання. Кожен робочий хост має один процес контролера та кілька процесів-виконавців. І контролер, і виконавці постійно здійснюють опитування (контролер опитує зовнішні черги, а виконавці - контролер, щоб отримати від нього задачу).

Контролер серцебиття та стану здійснює оновлення статусів задач в сховищі завдань та перевіряє. чи не ввійшла задача у нескінченний цикл.

Життєвий цикл задачі виглядає таким чином:

1. Користувач виконує RPC запит до фронтенд-сервера з інформацією про завдання, включаючи час виконання;
2. Сервер створює компонент та асоціацію завдання в Edgestore;
3. Коли настав час обробити завдання, споживач бази витягує завдання з Edgestore і надсилає його до відповідної черги;
4. Виконавець робить виклик до контролера, який надає виконавцю задачу. Далі контролер робить виклик до контролер серцебиття та стану, щоб оновити стан задачі. І тільки тоді контролер надсилає задачу виконавцю;
5. Виконавець викликає лямбда-функцію для завдання. Під час обробки виконавець робить запити серцебиття, щоб сигналізувати системі про здоровий процес виконання задачі. Після завершення обробки виконавець

надсилає повідомлення про завершення задачі, щоб оновити її статус;

6. Отримавши виклики серцебиття та оновлення статусів, контроллер серцебиття та стану оновлює компоненти та асоціації в Edgestore.

Кожне оновлення стану в життєвому циклі завдання супроводжується оновленням до наступної позначки часу тригера в асоціації завдання. Це гарантує, що споживач бази повторно виконує завдання, якщо до настання наступного моменту часу запуску не зміниться стан завдання. Це допомагає ATF виконувати повторний запуск.

Розробники ATF планують розширити функціональність своєї платформи. В даний час ATF - це система одноразового планування завдань. Надання можливості періодичного виконання завдань було б корисним для розблокування нових можливостей для користувачів даної системи. Єдиною можливістю побудови ланцюгів завдання з використанням ATF є планування завдання, яке потім планує інші завдання під час свого виконання. Хоча це можливо зробити в поточній версії ATF, такий спосіб створення конвеєрів є важко контрольованим та неочевидним для інших розробників.

1.2.5. Elastic Stack

Ще одним цікавим прикладом системи автоматизації збору даних є Elastic Stack. На відміну від попередніх технологій, ця система є орієнтованою на збір журналів. Раніше цей стек назвали ELK (Elasticsearch, Logstash, Kibana), але нещодавно список технологій поповнився інструментом Beats [14].

Разом ці різні компоненти найчастіше використовуються для моніторингу, усунення несправностей та захисту IT-середовищ (хоча для стеку ELK існує набагато більше випадків використання: таких як бізнес-аналітика та вебаналітика). Beats та Logstash дбають про збір та обробку даних, Elasticsearch забезпечує їх зберігання, а Kibana надає користувальницький інтерфейс для їх візуалізації. Цей стек надає користувачам потужну платформу, яка збирає та обробляє дані з декількох джерел, зберігає ці дані в одному централізованому

сховищі, яке може масштабуватися в міру зростання даних, та надає набір інструментів для аналізу та візуалізації даних.

Elasticsearch - головна частина найпопулярнішої на сьогодні платформи аналізу журналів. Elasticsearch в контексті стеку використовується для пошуку збереження та аналізу журналів. Elasticsearch - це пошуковий движок, який базується на Apache Lucene. Він був випущений у 2010 році, має повністю відкритий код і побудований на Java. Elasticsearch класифікується як NoSQL база даних [15]. На відміну від більшості баз даних, Elasticsearch приділяє велику увагу всім видам пошуку. Elasticsearch надає зручний REST API для виконання операції.

Logstash - це інструмент з відкритим кодом, розроблений для обробки потокової передачі великої кількості журналів з декількох джерел [16]. Після включення в Elastic стек він почав відповідати за обробку повідомлень журналів, їх трансформацію, надсилання у пункт призначення для подальшого зберігання. Завдяки великій екосистемі плагінів, Logstash можна використовувати для збору та перетворення широкого спектру різних типів даних. Існує понад 200 різних плагінів для Logstash.

Kibana - це користувальницький інтерфейс (рис 1.9) з повністю відкритим кодом на основі браузера, який можна використовувати для пошуку, аналізу та візуалізації даних, що зберігаються в індексах Elasticsearch (Kibana не може використовуватися разом з іншими базами даних) [17]. Kibana особливо відома та популярна завдяки своїм багатим графічним та візуалізаційним можливостям, які дозволяють користувачам досліджувати великі обсяги даних. Цей інструмент можна встановити на Linux, Windows і MacOS.

Beats - це колекція відправників журналів з відкритим кодом, які діють як агенти, встановлені на різних серверах у вашому середовищі для збору журналів або показників [18]. Написані на мові програмування Go, ці вантажовідправники були спроектовані для того, щоб мати незначний вплив на обчислювальну

машину - вони залишають невеликий слід при встановленні, ефективно використовують ресурси та функціонують без будь-яких залежностей. Дані, що збираються різними beat-ми, різняться: файли журналів у Filebeat, мережеві дані у Packetbeat, системні та службові метрики у Metricbeat, журнали подій Windows у Winlogbeat тощо. Крім beat-ів компанії Elastic, є beat-и, які розроблені спільнотою програмістів.

Рис. 1.9. Графічний інтерфейс Kibana

Різні компоненти в Elastic стеку були розроблені для того, щоб взаємодіяти між собою без зайвої конфігурації. Однак усе залежить саме від того, який результат потрібен розробникам. Для систем з малим навантаженням буде досить класичної конфігурації стеку, що складається з Beats, Logstash, Elasticsearch та Kibana.

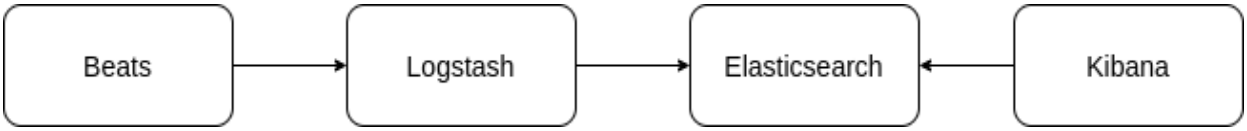


Рис. 1.10 Класична конфігурація Elastic Stack

Однак для обробки більш складних конвеєрів, побудованих для великих обсягів даних, у класичну архітектуру додають компоненти для стійкості (Kafka, RabbitMQ, Redis) та безпеки (Nginx).

Elastic Stack – це дуже цікавий стек технологій, який має купу готових рішень. На відміну від прикладів, що були розглянуті вище, ця система орієнтована не на збір даних в прямому сенсі цього слова. Elastic Stack орієнтовний саме на завантаження та збереження вже зібраних даних для подальшого аналізу. Користуючись цим стеком не можна виконати збір даних на вимогу, не можна зробити складних операцій перетворення.

Таблиця 1.1. Порівняння оглянутих сервісів

	Celery	Luigi	Airflow	ATF	Elastic Stack
Орієнтованість	Автоматизація простих завдань	Автоматизація лінійних конвеєрів	автоматизація DAG-ів	автоматизація завдань “на вимогу”	автоматизація збору журналів
Масштабованість	Наявна	Обмежена архітектурою з цільовими файлами	Наявна	Наявна	Наявна
Контроль стану	Обмежений	Обмежений цільовими файлами	Наявний	Наявний	Відсутній
Повторне виконання	Наявне	Наявне	Наявне	Наявне	Відсутнє
UI	Через розширення	Наявне	Наявне	Відсутнє	Наявне

ВИСНОВКИ ДО РОЗДІЛУ 1

У результаті виконання аналізу предметної області та вивчення досвіду інженерів при побудові існуючих рішень систем автоматизації збору даних можна дійти наступних висновків:

1. Передбачити час виконання збору, обробки та завантаження даних є вкрай важким завданням через великі об’єми та відсутність контролю за джерелами даних. Тому розробники часто проектують свої системи таким чином, щоб ці задачі виконувалися у фоновому режимі та не заважали сервісу приймати заявки на нові завдання.
2. Важливою функціональністю систем автоматизації процесу збору даних є спостереження за станом задач. Така особливість не тільки допомагає користувачу прозоро оцінювати готовність завдань, а й налагодити можливість повторного виконання невдало завершених задач.
3. Кожна з досліджувальних типів систем автоматизації збору даних має свою орієнтованість на вирішення конкретних завдань, тому для майбутньої роботи буде вибір такої архітектури сервісу, яка дозволить використовувати вищезгадані та знайомі багатьом розробникам технології в середині своєї екосистеми.

.

РОЗДІЛ 2
ВИБІР ТА ОБҐРУНТУВАННЯ ЗАСОБІВ РЕАЛІЗАЦІЇ

2.1 Класифікація задач обробки та збору даних

Концептуально кожну задачу по роботі з даними можна поділити на три кроки: збір, обробку та завантаження. Таку концепцію перетворення “сирих” даних називають ETL (Extract, Transform та Load) [19]. Опишемо кожен крок в цій моделі.

З точки зору ETL, архітектуру сховища даних можна розділити на три компоненти:

- джерело даних (база даних, сторонній сервіс, файли);
- проміжна область (тимчасові структури даних для передачі їх у фазу перетворення);
- отримувач даних (сховище, в якому зберігаються результати етапу завантаження).

Початковим етапом в концепції ETL є процедура вивантаження даних із їх джерела. **Виймання даних** - це дуже важливий етап, бо без його коректного виконання неможливі подальші етапи перетворення та завантаження. Залежно від періодичності збору даних, ця задача буде проходити різним чином. Вивантаження даних займає певний час, який називається вікном вивантаження.

Часто виймання даних здійснюється з різних джерел. Кожна система може мати різний формат та структуру даних. Тому ця фаза спрямована на перетворення даних в єдиний формат, очікуваний на наступних етапах.

Також етап збору може передбачати валідацію даних. У цьому випадку відбувається перевірка даних з джерел на відповідність певним правилам та шаблонам. Якщо дані є інвалідні, вони частково або повністю відкидаються. Відхилені дані можуть відправлятися назад у джерело збору для подальшого

аналізу.

Збір даних можна організувати двома способами: вилучення даних за допомогою спеціалізованих програмних засобів; вилучення даних засобами тієї системи, в якій вони зберігаються. Після збору дані поміщаються у проміжну область збереження. На етапі перетворення дані приводяться до виду більш зручного для подальшого аналізу та готуються до розміщення в отримувачі даних.

У процесі перетворення даних в ETL найчастіше виконуються наступні операції:

- проекція (вибір лише певних стовпців для завантаження);
- агрегація;
- переклад (наприклад, якщо вихідна система кодів чоловіків помічається як «1» та жінок — як «2», але вхідна система має коди чоловіків як «Ч» і жінок як «Ж»);
- створення нових даних на основі отриманих (наприклад, розрахунки по формулам);
- перевірка даних (відрізняється від валідації на етапі збору своєю продиктованістю бізнес логікою, а не формою).

Процес завантаження даних полягає в тому, що оброблені дані з проміжної області переміщуються в отримувач даних. При черговому завантаженні в сховище даних переноситься не вся інформація з джерел, а тільки та, яка була змінена протягом проміжного часу, що пройшов з попередньої завантаження. При цьому виділяють два потоки:потік додавання - в сховище даних передається нова, що раніше не існувала інформація; потік доповнення - в сховище даних передається інформація, яка існувала раніше, але була змінена або доповнена.

Для розподілу даних при завантаженні на потоці використовуються метаінформацію даних. Вони фіксують стан даних в деякі моменти часу і

визначають, які з них були змінені або доповнені.

2.2 Обґрунтування високорівневої архітектури системи

Кожна система автоматизації збору даних, яку було розглянуто у розділі 1, має власний тип задач, який найкраще підходить для її використання. У сучасних реаліях для автоматизації завдань збору, обробки та завантаження даних краще підходить концепція організації ланцюгів задач у форму направленого ациклічного графа, чим і зумовлена популярність фреймворку Apache Airflow. Але, керуючись досвідом багатьох компаній, можна стверджувати, що існують такі алгоритми роботи з даними, для яких перетворення у DAG не має ніякого сенсу. Для того, щоб досягти більшої універсальності та покрити ці два типи задач, було вирішено застосувати ATF-подібну високорівневу архітектуру (рис 2.1), яка б делегувала виконання завдань двом групам виконавців: робітникам та конвеєрам.

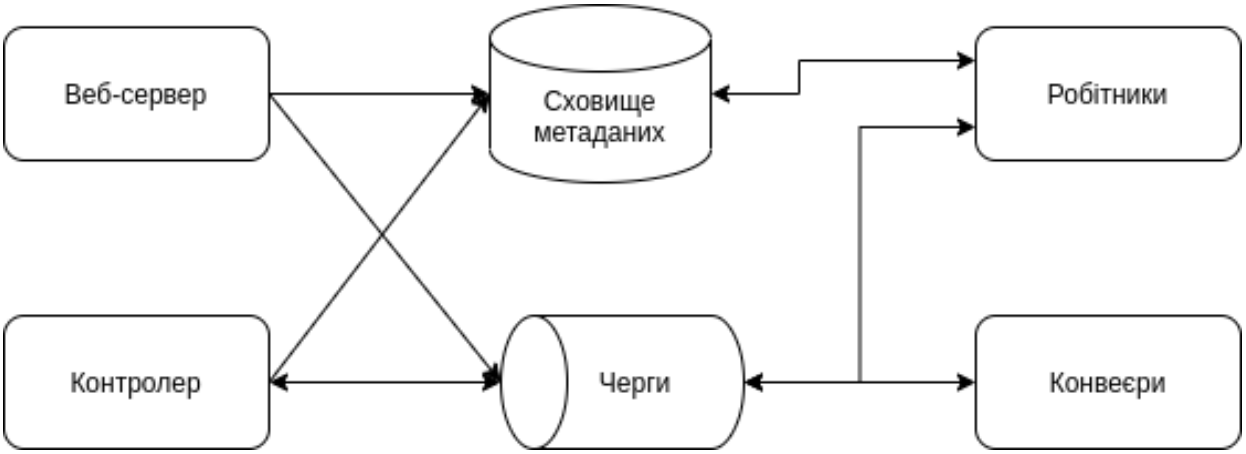


Рис. 2.1 Високорівнева архітектура системи автоматизації процесу збору даних

Веб сервер має відповідати за обслуговування HTTP - запитів. Його головна мета - надавати користувачу зручний та зрозумілий REST API для створення завдань, їх перегляду, відслідковування стану. При створені завдання веб сервер має зберегти його інформацію у сховище та надіслати повідомлення у чергу задач контролера.

У цій архітектурі **контролер** відіграє роль розподільника завдань. Сервіс приймає рішення про те, в яку чергу відправити те чи інше завдання на основі отриманого повідомлення та інформації задачі, яку він може отримати зі сховища при необхідності. Додатковим плюсом такої архітектури є той факт, що за допомогою контролера можна легко інтегрувати внутрішні сервіси, які часто спілкуються між собою саме використовуючи черги задач, коли їм потрібен виклик відкладеної процедури. Наприклад, якщо внутрішньому сервісу потрібно створити задачу, замість того, щоб виконувати HTTP- запит у вебсервер та чекати його відповіді, він може надіслати потрібне повідомлення в чергу контролера.

Така система збору потребує місце, в якому будуть зберігатися дані, пов’язані із задачами. Такий компонент в цій архітектурі - це **сховище метаданих**. Це може бути будь-яка база даних, яка гарантує довговічність. Довговічність - це умова, що незалежно від усіх можливих проблем результати завершених атомарних операцій будуть збережені. Тому така база даних, як Redis, не підходить для того, щоб зберігати в ній інформацію про задачі. Це сховище зберігає дані в оперативній пам'яті та робить знімки на жорсткий диск з якоюсь періодичністю. Це означає, що існує вірогідність виникнення збою саме в проміжках між створенням знімків, що призведе до втрати даних. На роль сховища метаданих підходять усі реляційні та документо орієнтовані СУБД.

Черги задач виконують важливу роль в цій системі автоматизації збору даних. Вони зв’язують всі сервіси між собою, формуючи єдиний механізм. Саме завдяки чергам можливе виконання у фоновому режимі, паралелізація та пріоритезація завдань. До цих черг можуть мати доступ інші внутрішні сервіси для забезпечення викликів відкладених процедур.

Конвеєри - це повністю автономна система із власним WEB UI, сховищем метаданих та чергами. Усе спілкування з іншими компонентами сервісу має відбуватися через чергу контролера заради спрощення слідкування. У загальній

системі автоматизації збору даних конвеєр розглядається як суцільна задача. Під час виконання конвеєр надсилає спеціальні повідомлення про зміну стану задачі в чергу контролера. Повторне виконання забезпечується самою системою конвеєра.

Робітники виконують ті завдання, які не потребують розгляду їх як конвеєра. Це задачі створення й оновлення інформації завдань, передобробка даних у зручний для сервера формат та задачі нотифікації користувача про стан його завдань (надсилання повідомлень із використанням електронної пошти або акаунтів соціальних мереж). Функціональність повторного виконання для робітників реалізується на їх рівні.

Такий сервіс вбирає в себе всі найкращі аспекти з розглянутих раніше систем автоматизації збору даних. Використання ATF-подібної архітектури надає системі гнучкості. Виконавець (конвеєр або робітник) може реалізувати задачу, використовуючи будь-який технологічний стек. Головна умова інтеграції - це підтримка протоколу та інтерфейсу черг. Тим чином, поділ усіх виконавців на робітників та конвеєри надає користувачу системи вибір способу реалізації процесу збору та обробки даних, забезпечує гнучкість та можливість створення оптимального рішення для конкретної задачі.

2.3 Вибір засобів розробки

Аналізуючи високорівневу архітектуру системи, розуміємо, що необхідно обрати стек для веб сервера та контролера, СУБД-сховища метаданих, технологію для черг задач, робітників та конвеєрів.

2.3.1 Мова програмування

Мову програмування слід обирати, зважаючи на інструменти, бібліотеки та фреймворки, якими вона забезпечується. Найпопулярнішою мовою програмування для автоматизації процесів є Python. Python - це об'єктно-орієнтовна високорівнева мова програмування [20]. Синтаксис Python дуже простий та гарно читаємий, саме цим і зумовлена його популярність в

академічному середовищі та серед розробників прикладного програмного забезпечення. Ця мова програмування дозволяє швидко прототипувати сервіси та додатки.

Саме наявність фреймворків та бібліотек для автоматизації процесу збору даних є головним приводом вибору Python. Для компоненту робітника можна вибрати Celery або Faust, для конвеєрів - Apache Airflow, а для вебсервера існує велика кількість технологій: Django, Flask, Tornado, aiohttp тощо.

2.3.2 Сховище метаданих

Головною вимогою до СУБД є довговічність. Розглянемо два підходи до того, в якій формі можна зберігати дані: документо-орієнтований та реляційний.

Найпопулярнішою документо-орієнтованою СУБД є **MongoDB**. MongoDB - це NoSQL база даних, яка використовує документи (схожі на JSON) для управління інформацією.

Замість використання таблиць і рядків, як у реляційних базах даних, MongoDB використовує колекції та документи [21]. Документи складаються з пар ключ-значення, які є основною одиницею даних (рис 2.2).

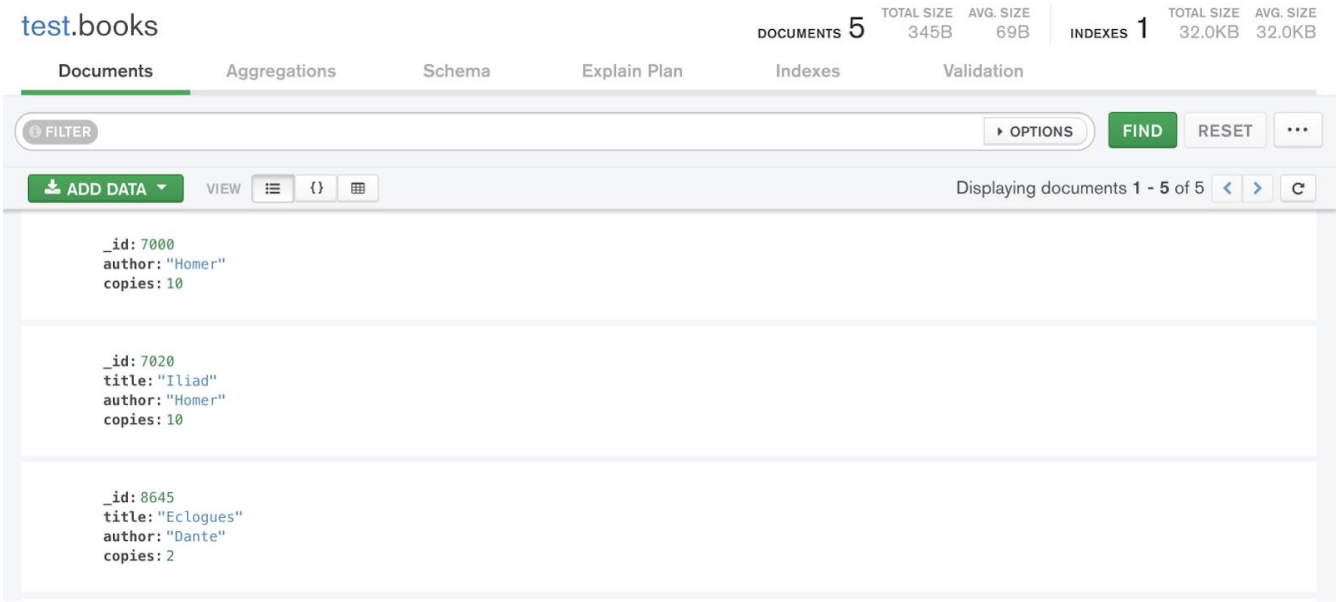


Рис. 2.2 Вигляд колекції MongoDB у програмі Compas

Колекції містять набори документів, що є еквівалентом таблиць. MongoDB зберігає документи у вигляді BSON для оптимізації, що є двійковим форматом представлення простих структур даних і асоціативних масивів (JSON).

Використання BSON дозволяє змінювати схему колекцій без наслідків. Окремий документ є власним екземпляром схеми. З часом схема може бути змінена без наслідків для бази даних. Варто зазначити, що це дуже зручна особливість MongoDB під час розробки, але водночас проблемне місце в момент налагодження програми.

У Python екосистемі найчастіше з реляційних баз даних обирають **PostgreSQL**. PostgreSQL - це система управління реляційними базами даних з відкритим кодом [22]. Postgres використовує SQL для визначення, доступу, модифікації та управління даними. У Postgres наявний власний діалект SQL під назвою pgSQL. Він додає нові типи даних та синтаксичні конструкції. Інші реляційні бази даних мають свої власні діалекти SQL, що призводить до незначних відмінностей між ними.

Оскільки ця база даних має реляційну природу, основна частина інформація зберігається у вигляді таблиць (рис 2.3). На відміну від MongoDB, уся схема даних повинна бути розроблена та налаштована під час її створення. Можна змінити схему після створення, це називається міграцією. Інколи процес міграції може бути складним з різних бізнесових причин.

У реляційних базах даних прийнято зберігати дані в нормалізованому вигляді. Нормалізація включає в себе створення таблиць та встановлення відносин між цими таблицями відповідно до правил, призначених для захисту даних і забезпечення більшої гнучкості бази даних за рахунок виключення надмірності і неузгодженості залежності. Для встановлення відносин між таблицями в PostgreSQL можуть використовувати зовнішні ключі. Зовнішні ключі дозволяють посилатися з рядка однієї таблиці на рядок іншої, використовуючи унікальне поле рядка, на який відбувається послання.

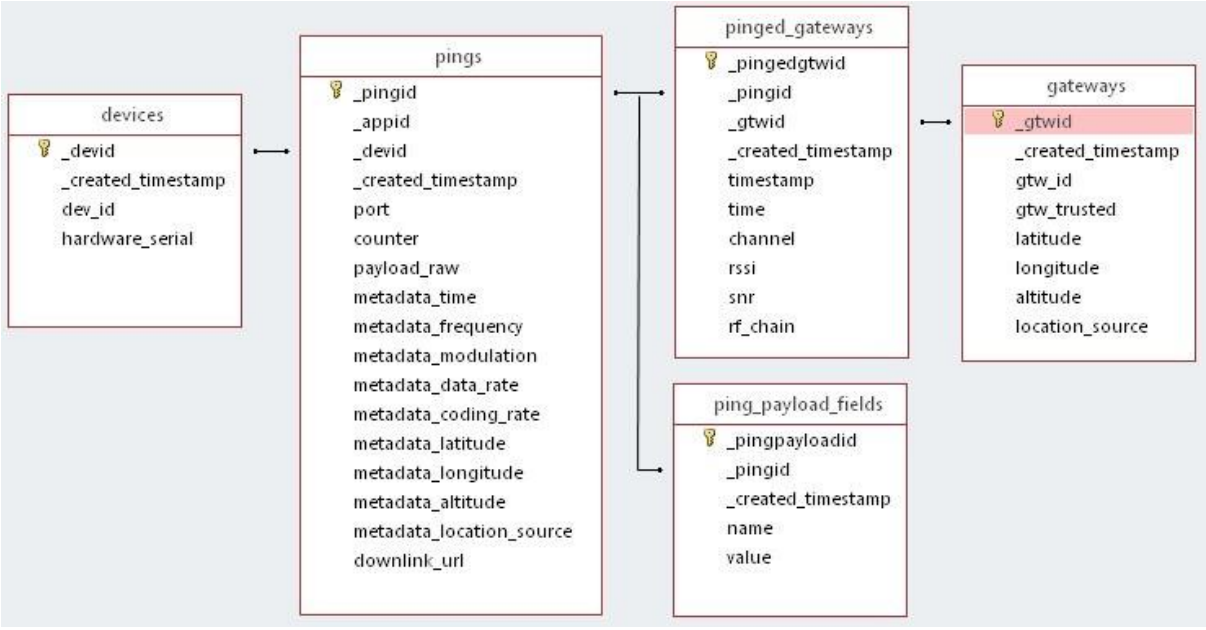


Рис. 2.3 Приклад схеми реляційної бази даних

Транзакції Postgres відповідають принципам ACID (Atomicity, Consistency, Isolation, Durability). Цей набір властивостей гарантує консистентність та довговічність даних. Під час транзакцій в RDBMS ACID виконується для всіх даних бази, на відміну від MongoDB, що підтримує ці властивості лише на рівні одного документа.

Дані які буде зберігати в собі сховище метаданих, більше підходить для реляційної моделі, тому прийняте рішення використовувати PostgreSQL як СУБД.

2.3.3 Черги задач

Роль посередника між сервісами виконує компонент черги задач. Існує велика кількість інструментів та технологій, які можуть забезпечити ці функції. На даний момент часу двома стовпами в сфері розсилки повідомлень стали такі технології: RabbitMQ та Apache Kafka. Порівняймо їх між собою.

RabbitMQ - це надійний та давно зарекомендований брокер повідомлень, який підтримує кілька протоколів: AMQP, MQTT, STOMP. Ця технологія демонструє високу пропускну здатність [23]. Типовий варіант використання - це обробка

фонових задач або робота посередника повідомлень між сервісами. Kafka - це шина повідомлень, оптимізована для потоків даних. Kafka можна розглядатися як довготривалий посередник повідомлень, де програми можуть обробляти та зберігати потокові дані на диск [24].

Головна різниця між Apache Kafka та RabbitMQ полягає в тому, що черги повідомлень у Kafka постійні. Надіслані дані зберігаються до тих пір, поки не пройде визначений період часу або поки черга не досягне свого максимального розміру. Це означає, що повідомлення у Kafka не видаляються після їх споживання. Така особливість надає можливість відтворення повідомлень або їх повторного використання. У RabbitMQ повідомлення зберігаються, доки споживач його не отримає. Як тільки повідомлення акредитовано, воно видаляється з черги.

RabbitMQ здатний підтримувати велику кількість протоколів, але часто розробники використовують AMQP. AMQP - це стандартизований протокол для обміну асинхронними повідомленнями. Його використовує не тільки RabbitMQ, а й Apache ActiveMQ, Amazon SQS, Apache Qpid, StormMQ тощо. Це означає, що за необхідності можна замінити RabbitMQ на іншу технологію, не змінюючи клієнтський код. У свою чергу, Kafka використовує спеціальний протокол поверх TCP / IP. Цю технологію не можна просто замінити на іншу, оскільки це єдине програмне забезпечення, яке реалізує цей протокол.

Однією з головних переваг RabbitMQ є можливість гнучко направляти повідомлення [25]. Прямо або за допомогою регулярного виразу маршрутизація дозволяє повідомленням потрапляти до певних черг без додаткового коду. Треба розуміти, що повідомлення в RabbitMQ не надсилаються безпосередньо в чергу. Натомість клієнт надсилає повідомлення в обмінник. Обмінник - це агенти маршрутизації повідомлень, визначений віртуальним хостом RabbitMQ. Він відповідає за маршрутизацію повідомлень до різних черг за допомогою атрибутів заголовка, прив'язок ("посилання", яке дається для встановлення

зв’язку між чергою та обмінником) та ключів маршрутизації (атрибут повідомлення, на який дивиться RabbitMQ, приймаючи рішення про те, в які черги його перенаправити).

Kafka не підтримує маршрутизацію. У цій технології існують топік - аналог черг та партиції - частина топіка, яка визначається за якоюсь ознакою повідомлення. Саме використовуючи партиції,можна організувати щось схоже на маршрутизацію в RabbitMQ.

Починаючи з версії 3.5.0, RabbitMQ має підтримку пріоритетних черг. Це означає, що для черги можна встановити діапазон пріоритетів. Під час публікації повідомлення встановлюється його пріоритет. Залежно від пріоритету, воно розміщується у відповідному місці в черзі. У Kafka повідомлення не може бути відправлене з пріоритетом..

Брокери не надають гарантії, що повідомлення дійде до споживачів. Як клієнтам, так і споживачам потрібен механізм підтвердження доставки та обробки. І Kafka, і RabbitMQ мають підтримку клієнтських підтверджень, таким чином ці технології гарантують, що опубліковані повідомлення надходять до брокера.

Коли RabbitMQ доставляє повідомлення споживачеві, він повинен вирішити, чи слід вважати повідомлення обробленим. RabbitMQ може розглядати повідомлення, доставленим після його розсилки, або чекати, поки споживач підтвердить вручну його отримання. Обробник RabbitMQ також може відправити негативне підтвердження, коли він не справляється з повідомленням. Повідомлення буде повернуто до черги, з якої воно надійшло, ніби це нове повідомлення. Такий механізм є корисним у випадку тимчасової несправності з боку споживача. Kafka зберігає зміщення для кожного повідомлення в розділі і клієнт може встановлювати теперішню позицію самостійно.

І хоча обидві ці технології підійдуть на роль черг задач у системі автоматизації процесу збору даних, вибір пав саме на RabbitMQ. Це зумовлено його

популярністю (Celery та Apache Airflow теж використовують саме цей брокер повідомлень), простотою налаштування, гнучкою системою маршрутизації та нативною підтримкою пріоритезації повідомлень

.2.3.4 Стек технологій веб сервера та контролера

Мова програмування Python надає величезний вибір фреймворків та бібліотек для створення вебсерверів. Найпопулярнішими з них є Django та Flask.

Django - це фреймворк вебдодатків, який надає багато готового функціоналу з самого початку, дозволяючи розробникам сконцентруватися на бізнес-логіці.

Django є безкоштовним фреймворком з відкритим кодом і має чудово складену документацію та активну спільноту, яка готова допомогти при виникненні проблем.

Flask - це мікрофреймворк, головною філософією якого є постачання користувачам лише необхідних рішень. Flask поставляється з деякими стандартними функціональними можливостями і дозволяє розробникам додавати будь-яку кількість бібліотек або плагінів для розширення. Flask має просте для вивчення API та зрозумілу документацію. Фреймворк не надає ніяких інструментів для роботи з базою даних, але існують розширення для Flask, які інтегрують у нього бібліотеку SQLAlchemy. На відміну від Django ORM, SQLAlchemy надає не тільки функції ORM, а й будівника SQL запитів, що критично важливо для оптимізації роботи з СУБД.

Через те, що велика кількість функціональності Django не буде використана при написанні веб сервера, а SQLAlchemy і так буде використовуватися контролером, була обрана зв’язка Flask та SQLAlchemy для створення веб сервера.

2.3.5 Робітники та конвеєри

Для робітників було обрано технології Celery, а для конвеєрів - Apache Airflow. В подробицях ці інструменти були описані в розділі 1. Вони були обрані через свою популярність, простоту та зрозумілість.

ВИСНОВКИ ДО РОЗДІЛУ 2

У результаті розгляду інформації в даному розділі щодо класифікації задач обробки та збору даних, вибору та обґрунтування технології та засобів реалізації можна зробити наступні висновки:

1. Було класифіковано завдання збору даних як ETL- задачу, виокремлено основні етапи та складові завдань даного типу, що вплинуло на подальшу розробку високорівневої архітектури системи.
2. Після розгляду аналогів систем автоматизації збору даних було були проаналізовані найкращі їхні сторони. Використовуючи метод синтезу, було розроблено та вибрано високорівневу архітектуру, яка підходить для організації ETL-задач.
3. Спираючись на розроблену високорівневу архітектуру системи, був проведений аналіз існуючих програмних застосунків та технологій для імплементації майбутнього програмного продукту. Були обрані технології, які найкраще підходять для реалізації системи автоматизації збору даних.

РОЗДІЛ 3
ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ

3.1. Принципи архітектури коду

3.1.1 Принцип розповсюдження спільної логіки

Спираючись на високорівневу архітектуру із другого розділу, можна зробити висновок, що система автоматизації збору даних поділяється на мікросервіси. І хоча кожен із них виконує свої функції, сервіси мають велику кількість спільних сценаріїв, наприклад, відправка повідомлень у контролер або доступ до метаданих задач у сховищі.

Якщо реалізовувати мікросервісну архітектуру, не звертаючи увагу на спільну логіку, можна стикнутися з проблемою неузгодженості коду. Особливо неприємною є ситуація, коли сервіси підтримуються різними командам, і для того, щоб змінити логіку спільних функцій розробникам різних команд, доведеться переписувати код в різних сервісах. Такий підхід є незручним, неефективним та спроможним викликати багато помилок.

Фундаментальним принципом розробки програмного забезпечення є принцип DRY (Don't Repeat Yourself). Його суть проста і інтуїтивно зрозуміла програмістам: “Кожна частина знань або логіки повинна мати єдине, однозначне представлення в системі.” [26] Керуючись цим принципом, розробники мікросервісних архітектур вигадали два популярних підходи для розповсюдження спільних функццій між сервісами.

Перший підхід полягає в тому, що програмісти постачають спільну логіку як бібліотеки. Такий підхід можна виконувати за допомогою пакетних менеджерів чи спеціальних утиліт, наприклад, Bit. Це дозволяє тримати код усіх сервісів та їх спільної логіки окремо. Такі репозиторії легше сприймати розробникам, але важко підтримувати в узгодженому стані. Нерідко виникають ситуації, коли сервіси в своїх репозиторіях мають конфлікти версії та не можуть коректно працювати один з одним у такому стані.

Другий підхід - це використання стратегії монорепозиторію. Суть цього підходу в тому, що для зберігання коду всіх сервісів використовується один репозиторій систем контролю версій [27]. Таким чином легше контролювати узгодженість сервісів між собою, з'являється можливість дослідження коду всіх сервісів та широкі можливості для рефакторинга. Платою за ці плюси є великі розміри такого репозиторію та проблеми з безпекою (якщо буде скомпрометований монрепозиторій, зловмисник буде мати доступ до внутрішнього коду всіх сервісів). Для розповсюдження спільної логіки досить створити модуль в репозиторії, до якого будуть звертатися всі сервіси, яким це потрібно.

У своїй роботі я обрав підхід монорепозиторію, бо кількість коду, який буде написаний під час розробки, є відносно невеликою, а програмне забезпечення є відкрите.

3.1.2 Принцип предметно-орієнтованого проектування

Через те, що в системі наявна велика кількість спільної бізнес- логіки, має сенс сконцентруватися на написанні такого програмного коду, який можна було б використовувати в різних сервісах, незважаючи на фреймворки, за допомогою яких вони створені. Для імплементації даної вимоги було прийняте рішення керуватися принципами "Чистої Архітектури" Роберта Мартіна та використовувати патерни предметно-орієнтованого проектування.

Предметно-орієнтованого проектування - це концепція того, що структура та мова програмного коду (імена класів, методи класів, змінні класу) повинні відповідати бізнес-домену. Цей термін був придуманий Еріком Евансом у його однойменній книзі [28].

Головною ідею всіх цих правил, що допомагають досягти незалежності бізнес-логіки від конкретних бібліотек та фреймворків, є розділення коду програмного забезпечення на прошарки та збереження ієрархії рівнів коду (рис 3.1).

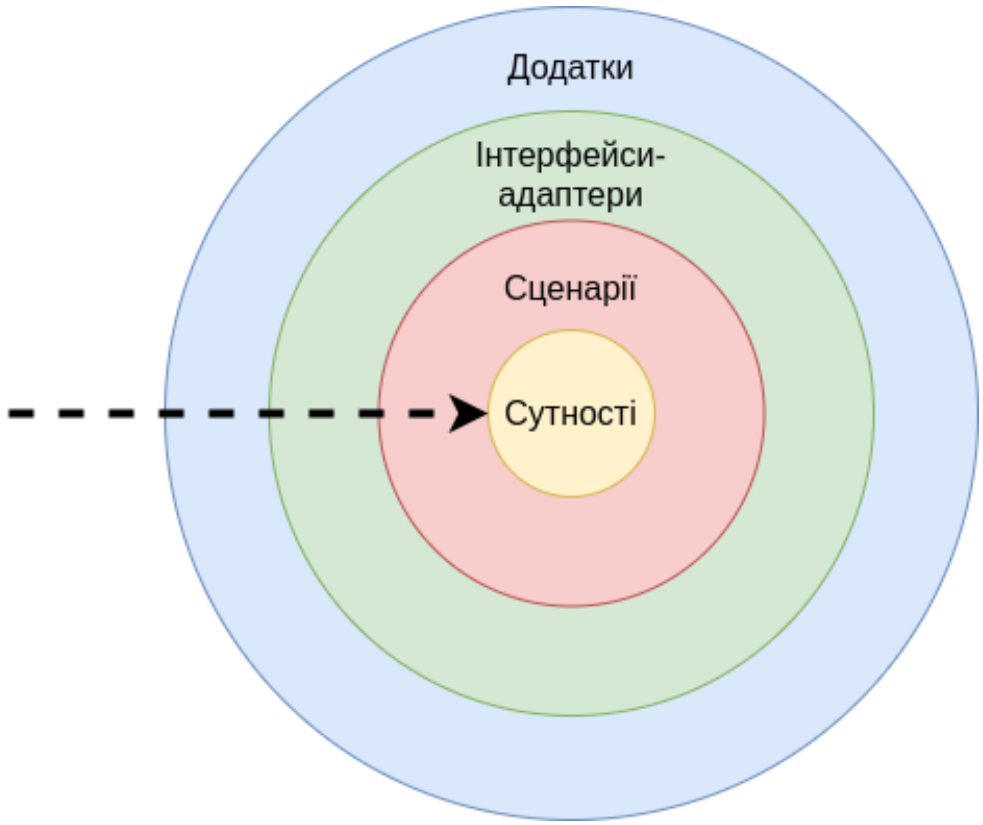


Рис. 3.1 Направлення ієрархії залежності між прошарками коду

Згідно чистої архітектури, найголовнішим прошарком всієї програми є **рівень сутностей**. Цей прошарок є базисним, від сутностей залежить інший код. На цьому рівні описуються основні елементи бізнес-логіки. У DDD цей рівень описують такими паттернами: сутність, об’єкт значення, агрегат та подія.

Сутність - об'єкт, який визначається не своїми атрибутами, а своєю ідентичністю та важливість для бізнесу.

Об’єкт значення - це об’єкт, який містить атрибути, але не має концептуальної ідентичності. До них слід ставитись як до незмінних.

Група об’єктів, пов’язаних між собою кореневою сутністю, - **агрегат**. Об'єкти за межами агрегату можуть зберігати посилання на корінь, але не на будь-який інший об'єкт агрегату. Корінь сукупності відповідає за перевірку узгодженості змін у сукупності.

Подія - об’єкт домену, який визначає подію (те, що відбулося в системі).

Наступним рівнем є **прошарок** сценаріїв. На ньому відбувається виконання коду операцій над елементами рівня сутностей та абстракціями з вищих рівнів. Головною задачею функцій є оркестрація. Цей прошарок знає про існування сутностей, тому є залежним від нього. Але логіка цього рівня є незалежною від фреймворків, драйверів, сховищ даних та інших елементів систем. Це досягається шляхом впровадження важливого паттерну SOLID [29], а саме: принципу інверсії залежностей (DIP - dependency inversion principle). Суть цього принципу полягає в тому, що системний код залежить не від конкретної реалізації потрібних йому об'єктів, а лише від абстракції. Впровадження цього правила дає змогу сценаріям залежати від інтерфейсів адаптерів елементів системи, а не від їх імплементації. При необхідності в сценарій можна передати будь-яку реалізацію адаптера. Таким чином цей прошарок залежить від інтерфейсу об’єкта для доступу до елементів системи, а не від конкретної реалізації.

Наступним прошарком є рівень інтерфейсів-адаптерів. Цей рівень реалізує очкуваний сценаріями API для взаємодії з елементами системи. Він залежний від прошарку сценаріїв, бо його елементи зобов'язані реалізовувати конкретний інтерфейс. На цьому рівні можна зустріти такі патерни DDD: репозиторій, одиниця роботи, сервіси, шина повідомлень.

Останнім прошарком є рівень додатку. На ньому знаходяться фреймворки, драйвера, бази даних, графічний інтерфейс та інші елементи системи. Цей прошарок повністю залежний від усіх нижчих рівнів. Він підставляє конкретні реалізації під потрібні інтерфейси.

Далі відбудеться огляд реалізації системи автоматизації збору даних по прошаркам, що викладені в цьому підрозділі

3.2 Опис сутностей

3.2.1 Task

Сутність “Задача” - ключова сутність в системі. Для того, щоб запустити якусь

задачу в системі автоматизації збору даних, користувач повинен створити цю сутність. При цьому Task є одночасно і агрегатом для сутностей TaskLog та TaskMetadata. Задача зберігає список усіх журналів, які з нею пов'язані та містити метадані.

Під час проектування та створення системи було визначено, що сутність Task має складатися з наступних атрибутів:

- унікальний ідентифікатор, за яким можна встановити конкретну задачу;
- поля, які характеризують задачу: дата та час створення, статус, пріоритет, аргументи задачі;
- метаінформація, що характеризує тип цієї задачі;
- список журналів, що пов'язані з задачею.

Сутність відображена в мові програмування Python за допомогою бібліотеки dataclass (рис 3.2).

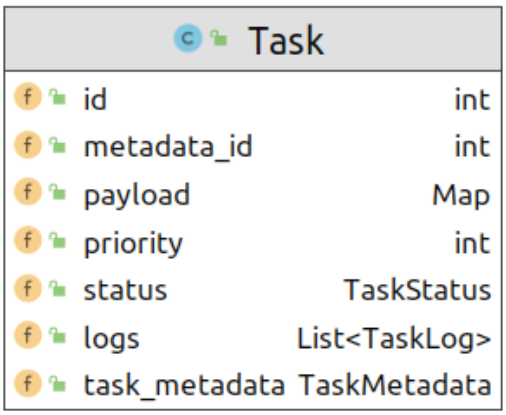


Рис. 3.2 Клас Task

Клас Task має наступні поля:

- id - унікальний ідентифікатор задачі типу int;
- metadata_id - ідентифікатор метаінформації, за допомогою якої створено конкретну задачу;
- payload - поле типу dict, що відображає аргументи, які передали для

виконання задачі;

- status - атрибут, що визначає стан задачі (під час розробки системи автоматизації збору даних було прийнято рішення обмежити це поле класом Enum з п’ятьма значеннями: CREATED, QUEUED, CONSUMED, READY, ERROR);
- created_date - поле типу datetime, що означає час створення задачі;
- priority - визначає пріоритет задачі в системі, де 1 - найнижчий пріоритет, а 5 - найвищий;
- task_metadata - атрибут, що містить в собі сутність TaskMetadata;
- logs - поле, що є списком сутностей TaskLog.

3.2.2 TaskMetadata

Сутність TaskMetadta - це допоміжна інформація про задачі, що характеризуються однаковим типом. У однієї задачі може бути лише одина метаінформація, але у метаінформації може бути багато задач. Таким чином встановлюється зв’язок “один до багатьох” між метаінформацією та задачею.

Під час проектування та аналізу системи автоматизації та збору даних було визначено, що сутність метаінформації повинна мати наступні поля:

- унікальний ідентифікатор;
- назву;
- назву черги, до якої має попасти задача даного типу;
- схема аргументів задачі даного типу, за якою можна проаналізувати коректність вводу задачі.

В програмному коді був створений дата клас TaskMetadata (рис 3.3), який відображає дану сутність.

Сутність TaskMetadata описана в класі такими полями:

- id - унікальний ідентифікатор типу int;
- name - атрибут типу str, що описує ім’я сутності;

- queue - назва черги, в яку потрапляє задача цього типу;
- schema - поле типу dict, яке за допомогою спеціального синтаксису описує схему аргументів задач цього типу.

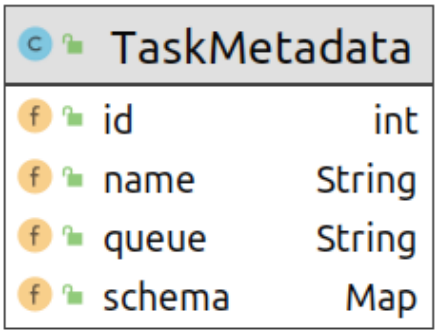


Рис. 3.3 Клас TaskMetadata

3.2.3 TaskLog

TaskLog - це сутність, що відображає журнали, які пов’язані з якимось завданням. Ця сутність відображає повідомлення про зміни стану, помилку чи додаткову інформацію. У одного журналу може бути лише одна задача, але у задачі може бути багато журналів. Це означає, що між завданням та журналом має бути встановлений зв’язок “один до багатьох”.

Сутність TaskLog повинна мати такі поля:

- id - унікальний ідентифікатор;
- task_id - ідентифікатор задачі, з якою пов’язаний даний журнал;
- type - тип журналу;
- message - коротке повідомлення;
- description - довге повідомлення.

В програмі був створений клас TaskLog (рис 3.4), що реалізує дану сутність в системі.

Цей клас має наступні поля:

- id - атрибут типу int, що є унікальним ідентифікатором журналу;

- task_id - ідентифікатор сутності Task, типу int;
- type - атрибут, який відображає тип журналу (під час розробки було вирішено встановити цьому полю тип Enum із такими значеннями: MESSAGE, STATUS_CHANGE, ERROR);
- message - поле типу str, що означає коротке повідомлення;
- description - опціональне поле для додаткової інформації типу optional[str], яке можна не вводити.

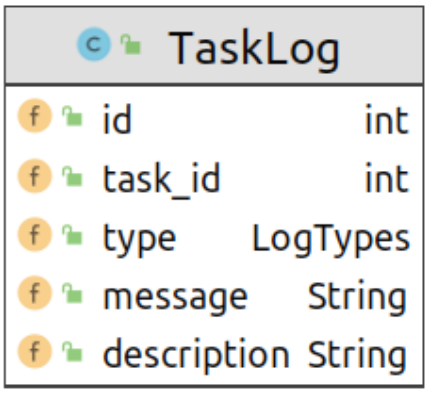


Рис. 3.4 Клас TaskLog

3.2.4 Event

Event - це не домена подія, яка вказувалась вище при описі принципів архітектури. Цей об’єкт має лише одну ціль - стандартизація повідомлень, які передаються між сервісами. З точки зору домено-вмотивованої архітектури Event більше схожий на патерн Data Transfer Object (DTO). DTO - це простий іммутабельний об’єкт, який використовують для стандартизації передавання інформації між підсистемами.

Для Event DTO були встановлені такі атрибути:

- name - назва події;
- priority - пріоритет цієї події;
- payload - додаткова інформація, яка може знадобитися обробнику подій.

У програмному коді сутність Event була реалізована, шляхом використання декоратору dataclass (рис 3.5) з опцією frozen=True, що дало можливість забезпечити іммутабельність даного об'єкта.

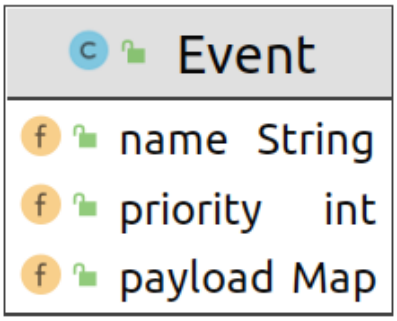


Рис. 3.5 Клас Event

Цей клас має такі атрибути:

- name - поле типу str, що відображає ім’я події;
- priority - пріоритет події, тип int;
- payload - атрибут типу dict. що містить в собі додаткову інформацію про подію.

3.3 Опис способу доступу до даних

3.3.1 Використання ORM у DDD

У останні роки дуже популярною є ідея використання ORM для роботи з базою даних. Але класичний приклад використання ORM в прикладному коді, коли ми описуємо моделі та використовуємо їх як сутності бізнес-логіки, порушує принципи чистої архітектури та DDD. Відповідь на питання, чому це відбувається, лежить у самому визначенні ORM.

Отже, ORM (Object Relational Mapper) - це технологія в програмному забезпеченні, яка надає інструменти проектування елементів архітектури СУБД на сутності мови програмування. У випадку з реляційними базами даних це може бути проекція таблиць на класи, а рядків - на об’єкти цього класу. Проблема в

тому, що дана технологія зумовлює жорстку зв’язаність сутностей вашої системи та реалізації конкретної СУБД. Таким чином порушується принцип незалежності рівня сутностей від прошарку додатку.

Очевидний шлях вирішення цієї проблеми є відмова від ORM на користь патернів прошарку адаптерів-інтерфейсів та виконання в них запитів до бази даних. Але таке рішення має і свої недоліки: кількість однотипного коду, який виконує перетворення сутностей у формат зрозумілий для СУБД і назад, зростає дуже швидко.

Існує і альтернативний шлях. Бібліотеки та ORM, які надають відносно низькорівневий доступ до своїх абстракцій, можуть мати інструменти для ручного описання проекцій таблиць на класи. Наприклад, в SQLAlchemy присутня функція ручного мапінгу класів на таблиці бази даних. Цей підхід не порушує принципів DDD, бо в такому випадку сутності описуються стандартними інструментами мови програмування, а вже потім встановлюється відповідність між ними та таблицями СУБД. Таким чином відновлюється ієрархія залежності сутностей від елементів баз даних. При цьому логіку перетворення об’єктів бере на себе ORM. У такій реалізації існують свої мінуси: вірогідність конфліктів між ORM та класами сутностей системи.

Саме таким чином ORM використовується у даній роботі, незважаючи на мінуси, на мою думку. такий підхід є підходящим компромісом між принципами DDD та зручністю сучасних ORM.

3.3.2 Об’єкти доступу до даних

Хоч в даній роботі використовується ORM, побудова запитів за її допомогою у сценаріях порушувала б принципи ієрархії залежностей між прошарками коду. Для виконання запитів у СУБД використовується ідея із DDD під назвою об’єкт доступу до даних. Data Access Object (DAO) - це патерн в розробці програмного забезпечення, який надає абстрактний інтерфейс до будь-якого сховища даних. Слід зауважити, що для кожної сутності слід створювати свій власний DAO,

навіть якщо вони зберігаються в одній базі даних. Таке правило зумовлене наданням можливості збереження даних в різних сховищах. Сценарії спираються на абстрактні інтерфейси об’єктів доступу до даних. На вищих рівнях буде відбуватися рішення про постановку конкретної реалізації.

У цій роботі були створені інтерфейси та реалізації (PostgreSQL) для сутностей Task та TaskMetadata. Слід зауважити, що в Python не має інтерфейсів, тому замість них в цій роботі використовуються абстрактні класи, що не реалізують жодного методу.

Абстрактний клас TaskDAO оголошує методи для збереження, оновлення стану, отримання однієї чи багатьох задач. TaskDAO успадковує базовий клас abc.ABC для використання можливостей абстрактних класів. PgTaskDAO реалізує інтерфейс для роботи з PostgreSQL (рис 3.6).

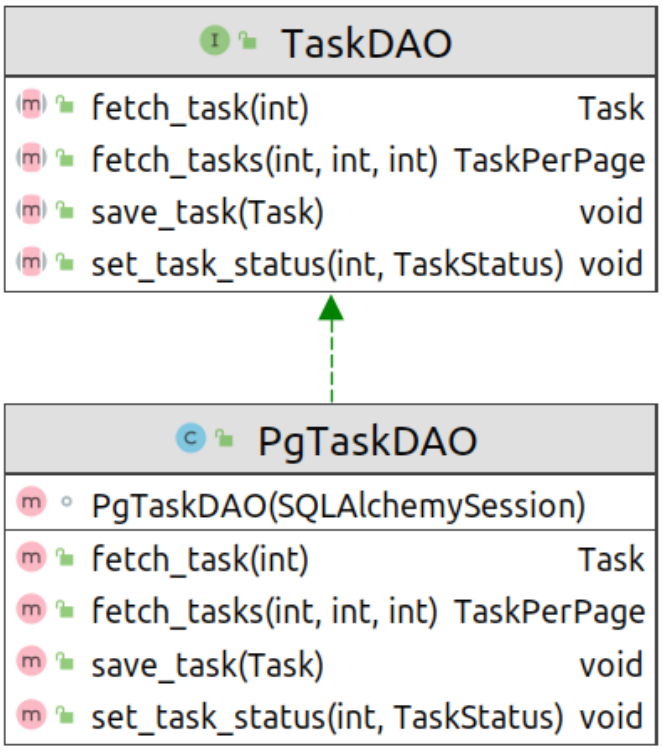


Рис. 3.6 Клас PgTaskDAO

Цей клас приймає в конструктор об’єкт класу Session, що є обгорткою над транзакцією реляційних баз даних. Важливо те, що в методах DAO не викликаються директиви COMMIT та ROLLBACK. Їх виклик має відбутися саме на рівні бізнес- логіки, щоб не порушувати консистентність даних в СУБД, бо на прошарку сценаріїв може бути потреба виконання кількох операцій DAO в одній транзакції.

Таким самим чином був запрограмований MetadataDAO та PgMetadataDAO (рис 3.7). Інтерфейс оголошує методи лише для читання одного та багатьох TaskMetadata.

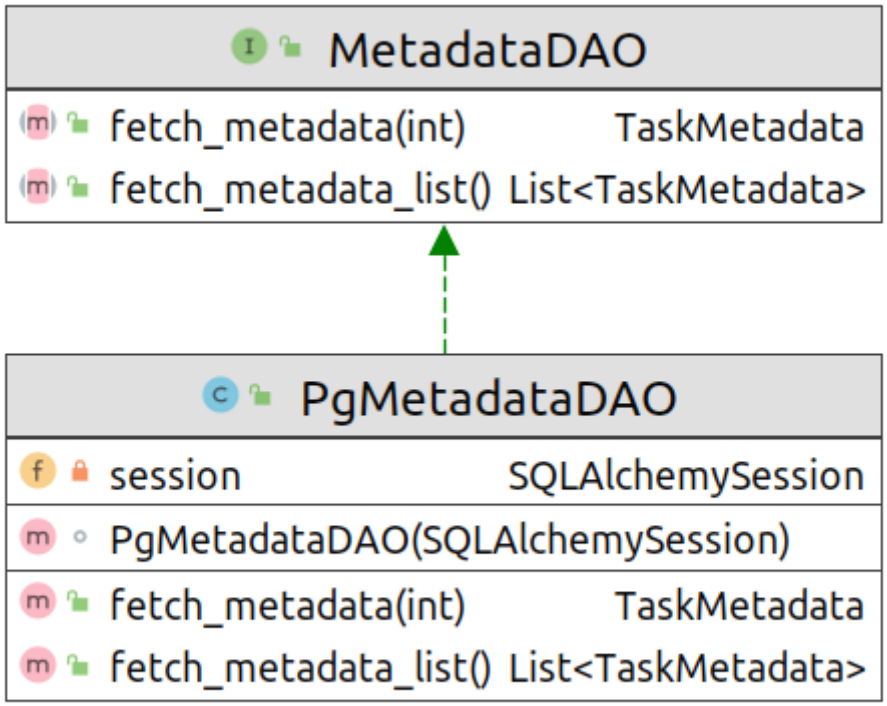


Рис. 3.7 Клас PgMetadataDAO

Інтерфейс для об’єкту длступу до даних сутносіт TaskLog (рис 3.7) оголошує лише один метод, а саме: метод додавання нового журналу до сутності Task. Так як ця сутність не є повністю незалежною від Task і не має без неї сенсу, читання журналів відбувається в методах TaskDAO, якщо це є необхідним.

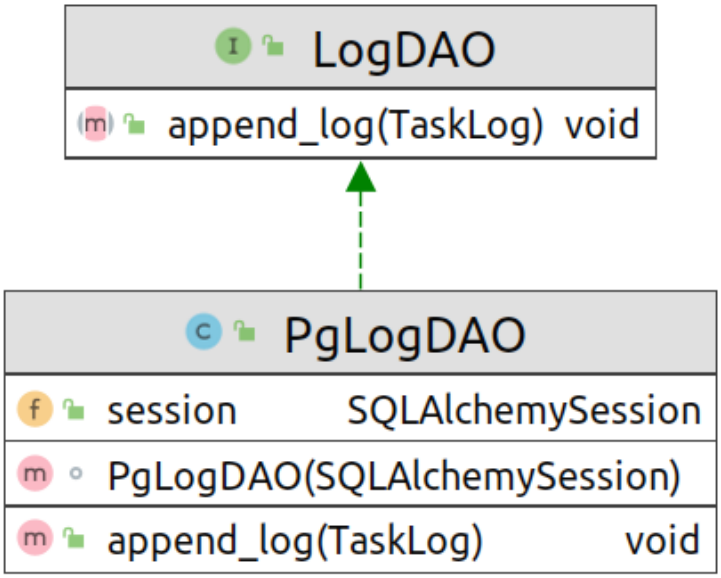


Рис. 3.8 Клас PgLogDAO

3.3.3 Одиниця роботи

З одного боку, делегування завершення транзакції PostgreSQL у рівень сценаріїв надає можливість коректно працювати з даними, але з іншого - це порушення ієрархії прошарків, бо в такому випадку рівень сценаріїв “знає” про транзакційну природу сховища в DAO. У цій роботі для вирішення даного конфлікту використовується паттерн DDD під назвою одиниця виконання.

Unit of Work (UoW) - це патерн, що дозволяє виконувати атомарні транзакції над списком об’єктів. У цьому терміні атомарність означає виконання або повністю всіх операцій в транзакції, або їх повну відміну. Насправді об’єкт Session, що передається аргументом в реалізації DAO, теж є імплементацією цього патерну бібліотекою SQLAlchemy.

UoW в даній роботі теж має свій інтерфейс та конкретну реалізацію. Абстрактний UoW інкапсулює в собі всі абстрактні патерни та надає методи для початку, підтвердження та відміни транзакцій. Для лаконічності та краси синтаксису було вирішено зробити можливість роботи з цим класом за

допомогою контекстних менеджерів Python. Цей клас реалізує контракт контекстного менеджера, а саме: магічні методи __enter__ та __exit__. Конкретна реалізація для роботи з PostgreSQL має отримати в конструктор контекстний менеджер для роботи з Session SQLAlchemy. Діаграма класів UoW зображені на рис. 3.9.

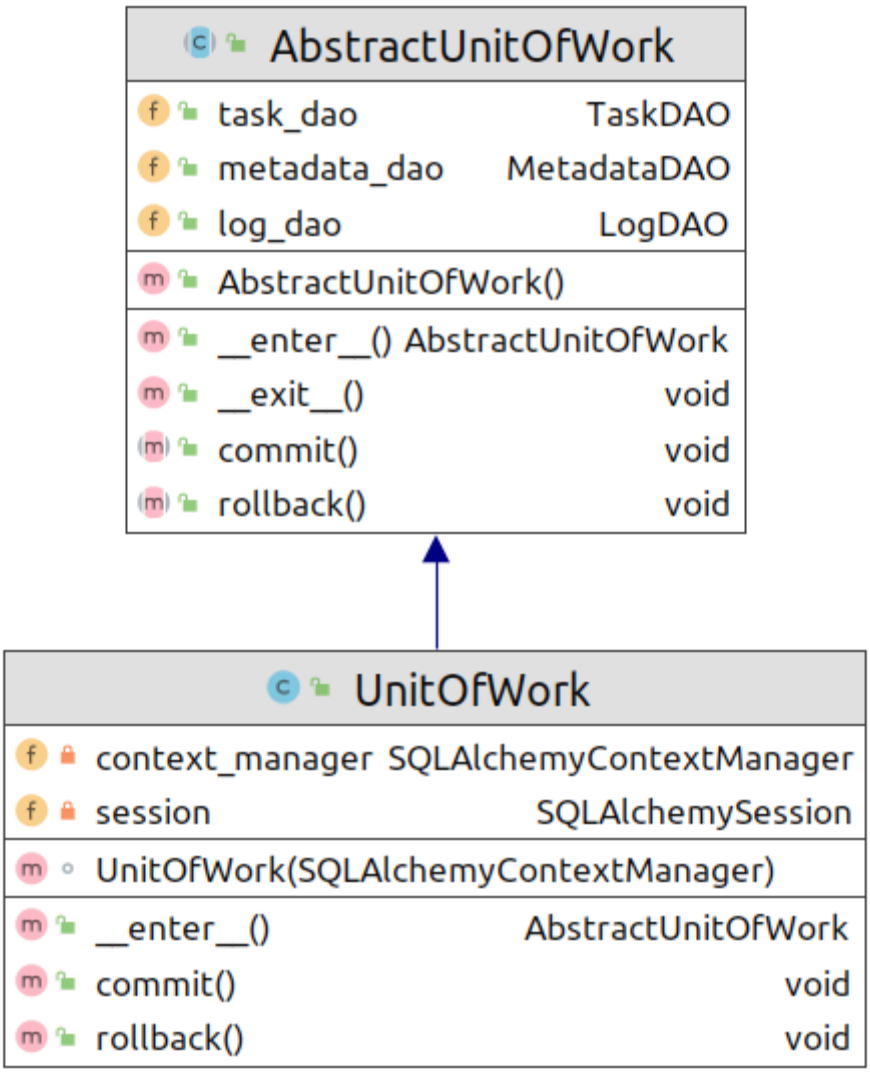


Рис. 3.9 Клас UnitOfWork

Таким чином, тепер рівень сценаріїв спирається на інтерфейс AbstractUnitOfWork, за допомогою якого можна працювати з логічним транзакціями, не знаючи їх конкретну реалізацію.

3.4. Сервіси

У контексті даної роботи сервіси - це не клас у коді, в якому скупчена бізнес логіка, як наприклад у фреймворке Java Spring. В класичній реалізації сервіси знаходяться на рівні сценаріїв, а в імплементації системи автоматизації збору даних сервіси знаходяться на рівні інтерфейсів-адаптерів. Сервіси - це абстрактні інтерфейси, які не підходять для того, щоб бути DAO. Вони зберігають інформацію, що не належить системі. Також ці сервіси не можуть бути окремими сценаріями, бо вони зав’язані на конкретну реалізацію.

3.4.1 Шина повідомлень

Шина повідомлень - це абстракція, яка надає методи доступу до мікросервіса контролера з високорівневої архітектураи. Цей інтерфейс має лише один метод - надсилання події. Реалізація цього інтерфейсу (рис 3.10) спирається на Celery задачу, яка займається маршрутизацією подій по різним чергам.

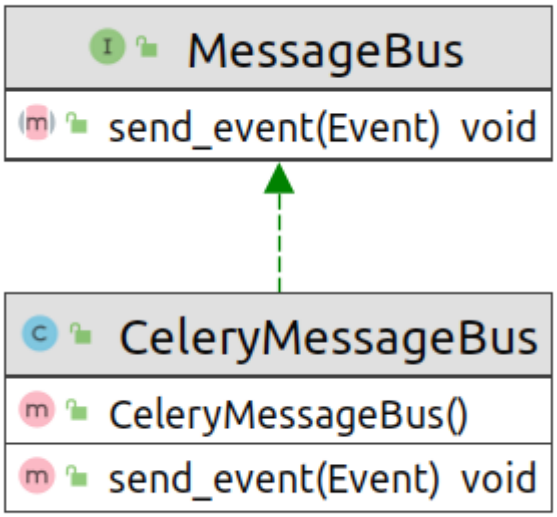


Рис. 3.10 Клас CeleryMessageBus

3.4.2 Сервіс доступу до конвеєрів

Цей сервіс є абстрактним інтерфейсом для доступу до конвеєрів. За допомогою нього можна дістати імена всіх конвеєрів та розпочати роботу одного

з них. Конкретна реалізація цього сервісу залежить від REST API Apache Airflow (рис 3.11). На жаль, Airflow не надає кращих технологій для мікросервісної взаємодії, наприклад gRPC сервер чи доступ до черги задач. Але можна очікувати, що такі API з'являться, бо ця технологія з кожним роком стає популярнішою.

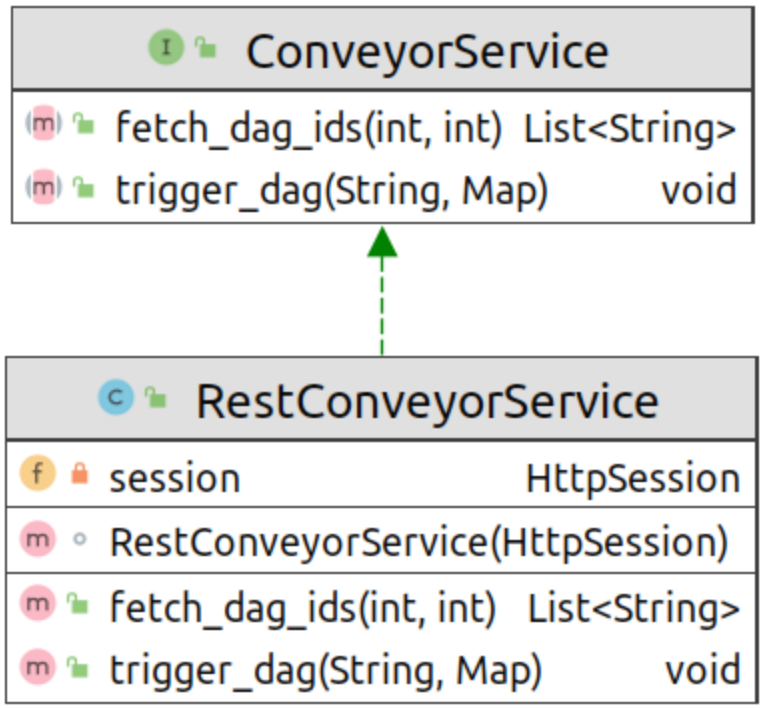


Рис. 3.11 Клас RestConveyorSrvice

3.5. Сценарії

Рівень сценаріїв накопичує в собі всю бізнес-логіку. Як було згадано вище, в деяких мовах програмування, наприклад Java, прийнято зберігати у класичних сервісах. Але в своїй книжці про DDD Еванс обговорює ідею операцій про те, що сценарії по своїй природі не мають чіткого визначення, яким саме чином вони будуть відображені в коді. Усі сценарії даної роботи дуже нагадують звичайні функції. Тому так як Python - це мультіпарадігменна мова програмування, було прийнято рішення зробити всі сценарії функціями, які приймають потрібні їм

залежності як аргументи. Опишем всі спільні сценарії.

3.5.1 Створення задачі

Згідно декларації функції створення задачі (рис 3.12), клієнтський код має передати на вхід унікальний ідентифікатор метаінформації, аргументи виконання завдання, одиницю роботи та шину повідомлень.

```
def create_task(
    metadata_id: int,
    payload: dict,
    uow: AbstractUnitOfWork,
    message_bus: MessageBus,
) -> Task
```

Рис. 3.12 Декларація функції створення задачі

Ця функція створює нову сутність Task, зберігає її за допомогою TaskDAO (він знаходиься у об’єкті uow) та надсилає повідомлення про створення задачі за допомогою шини повідомлень. Слід зазначити, що після збереження задачі в DAO функція викликає commit() в AbstractUnitOfWork для того, щоб зберегти задачу, навіть якщо станеться помилка при відправці повідомлення у мікросервіс контролера.

3.5.2 Оновлення статусу задачі

По аргументам в декларації функції-сценарію оновлення статусу задачі (рис 3.13) можна зробити висновок, що клієнтський код має передати як аргументи: унікальний ідентифікатор задачі, для якої виконується оновлення, новий статус, журнал повідомлення про зміну задачі та одиницю роботи.

Для зміни стану задачі потрібно зробити оновлення стану сутності Task за допомогою TaskDAO та зберегти журнал про цю подію за допомогою LogDAO. Тут дуже важливим є той факт, що ці дії виконуються за допомогою двох не пов’язаних між собою SQL запитах. Завдяки тому, що життєвий цикл транзакції бази даних контролює одиниця роботи, виконання цих запитів відбувається в

одній транзакції. Це забезпечує атомарність цієї оаерації та консистентність даних, інакше кажучи або всі запити виконаються коректно, або вся транзакція відміниться.

```
def update_task_status(
    task_id: int,
    status: TaskStatus,
    log_message: TaskLog,
    uow: AbstractUnitOfWork,
) -> None
```

Рис. 3.13 Декларація функції оновлення статусу задачі

3.5.3 Пагінація задач

Операція діставання списку всіх задач може бути важкою задачею для сховища та сервера. Якщо їх кількість буде великою (що можливо у високонавантажених системах), такі запити можуть сповільнити всю систему. Для уникнення такої прикрої можливості, було вирішено застосувати техніку пагінації до цієї операції. Ідея такої оптимізації тому, що ми читаємо списки об’єктів по частинам. В даній роботі реалізована по сторінкова пагінація. Користувач ітерується запитвми по сервісу, поки не завантажить потрібну йому кількість сторінок.

Для отримання задач по сторінці з можливою фільтрацією їх по метаінформації використовується функція get_tasks (рис 3.14).

```
def get_tasks(
    page: int,
    per_page: int,
    uow: AbstractUnitOfWork,
    metadata_id: Optional[int] = None,
) -> TasksPerPage
```

Рис. 3.14 Декларація функції пагінації по задачам

У ній за допомогою TaskDAO дістаються задачі для конкретної сторінки та рахується кількість усіх здач в системі. З цієї інформації формується dataclass, в якому містяться : задачі потрібної сторінки, кількість всіх задач, кількість всіх сторінок, номера наступної та минулої сторінки.

3.5.4 Інші сценарії читання

Всі інші сцеарії читання (рис 3.15) є простими обгортками над класами DAO, тому не має сенсу розглядати їх в подробицях.

```
def get_task(task_id: int, uow: AbstractUnitOfWork) -> Task

def get_metadata_list(uow: AbstractUnitOfWork) ->
List[TaskMetadata]

def get_metadata(metadata_id: int, uow: AbstractUnitOfWork) ->
TaskMetadata
```

Рис. 3.15 Декларація інших функції-сценаріїв читання

3.6. Ін’єкція залежностей

Код написаний за правилом ієрархії прошарків містить в собі велику кількість абстракцій та інтерфейсів. Відслідковувати ланцюги залежностей та контролювати способи створення об’єктів стає дуже важко. Тому розробники прикладного програмного забезпечення використовують для таких цілей технологію під назвою ін’єкція залежностей та контейнери інверсії контролю [9].

Впровадження залежностей - це стиль налаштування об'єкта, при якому поля об'єкта задаються зовнішньої сутністю. Іншими словами, об'єкти налаштовуються зовнішніми об'єктами. За те, яким чином буде створена залежність, відповідають Inversion of Control Containers (IoC Containers). Їх називають називають контейнерами, а не фабриками, тому що вони не тільки створюють залежність, а й контролюють її життєвий цикл.

У світі Python існують кілька технологій та бібліотек, що допомагають

реалізувати цей паттерн. Найвідомішим з них є бібліотека injector [10]. Вона є зручною і лаконічною, підтримує синтаксис підказок типів в Python та має розширення для використання з фреймворком Flask. У даній роботі використовується саме ця бібліотека для впровадження залежностей.

3.7. Рівень додатків

Це найвищій рівень в ієрархії прошарків коду. Рівень додатків залежить від сценаріїв бізнес-логіки. Саме в цьому підрозділі стає зрозуміло, чому було прийнято рішення написання коду в DDD стилі. Деякі сценарії перевикористовуються у різних сервісах.

3.7.1. REST API

Система автоматизації збору даних надає REST API (Representational State Transfer Application Programming Interface) для взаємодії з нею.

Таблиця 3.1. Опис прикладного програмного інтерфейсу

Шлях	Опис
POST /api/tasks	Створити задачу
GET /api/tasks?page=:int &per_page=:int &metadata_id=:int	Посторінкове отримання завдань
PATCH /api/tasks/:id	Оновлення статусу задачі по id
GET /api/tasks/:id	Отримання задачі
GET /api/metadata	Отримання списку метаінформації задач
GET /api/metadata/:id	Отримання однієї сутності метаінформації

Для зручного тестування REST API, по шляху GET /api доступний Swagger UI (рис 3.16).

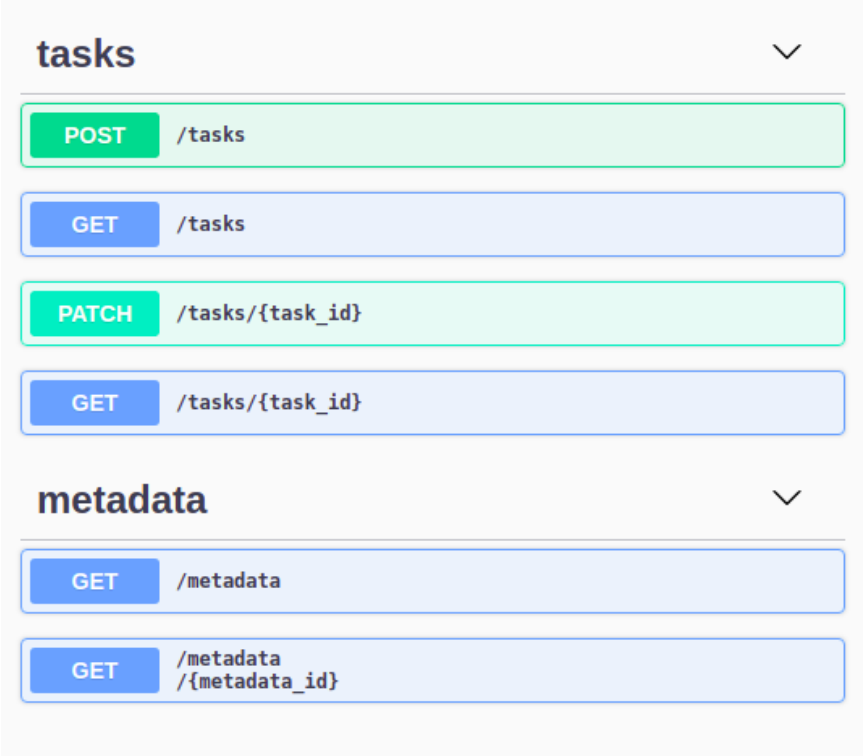


Рис. 3.16 Swagger UI для REST API системи

3.7.2. Адмінська панель

Окрім REST API, вебсервер надає ще й адмінську панель для зручного керування тим функціоналом, що не має бути доступним звичайним користувачам. Ця панель була побудована завдяки розширенню фреймворку Flask під назвою Flask-Admin. На ній можна створювати, редагувати, видаляти задачі, журнали та метаінформацію.

3.7.3. Контролер та робітники

Контролер та робітники були реалізовані за допомогою фреймворка для асинхронних задач Celery. І хоча ці два сервіси мають спільну кодову базу, вони мають різні черги задач, тому вони не будуть блокувати один одного, якщо запускати їх в різних процесах.

Ще один плюс цього рішення - це спільний WEB UI Flower та можливість локального запуска споживачів черг в одному процесі (рис. 3.17). Інформація про черги задач в Flower WEB UI зображена на рисунку 3.18.

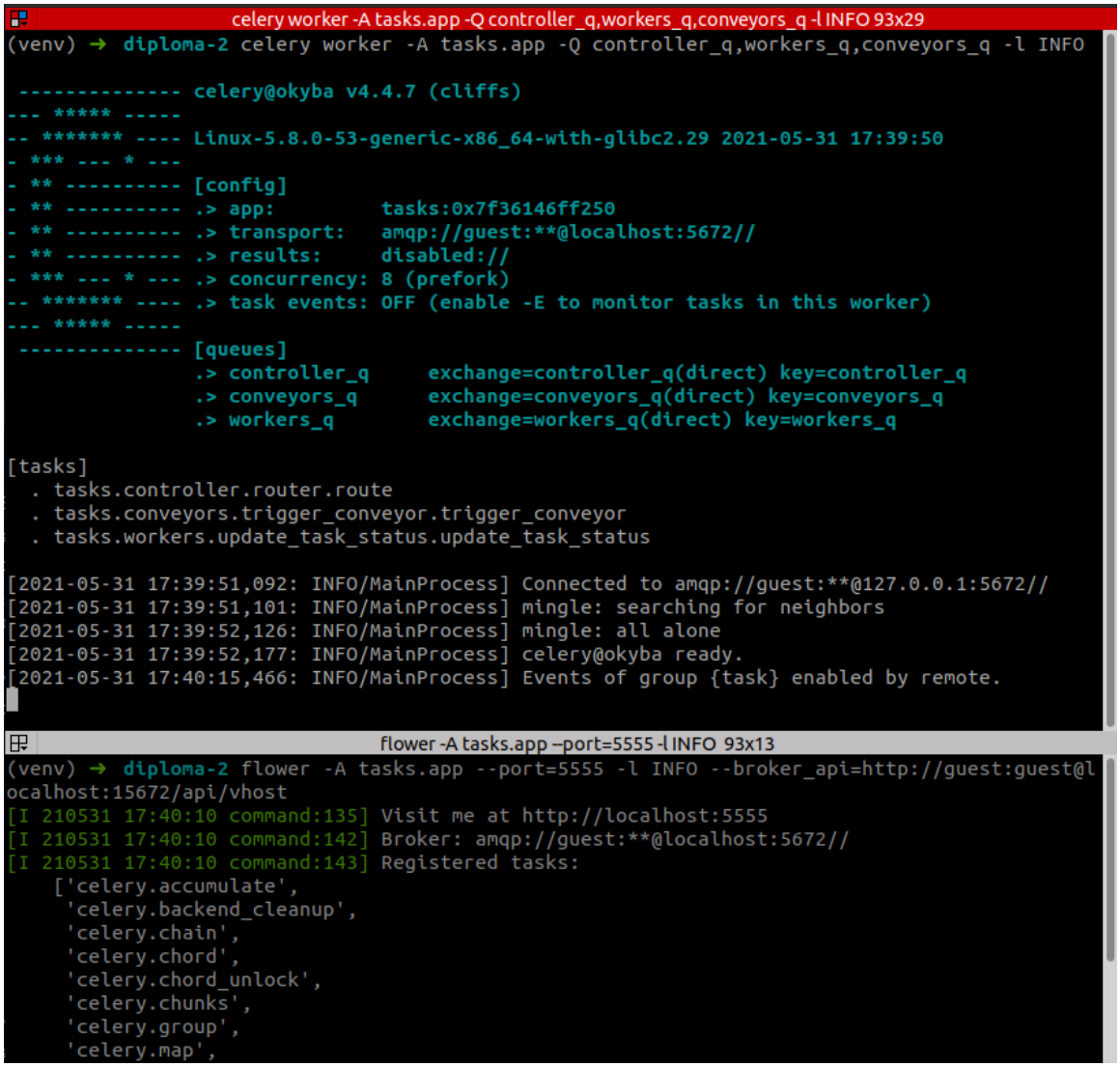


Рис. 3.17 Запуск контролера, робітників та Flower UI

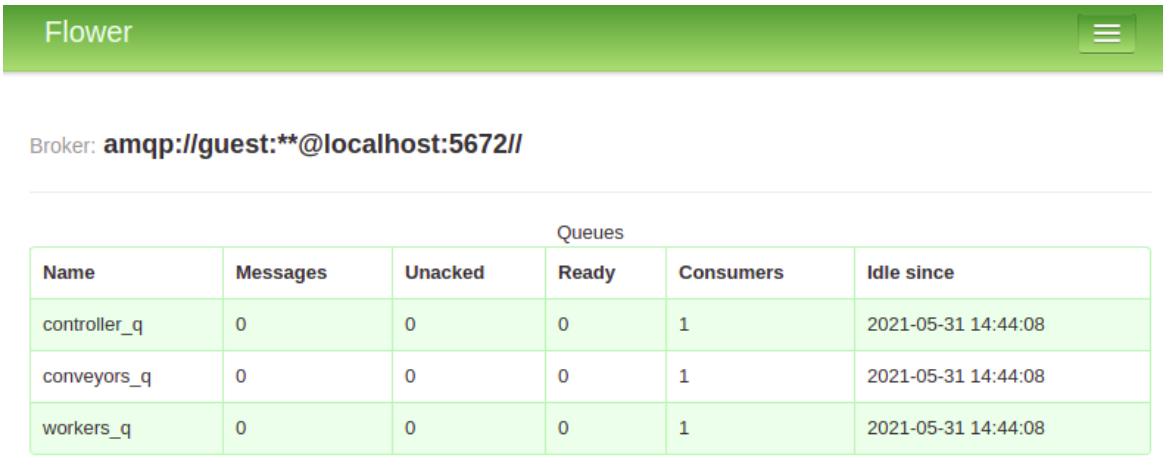


Рис. 3.18 Статистика у Flower UI про черги задач

3.7.4. Конвеєри

Як і було зазначено вище, конвеєри були створені за допомогою технології Apache Airflow. В цілях тестування коректності функціонування системи було створено 3 конвеєри зі схожою функціональністю. Їх алгоритм виконання виглядає таким чином:

1. Перевірка коректності вхідних даних;
2. Надсилання повідомлення про зміну статусу задачі в стан CONSUMED;
3. Збір даних з відкритих джерел (наприклад NASA API), за аргументами, що надає користувач минулому кроці;
4. Конвертація даних у CSV файл;
5. Збереження файлу в сховище Amazon S3;
6. Надсилання файлу на пошту користувача, яку він вказав у вхідних даних;
7. Повідомлення про зміну статусу в READY.

На кожному з цих кроків може статися помилка, розробник відповідальний зробив такий обробник помилок, що повідомити системі про цю подію. Найпростіший спосіб це зробити - це передати функцію зворотного виклику on_failure_callback при ініціалізації ациклічного графу Airflow. Таким самим чином можна змінювати стан при успіху роботи графа за допомогою on_succes_callback.

ВИСНОВКИ ДО РОЗДІЛУ 3

При розробці програмного забезпечення та написанні цього розділу було зроблено наступні висновки:

1. На початку розділу була описана проблема розповсюдження спільної логіки в мікросервісній архітектурі. Були розглянуті популярні підходи до вирішення доставки спільного коду та був обґрунтований вибір саме монорепозиторію для вирішення цієї проблеми в даній дипломній роботі.
2. По прошаркам коду було пояснене призначення кожного програмного елемента. Розглянуто основні сутності, об’єкти доступу до даних, одиницю роботи, сервіси та сценарії бізнес-логіки. Досліджено всі проблеми, з якими я стикнувся під час розробки програмного забезпечення та пояснено, яким чином їх можна вирішити найоптимальнішим шляхом.
3. Технології та інструменти, що були зазначені в другому розділі, були краще описані та застосовані на практиці. Було розглянуто та перевірено на практиці, яким чином можна використовувати сторонні бібліотеки та фреймворки, щоб не порушувати правило ієрархії прошарків коду.
4. За допомогою спільних сценаріїв бізнес-логіки було розроблено та продемонстровано всі необхідні компоненти системи автоматизації збору даних: вебсервер, контролер, робітники та конвеєри.

ВИСНОВКИ

У результаті виконання даної дипломної роботи була розроблена сучасна система автоматизації збору даних, яка орієнтована на виконання завдань “на вимогу”. Цей сервіс можна використовувати як окремий додаток, так і як частину великої мікросервісної системи. Даний сервіс надає REST API для синхронної взаємодії, а для асинхронних викликів система пропонує стандартизовану точку входу для маршрутизації повідомлень через AMQP протокол. Програмний комплекс спроектований таким чином, щоб при збільшенні навантаженні на нього в розробників була можливість горизонтального масштабування системи шляхом збільшення кількості екземплярів компонентів сервісу.

У першому розділу дипломної роботи були розглянуті існуючі аналоги систем автоматизації збору та роботи з даними. Були виявлені переваги та недоліки цих технологій, що допомогло спроектувати таку архітектуру сервісу, що могла б вирішувати ширший аспект проблем.

У другому розділі була виконана класифікація завдання збору даних як ETL задачі, що допомогло краще зрозуміти вимоги до проектованої системи. На основі інформації з першого розділу була розроблена високорівнева архітектура сервісу, що може вирішувати прості та конвеєрні задачі збору даних, легко горизонтально масштабуватися та інтегруватися у вже існуючі мікросервісні системи. Був обраний стек технологій, що дозволив максимально ефективно розробити сервіс автоматизації збору даних.

У третьому розділі були розглянуті та обґрунтовані принципи написання програмного коду системи. Для досягнення цілі розповсюдження спільної логіки відбулася класифікація коду на прошарки. Були описані шляхи розробки компонентів сервісу, пояснені архітектурні рішення та способи впровадження технологій та інструментів з другого розділу таким чином, щоб не порушувати правила DDD. Кожна компонента системи автоматизації збору даних була розглянута та описана.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. What’s a Task Queue? Документація Celery. — [Електронний ресурс] Режим доступу https://docs.celeryproject.org/en/stable/getting-started/introduction.html#what-s-a-task-queue
2. Messaging at Scale at Instagram — [Електронний ресурс] Режим доступу https://www.youtube.com/watch?v=E708csv4XgY
3. Understanding Celery's architecture — [Електронний ресурс] Режим доступу https://subscription.packtpub.com/book/application_development/9781783288397/7/ch07lvl1sec45/understanding-celery-s-architecture
4. Flower - Celery monitoring tool— [Електорний ресурс] Режим доступу https://flower.readthedocs.io/en/latest/
5. Building workflows. Документація Luigi — [Електронний ресурс] Режиим доступу https://luigi.readthedocs.io/en/stable/workflows.html
6. Apache Airflow overview — [Електронний ресурс] Режиим доступу https://uk.wikipedia.org/wiki/Apache_Airflow
7. Спрямований ациклічний граф — [Електронний ресурс] Режиим доступу https://ru.wikipedia.org/wiki/%D0%9E%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D1%8B%D0%B9_%D0%B0%D1%86%D0%B8%D0%BA%D0%BB%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B3%D1%80%D0%B0%D1%84
8. DAGs Concept. Документація Apache Airflow — [Електронний ресурс] Режиим доступу https://airflow.apache.org/docs/apache-airflow/stable/concepts/dags.html#dags
9. Architecture Overview. Документація Apache Airflow — [Електронний ресурс] Режиим доступу https://airflow.apache.org/docs/apache-airflow/stable/concepts/overview.html#architecture-overview
10. Executor Types. . Документація Apache Airflow — [Електронний ресурс]

Режиим доступу https://airflow.apache.org/docs/apache-airflow/stable/executor/index.html#executor-types

11. Pods. Документація Kubernetes — [Електронний ресурс] Режиим доступу https://kubernetes.io/docs/concepts/workloads/pods/
12. Why gRPC? — [Електронний ресурс] Режиим доступу https://grpc.io/
13. (Re)Introducing Edgestore ? — [Електронний ресурс] Режиим доступу https://dropbox.tech/infrastructure/reintroducing-edgestore
14. What is the ELK Stack? Why, it’s the Elastic Stack. Документація ElasticStack — [Електронний ресурс] Режиим доступу https://www.elastic.co/what-is/elk-stack
15. Elasticsearch overview — [Електронний ресурс] Режиим доступу https://ru.wikipedia.org/wiki/Elasticsearch
16. Logstash overview — [Електронний ресурс] Режиим доступу https://uk.wikipedia.org/wiki/Logstash
17. Kibana overview — [Електронний ресурс] Режиим доступу https://ru.wikipedia.org/wiki/Kibana
18. Beats overview — [Електронний ресурс] Режиим доступу https://www.elastic.co/beats/
19. Олександ Макрішов. Основні функції ETL систем . // Habr. — 2015. — [Електронний ресурс] Режим доступу https://habr.com/ru/post/248231/
20. Python overview — [Електронний ресурс] Режиим доступу https://uk.wikipedia.org/wiki/Python
21. Document Database. Документація MongoDB. — [Електронний ресурс] Режим доступу https://www.mongodb.com/manual/introduction/#document-database
22.. What Is PostgreSQL? Документація PostgreSQL. — [Електронний ресурс] Режим доступу https://www.postgresql.org/docs/13/intro-whatis.html

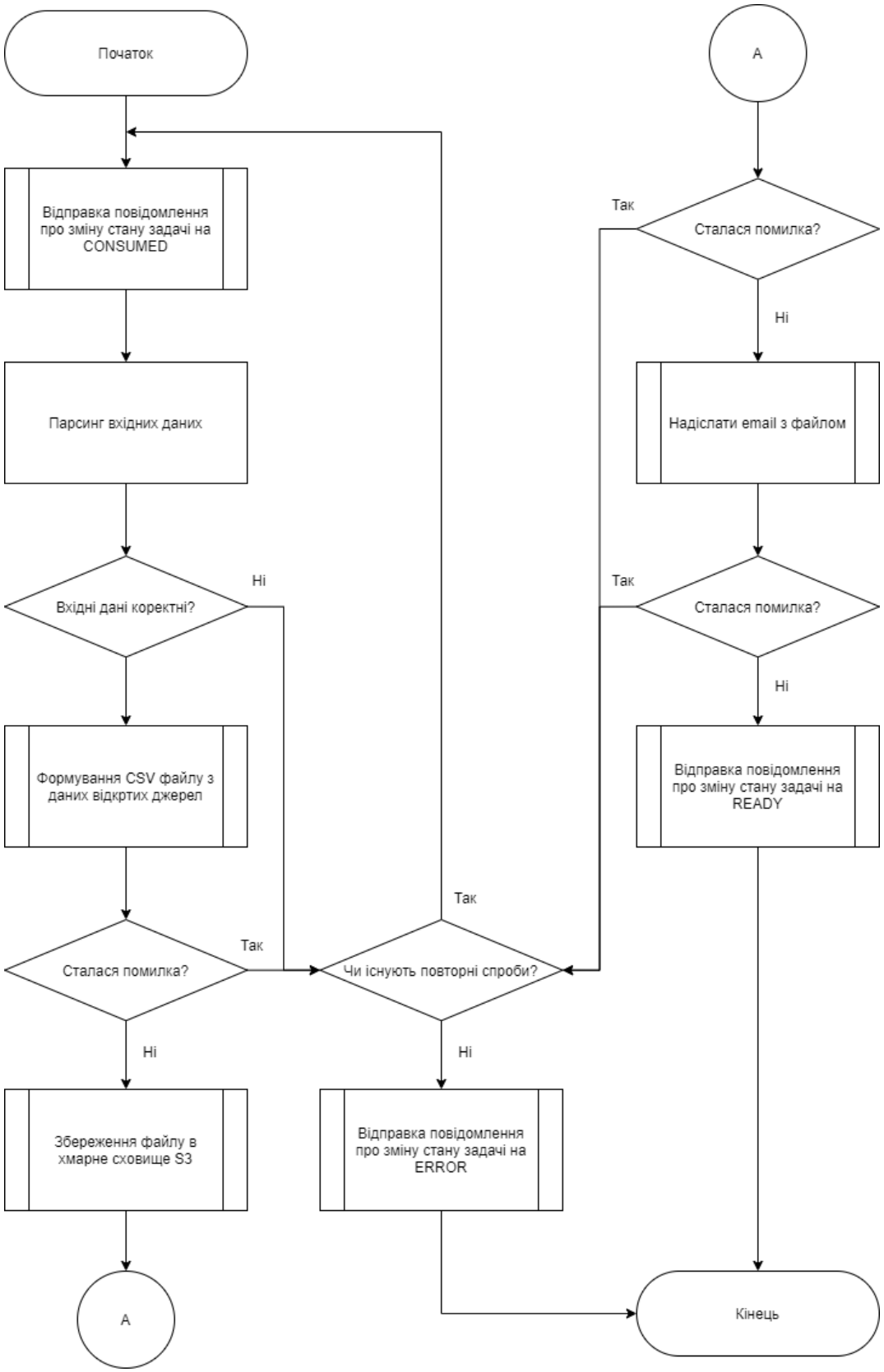
23. Networking and RabbitMQ. Документація RabbitMQ. — [Електронний ресурс] Режим доступу https://www.rabbitmq.com/networking.html
24. Apache Kafka is an event streaming platform. What does that mean? Документація Kafka. — [Електронний ресурс] Режим доступу https://kafka.apache.org/documentation/#intro_platform
25. Routing. Документація RabbitMQ. — [Електронний ресурс] Режим доступу https://www.rabbitmq.com/tutorials/tutorial-four-python.html
26.. Arvind Singh Baghel. Software Design Principles DRY and KISS // DZone. — 2018 — [Електронний ресурс] Режим доступу https://dzone.com/articles/software-design-principles-dry-and-kiss
27. Монорепозиторій — [Електронний ресурс] Режиим доступу https://ru.wikipedia.org/wiki/%D0%9C%D0%BE%D0%BD%D0%BE%D1%80%D0%B5%D0%BF%D0%BE%D0%B7%D0%B8%D1%82%D0%BE%D1%80%D0%B8%D0%B9
28. E. Evans Domain-Driven Design: Tackling Complexity in the Heart of Software //.Addison-Wesley— 2004. — P. 15
29. SOLID — [Електронний ресурс] Режиим доступу https://ru.wikipedia.org/wiki/SOLID_(%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)

ДОДАТОК 1
Система автоматизації процесу збору даних

Принципова схема – алгоритм роботи конвеєра
ІАЛЦ.467200.004 Д1

Аркушів 1

Київ – 2021



Зм.	Арк.	№ докум.	Підпис	Дата	*ІАЛЦ.467200.004 Д1*		
Розроб.		Киба О. Д.			*Принципова схема. Алгоритм роботи конвеєра*	Лит.	Арк. 1 / Аркушів 1
Перев.		Алещенко О. В.					
Реценз.						НТУУ «КПІ», ФІОТ, ІВ-72	
Н. Контр.		Сімоненко В. П.					
Затверд.		Стіренко С. Г.					

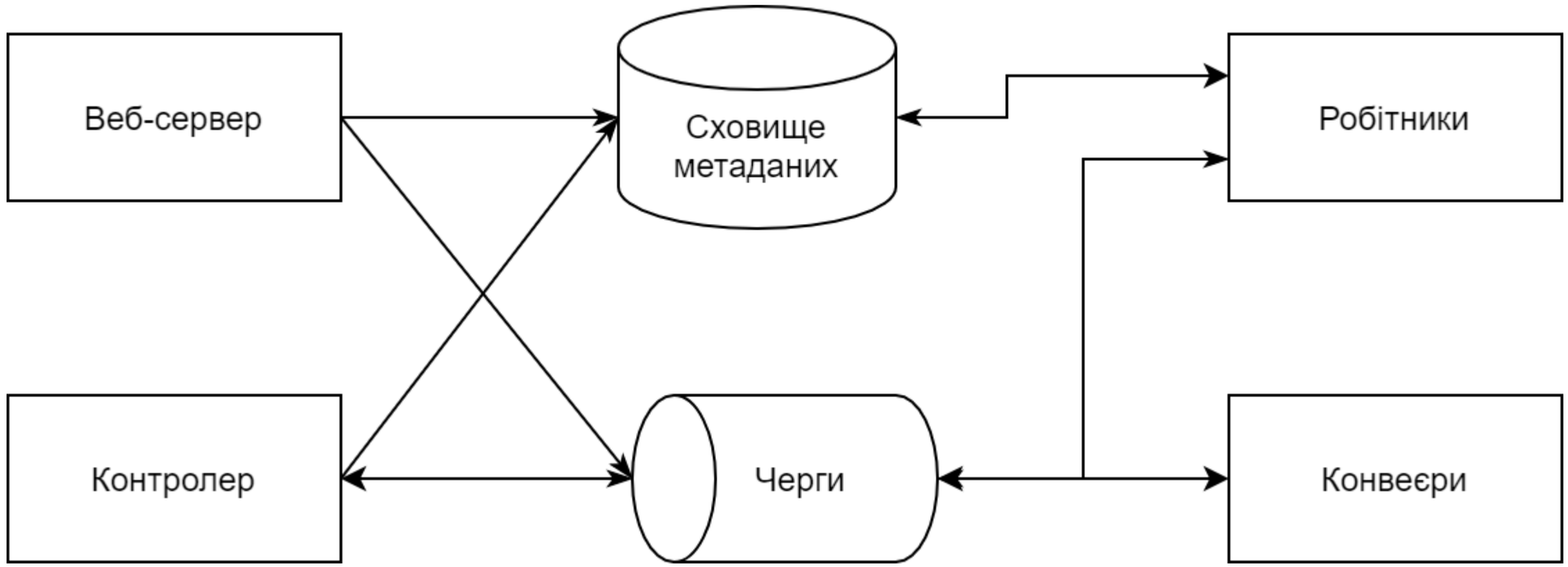
ДОДАТОК 2

Сервіс для навчання нових співробітників на підприємстві

Структурна схема – архітектура системи
ІАЛЦ.467200.005 Д2

Аркушів 1

Київ – 2021



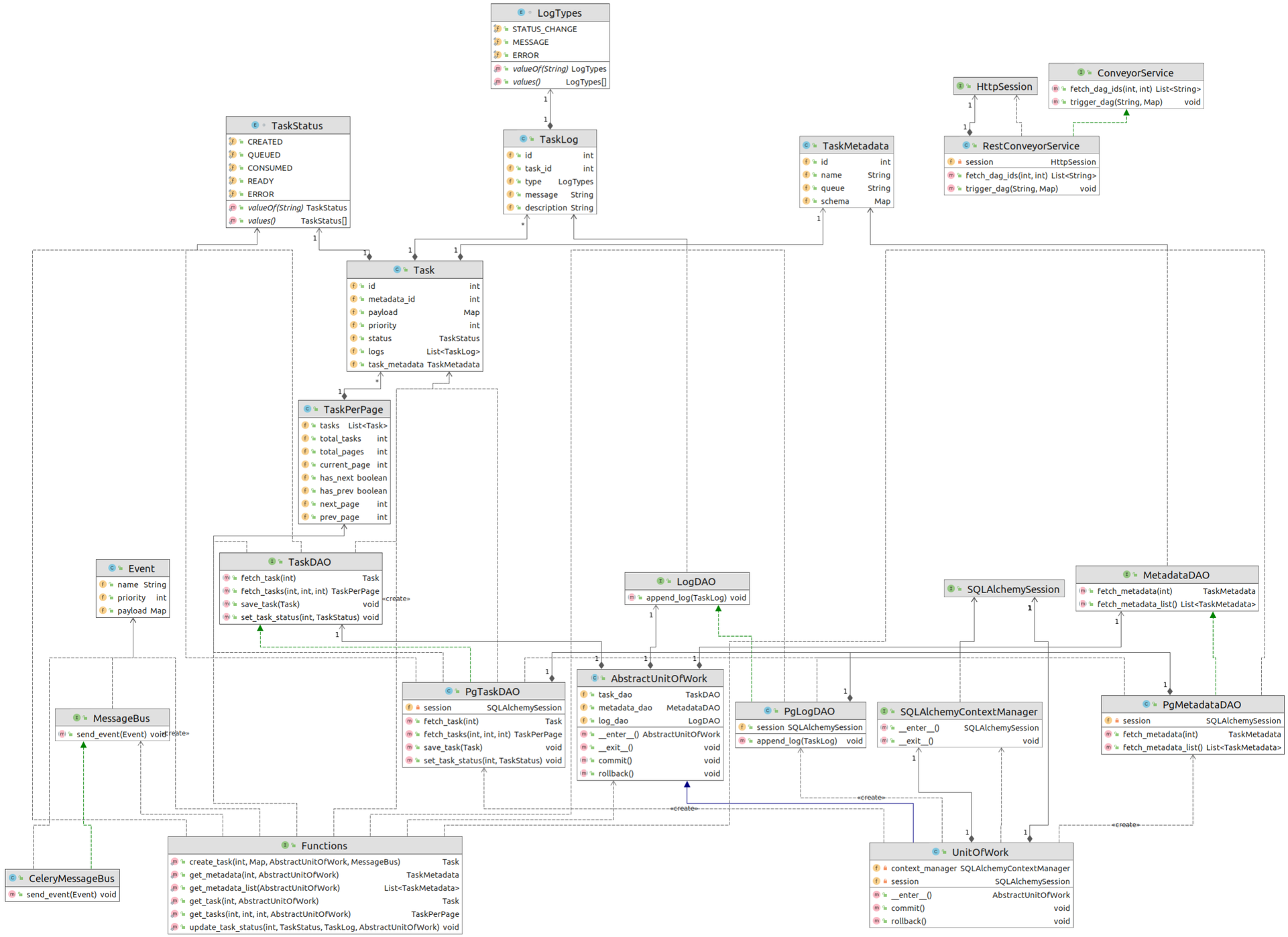
					ІАЛЦ.467200.005 Д2	
Змн	*Арк.*	*№ докум.*	*Підпис*	*Дат*	*Структурна схема. Архітектура системи*	*Літ.* / *Маса* / *Масш.*
Розроб.		Киба О. Д.				
Перевір.		Алещенко О. В.				*Аркуш* / *Аркушів*
Т. Контр.						
					Дипломна робота	НТУУ «КПІ», ФІОТ, ІВ-72
Н. Контр.		Сімоненко В. П.				
Затверд.		Стіренко С. Г.				

ДОДАТОК З
Сервіс для навчання нових співробітників на підприємстві

Функціональна схема – діаграма класів
ІАЛЦ.467200.006 ДЗ

Аркушів 1

Київ – 2021



					ІАЛЦ.467200.006 Д3	
						Літ. / Маса / Масш.
Змн	Арк.	№ докум.	Підпис	Дат	Функціональна схема. Діаграма класів	
Розроб.		Киба О. Д.				Аркуш / Аркушів
Перевір.		Алещенко О. В.				
Т. Контр.					Дипломна робота	НТУУ «КПІ», ФІОТ, ІВ-72
Н. Контр.		Сімоненко В. П.				
Затверд.		Стіренко С. Г.				

ДОДАТОК 4

Сервіс для навчання нових співробітників на підприємстві

Лістинг програми
ІАЛЦ.467200.007 Д4

Аркушів 10

Київ – 2021

domain.bl.py

```
from domain.models import *
from domain.dao.task import TasksPerPage
from domain.unit_of_work import AbstractUnitOfWork
from domain.message_bus import MessageBus


def create_task(
    metadata_id: int,
    payload: dict,
    uow: AbstractUnitOfWork,
    message_bus: MessageBus,
) -> Task:
    task = Task(metadata_id=metadata_id, payload=payload)
    uow.task_dao.save_task(task)
    uow.commit()

    payload = {**payload, "task_id": task.id}
    event = Event(task.task_metadata.name, payload)
    message_bus.send_event(event)

    return task


def send_event(event: Event, message_bus: MessageBus) -> None:
    message_bus.send_event(event)


def get_task(task_id: int, uow: AbstractUnitOfWork) -> Task:
    return uow.task_dao.fetch_task(task_id)


def get_tasks(
    page: int,
    per_page: int,
    uow: AbstractUnitOfWork,
    metadata_id: Optional[int] = None,
) -> TasksPerPage:
    return uow.task_dao.fetch_tasks(page, per_page, metadata_id)


def update_task_status(
    task_id: int,
    status: TaskStatus,
    log_message: TaskLog,
```

```
    uow: AbstractUnitOfWork,
) -> None:
    uow.task_dao.set_task_status(task_id, status)
    uow.log_dao.append_log(log_message)


def change_task_status_on_error(
    task_id: int,
    traceback: str,
    uow: AbstractUnitOfWork,
) -> None:
    log_message = TaskLog(
        task_id=task_id,
        type=LogTypes.ERROR,
        message=f'Changed task[{task_id}] status in ERROR state.',
        description=traceback,
    )
    update_task_status(task_id, TaskStatus.ERROR, log_message, uow)


def change_task_status(
    task_id: int,
    status: TaskStatus,
    uow: AbstractUnitOfWork,
    description: Optional[str] = None,
) -> None:
    log_message = TaskLog(
        task_id=task_id,
        type=LogTypes.STATUS_CHANGE,
        message=f'Changed task[{task_id}] status in {status} state.',
        description=description,
    )
    update_task_status(task_id, status, log_message, uow)


def get_metadata_list(uow: AbstractUnitOfWork) -> List[TaskMetadata]:
    return uow.metadata_dao.fetch_metadata_list()


def get_metadata(metadata_id: int, uow: AbstractUnitOfWork) -> TaskMetadata:
    return uow.metadata_dao.fetch_metadata(metadata_id)
```

domain.entities.py

```
from enum import Enum
from datetime import datetime
```

```
from dataclasses import dataclass, field
from typing import List, Optional


class TaskStatus(Enum):
    CREATED = 1
    QUEUED = 2
    CONSUMED = 3
    READY = 4
    ERROR = 5


class LogTypes(Enum):
    STATUS_CHANGE = 1
    MESSAGE = 2
    ERROR = 3


@dataclass
class TaskMetadata:
    name: str
    queue: str

    id: Optional[int] = None
    schema: dict = field(default_factory=lambda: {})

@dataclass
class TaskLog:
    type: LogTypes
    message: str

    id: Optional[int] = None
    task_id: Optional[int] = None
    description: Optional[str] = None


@dataclass
class Task:
    metadata_id: int
    id: Optional[int] = None
    payload: dict = field(default_factory=lambda: {})
    status: TaskStatus = field(default_factory=lambda: TaskStatus.CREATED)
    created_date: datetime = field(default_factory=datetime.now)
    priority: int = 1
    # relationships
```

```
    task_metadata: Optional[TaskMetadata] = None
    logs: List[TaskLog] = field(default_factory=lambda: [])


@dataclass(frozen=True)
class Event:
    name: str
    payload: dict
    priority: int = 0
```

domain.models.py

```
from sqlalchemy.dialects.postgresql import JSON

from domain.entities import *
from server.extensions import db


# TaskMetadata ORM declaration
task_metadata_table = db.Table(
    'task_metadata',
    db.Column('id', db.Integer, primary_key=True),
    db.Column('name', db.String(120), nullable=False),
    db.Column('queue', db.String(120), nullable=False),
    db.Column('schema', JSON, default=dict),
)

db.mapper(TaskMetadata, task_metadata_table)

# TaskLog ORM declaration
task_logs_table = db.Table(
    'task_logs',
    db.Column('id', db.Integer, primary_key=True),
    db.Column('task_id', db.Integer, db.ForeignKey('tasks.id'), nullable=False),
    db.Column('type', db.Enum(LogTypes), nullable=False),
    db.Column('message', db.String(120), nullable=False),
    db.Column('description', db.Text),
)

db.mapper(TaskLog, task_logs_table)

# Task ORM declaration
tasks_table = db.Table(
    'tasks',
    db.Column('id', db.Integer, primary_key=True),
    db.Column('metadata_id', db.Integer, db.ForeignKey('task_metadata.id'), nullable=False),
```

```
    db.Column('payload', JSON, default=dict),
    db.Column('status', db.Enum(TaskStatus), nullable=False, default=TaskStatus.CREATED),
    db.Column('created_date', db.DateTime, nullable=False, default=datetime.now),
)

db.mapper(
    Task,
    tasks_table,
    properties={
        'task_metadata': db.relationship(
            TaskMetadata,
            lazy='joined',
            backref=db.backref('tasks', lazy='subquery'),
        ),
        'logs': db.relationship(
            TaskLog,
            lazy='joined',
            backref=db.backref('task', lazy='joined'))
    }
)
```

domain.storage.py

```
from contextlib import contextmanager

from flask import has_app_context
from sqlalchemy import create_engine
from sqlalchemy.orm import Session, scoped_session, sessionmaker
from flask_sqlalchemy import Pagination

import settings
from domain.models import *


_scoped_session_factory = scoped_session(
    sessionmaker(bind=create_engine(settings.DATABASE_URI)),
)


def _session():
    try:
        yield db.session
    except Exception:
        db.session.rollback()
        raise
```

```
def _scoped_session():
    session = _scoped_session_factory()
    try:
        yield session
    except Exception:
        session.rollback()
        raise
    finally:
        _scoped_session_factory.remove()


@contextmanager
def session_manager() -> Session:
    if has_app_context():
        yield from _session()
    else:
        yield from _scoped_session()
```

server.namespaces.tasks.py

```
import domain.bl as bl
from domain.entities import TaskStatus

import server.namspaces.parsers as p
import server.mappers as mappers
from server.extensions import api
from server.injected_resource import InjectedResource

from tasks.app import route


tasks_namespace = api.namespace("tasks")

tasks_parser = p.tasks_pagination()
update_task_status = p.update_task_status()
create_task_model = p.create_task(tasks_namespace)


@tasks_namespace.route("")
class TasksResource(InjectedResource):

    @tasks_namespace.expect(tasks_parser, validate=True)
    def get(self):
        args = tasks_parser.parse_args()
        page, per_page, metadata_id = \
```

```
            args['page'], args['per_page'], args['metadata_id']

        with self.uow:
            pagination = bl.get_tasks(page, per_page, self.uow, metadata_id)
            return mappers.map_pagination(pagination)

    @tasks_namespace.expect(create_task_model, validate=True)
    def post(self):
        json = tasks_namespace.payload
        payload, metadata_id = json['payload'], json['metadata_id']
        with self.uow:
            task = bl.create_task(
                metadata_id,
                payload,
                self.uow,
                self.message_bus
            )

        return {"id": task.id}, 201


@tasks_namespace.route('/<int:task_id>')
class SingleTaskResource(InjectedResource):

    def get(self, task_id: int):
        with self.uow:
            task = bl.get_task(task_id, self.uow)

            if not task:
                return {"msg": "Task not found"}, 404

            return mappers.map_task(task)


    @tasks_namespace.expect(update_task_status, validate=True)
    def patch(self, task_id: int):
        args = update_task_status.parse_args()
        status, description = args['status'], args['description']
        status = TaskStatus[status]

        with self.uow:
            bl.change_task_status(task_id, status, self.uow, description)

        return {}, 204
```

tasks.controller.router.py

```
import logging
from typing import List, Optional

from celery import Celery

from tasks.app import app
from tasks.conveyors import trigger_conveyor
from domain.services.conveyor import ConveyorService


log = logging.getLogger(__name__)


def get_available_handlers(celery: Celery) -> List[str]:
    return [name for name in celery.tasks.keys()
            if not name.startswith('celery.')]


def get_workers_handler(name: str, celery: Celery) -> Optional[str]:
    handlers = get_available_handlers(celery)
    split_names = [i.split(".") for i in handlers]

    for i, n in enumerate(split_names):
        if n[-1] == name or handlers[i] == name:
            return handlers[i]


def get_conveyors_handler(name: str, service: ConveyorService) -> Optional[str]:
    names = service.fetch_dag_ids()
    for i in names:
        if i == name:
            return i


def handle_unknown(name, priority, payload) -> None:
    event = {"name": name, "priority": priority, "payload": payload}
    log.error(f"Unknown event in the route task: {event}!")


@app.task(bind=True)
def route(self, name: str, priority: int, payload: dict):
    workers_handler = get_workers_handler(name, app)

    if workers_handler:
```

```
        app.send_task(workers_handler, kwargs=payload, priority=priority)
        return

    service: ConveyorService = self.injector.get(ConveyorService)
    conveyors_handler = get_conveyors_handler(name, service)

    if conveyors_handler:
        service.trigger_dag(name, payload)
        return

    handle_unknown(name, priority, payload)
```

conveyors.dags.nasa_neo_dag.py

```
from datetime import timedelta

from airflow import DAG
from airflow.operators.python import PythonOperator
from airflow.utils.dates import days_ago
from marshmallow import Schema, fields, validates_schema, ValidationError
from domain.scrapers.nasa_neo import fetch_neo_dataset
import conveyors.dags.utils as utils

class PayloadSchema(Schema):
    task_id = fields.String(required=True)
    start_date = fields.Date(required=True)
    end_date = fields.Date(required=True)
    email = fields.Email(required=True)

    @validates_schema
    def validator(self, data, **kwargs):
        begin_date = data.get("start_date")
        end_date = data.get("end_date")

        if begin_date >= end_date:
            raise ValidationError("Begin date should be less then end date")

def run_scraping_job(templates_dict, **kwargs):
    """
    Run search scraping job for specific spider.
    """
    conf = utils.load_conf(templates_dict["conf"])

    payload_schema = PayloadSchema()
    payload = utils.validate_payload(conf, payload_schema)
```

```
    dataset = fetch_neo_dataset(payload["start_date"], payload["end_date"])

    filename = utils.create_filename(payload['task_id'])
    csv_io = utils.to_csv_io(dataset)
    utils.upload_to_s3(csv_io, filename)

def send_email_with_s3_job(templates_dict, **kwargs):
    conf = utils.load_conf(templates_dict["conf"])
    task_id, email = conf["task_id"], conf["email"]

    subject = f"Task {task_id} is ready!"
    contents = ["Dear, user!", "Your NASA NEO dataset ready.", "Please, check the attachments."]
    attachments = utils.download_from_s3(utils.create_filename(task_id))

    utils.send_email(to=email, subject=subject, contents=contents, attachments=attachments)

default_args = {
    "owner": "airflow",
    "depends_on_past": False,
    "start_date": days_ago(0),
    "retries": 5,
    "retry_delay": timedelta(minutes=5),
}
dag = DAG(
    dag_id="nasa_neo_task",
    default_args=default_args,
    description="NASA NEO scraping DAG",
    schedule_interval=None,
)
t1 = PythonOperator(
    task_id="neo_dataset_scraping",
    provide_context=True,
    python_callable=run_scraping_job,
    templates_dict={"conf": "{{dag_run.conf}}"},
    dag=dag,
)
t2 = PythonOperator(
    task_id="neo_dataset_sender",
    provide_context=True,
    python_callable=send_email_with_s3_job,
    templates_dict={"conf": "{{dag_run.conf}}"},
    dag=dag,
)
t1 >> t2
```