ФАКУЛЬТЕТ КОМП'ЮТЕРНИХ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ НАУК

Освітньо-кваліфікаційний рівень магістр
Спеціальність 121 "Інженерія програмного забезпечення"

ЗАТВЕРДЖУЮ
Завідувач кафедри КН,
_____А.В. Пукас
"____"
_____20___року

З А В Д А Н Н Я
НА ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Юзефович Ігор Михайлович

1.Тема роботи: Математичне та програмне забезпечення для підвищення безпеки виконання CI/CD процесів у середовищі Kubernetes / Mathematical and software to increase the security of CI / CD processes in the Kubernetes environment

Керівник роботи: Шевчук Р.П.

2. Строк подання студентом роботи

3. Вихідні дані до роботи: Середовище розробки – Kubernetes. Мова програмування - для серверної частини було обрано мову програмування Nodejs, для агенту встановленого в Kubernetes кластер користувача було обрано мову Go.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1. Особливості роботи у середовищі kubernetes. 2. Розробка методу інтегрування програмних продуктів у середовищі Kubernetes 3. Реалізація продукту в режимі реального часу

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень): узагальнена модель роботи системи, алгоритм взаємодії агента з сервером, алгоритм виконання задач, алгоритм прослуховування Docker демона, алгоритм логування вихідних даних контейнера, алгоритм роботи модуля, діаграма компонентів програмного модулю, схема графічного інтерфейсу, діаграма активності алгоритму запуску пайплайну та діаграма активності алгоритму отримання логів.

РЕЗЮМЕ

Кваліфікаційна робота містить 75 сторнки, 49 рисунків, 5 таблиць, 3 додатки та 20 джерел в переліку посилань.

Метою кваліфікаційної роботи є підвищення рівня безпеки в роботі CI/CD рішень за рахунок використання Kubernetes як середовища виконання пайплайнів.

Об'єктом дослідження – процес безперервної інтеграції та розгортання програмного забезпечення.

Предметом дослідження – методи та засоби виконання безпечних CI/CD процесів у середовищі Kubernetes.

Одержані висновки та їх новизна: Було розроблено архітектуру програмної системи та модель виконання пайплайну у приватному кластері користувача, які дозволили запобігти поширенню конфіденційних даних з сервером, у випадку коли пайплайн повинен мати до них доступ. Це було реалізовано за допомогою агента, який повинен бути встановленим у кластері користувача.

Ключові слова: пайплайн, попередження, програмний продукт, серверна частина, Kubernetes, Nodejs, Go, кваліфікаційна робота.

RESUME

Qualification work contains 75 pages, 49 drawings, 5 tables, 3 applications and 20 source in the list of references.

The purpose of the qualification work is to increase the level of security in the operation of CI/CD solutions due to the use of Kubernetes as a pipeline execution environment.

The object of research is the process of continuous integration and deployment of software.

The subject of the study is methods and means of performing secure CI/CD processes in the Kubernetes environment.

Received conclusions and their novelty: The architecture of the software system and the model of the execution of the pipeline in the private cluster of the user were developed, which made it possible to prevent the distribution of confidential data with the server, in the case when the pipeline should have access to them. This was implemented using an agent that must be installed in the user's cluster.

Keywords: pipeline, warning, software product, server part, Kubernetes, Nodejs, Go, qualification work.

ЗМІСТ

ВСТУП 4
РОЗДІЛ І АНАЛІЗ СУЧАСНОГО СТАНУ ПИТАННЯ ТА ОБҐРУНТУВАННЯ ЗАДАЧІ 9
1.1. Аналіз предметної області 9
1.2. Аналіз інструментів для автоматизації процесу CI/CD 11
1.3. Аналіз методів виконання кроків у середовищі Kubernetes 17
1.4. Аналіз методів зчитування вихідних даних кроків 19
1.5. Постановка задачі дослідження 21
Висновки до розділу І 21
РОЗДІЛ ІІ РОЗРОБКА АРХІТЕКТУРИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА МЕТОДУ ІНТЕГРУВАННЯ ПРОГРАМНИХ ПРОДУКТІВ У СЕРЕДОВИЩІ KUBERNETES 22
2.1. Аналіз інформаційного забезпечення 22
2.2. Розробка розподіленої архітектури програмного забезпечення 23
2.3. Розробка алгоритму взаємодії сервера й Kubernetes кластера 25
2.4. Розробка алгоритму виконання пайплайну в середовищі Kubernetes 27
2.5. Розробка методу зчитування вихідних даних контейнерів 29
2.6. Розробка структури графічного інтерфейсу 33
Висновки до розділу ІІ 34
РОЗДІЛ ІІІ РОЗРОБКА ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ДЛЯ ПІДВИЩЕННЯ БЕЗПЕКИ ВИКОНАННЯ CI/CD ПРОЦЕСІВ У СЕРЕДОВИЩІ KUBERNETES 35
3.1. Варіантний аналіз і обґрунтування вибору засобів реалізації програмного забезпечення 35
3.2. Розробка графічного інтерфейсу користувача 45
3.3. Розробка модуля прослуховування та логування вихідних даних кроків 47
3.4. Тестування програмного забезпечення 62
Висновки до розділу ІІІ 71
ВИСНОВКИ ТА ПРОПОЗИЦІЇ 73
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 74
ДОДАТОК А - ДЕКЛАРАЦІЯ ДОБРОЧЕСНОСТІ 76
ДОДАТОК Б - ЛІСТИНГ ОСНОВНИХ МОДУЛІВ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 77
ДОДАТОК В - КОПІЯ ПУБЛІКАЦІЇ 90

ВСТУП

Актуальність роботи. На сьогоднішній день швидкість з якою розробляються програмні продукти стає все більшою й більшою. Разом з цим зростає кількість проектів, ідей, які потребують якнайшвидшого вирішення. Є багато методологій й підходів до розробки програмного забезпечення, які намагаються вирішити ці проблеми, але часто проблема так і залишається на стороні розробки, а не її планування. Одним з найважливіших етапів розробки програмного продукту є розгортання програмного забезпечення. Це операції, які роблять програмну систему готовою до використання. Цей процес є частиною життєвого циклу програмного забезпечення.

Загальний процес розгортання складається з декількох взаємопов'язаних дій з можливими переходами між ними. Ці дії можуть відбуватися на стороні розробника або на стороні споживача або обох. Оскільки кожна програмна система є унікальною, точні процеси чи процедури в межах кожної діяльності важко визначити. Тому "розгортання" слід тлумачити як загальний процес, який повинен бути налаштований відповідно до конкретних вимог або характеристик.

Етап розгортки займає велику частину часу відносно безпосередньої розробки, що передбачає необхідність оптимізації. Найпростішим рішенням було створення скриптів, які в якійсь мірі автоматизували цей процес: не потрібно вручну збирати проекти, забезпечувати переміщення між машинами, видаляти попередні версії та встановлювати нові. Всі перелічені дії можна оптимізувати за допомого звичайних скриптів. Виявилася необхідність в особі, хто буде займатися запуском та налаштуванням цим скриптів. Після цього прийшла черга автоматизувати процес запуску скриптів, а саме визначити умови за яких це має ставатися та вхідні дані. Таким чином були створені перші CI/CD рішення.

CI/CD (continuous integration/ continuous deployment, безперервна інтеграцій/безперервне розгортання) – процес постійної інтеграції та розгортання [1].

Безперервну інтеграцію можна підсумувати так: ви хочете, щоб усі частини програмного засобу, що розробляється, надходили до одного й того ж самого місця, й проганялися через ті самі процеси в результаті чого був отриманий звіт, опублікованими у легкодоступному місці.

Найпростіший приклад безперервної інтеграції - пуш всього коду програми в єдиний репозиторій. Незважаючи на все це, це може здатися незручним, мати єдине місце, де ви «інтегруєте» весь свій код, але це є основою для розширення інших, більш досконалих практик.

Після того, як код прибув до репозиторія, є можливість запустити деякі процеси в цьому сховищі кожного разу, коли щось змінюється. Це може включати:

- запуск автоматичного сканування якості коду з генеруванням звіту про те, наскільки добре останні зміни дотримуються належних методів кодування;
- збирання проекту й запуск усіх можливих автоматизованих тестів, щоб переконатися, що ваші зміни не порушили жодної функціональності;
- створення та публікація звіту про покриттю коду тестами, щоб отримати уявлення про те, наскільки ретельними є автоматизовані тести.

Але побудування процесу безперервної інтеграції задача не із легких. Її організацією як правило займаються виключно фахівці в сфері DevOps. Організація відбувається у декілька етапів:

- отримання початкового коду з репозиторію;
- складання проекту;
- виконання тестів;
- розгортання готового проекту;
- відправлення звітів.

Для упорядковування й організації всіх кроків перелічених вище їх об’єднують в «пайплайни»(від англ. pipeline – трубопровід). Пайплайни складаються з кроків. Кожен крок являється виконанням вказаної користувачем команди в зазначеному ним середовищі. Для того що надати такий рівень свободи дій користувачу, в системах безперервної інтеграції використовуються віртуальні машини. Тобто кожен крок – це інструкція того, що буде зроблено в середині віртуальної машини. Але оскільки віртуальна машина це досить «важка» річ й потребує велику кількість ресурсів комп’ютера, розробники вирішили змінити їх контейнерами – полегшеною версією віртуальних машин, яка працює на основі ядра Linux [2].

Оскільки процес створення CI/CD рішень є важкою, до кінця не дослідженою й не тривіальною задачею було вирішено дослідити це питання й розробити свій прототип системи інтеграції програмного продукту в режимі реального часу.

*Мета та завдання дослідження***.** Метою роботи є підвищення рівня безпеки в роботі CI/CD рішень за рахунок використання Kubernetes як середовища виконання пайплайнів.

Основними задачами дослідження є:

- провести аналіз існуючих рішень і засобів з автоматизації інтеграції та розгортання програмного коду;
- розробити розгалужену архітектуру програмної системи;
- розробити алгоритм взаємодії сервера й Kubernetes кластера;
- розробити алгоритм виконання пайплайну в середовищі Kubernetes;
- розробити алгоритм збереження вихідних даних кроків;
- розробити інтерфейс програмного продукту;
- розробити систему інтеграції програмного продукту в режимі реально часу.

Об’єкт дослідження – процес безперервної інтеграції та розгортання програмного забезпечення.

Предмет дослідження – методи та засоби виконання безпечних CI/CD процесів у середовищі Kubernetes.

Методи дослідження. У процесі досліджень використовувались: методи проектування архітектури програмного забезпечення, методи побудови розподілених інформаційних систем.

Наукова новизна отриманих результатів.

1. Подальшого розвитку отримав метод виконання процесів безперервної інтеграції та розгортання за допомогою середовища виконання Kubernetes, який на відміну від існуючих передбачає використання додатковий модуля-агента, який встановлюється у кластер користувача та підтримує з’єднання з CI/CD платформою шляхом постійного опитування на предмет наявності нових «задач», які повинні бути виконані у кластері користувача. В результаті, з'являється можливість для користувача з власним Kubernetes кластером використовувати його у якості середовища виконання, навіть, якщо цей кластер знаходиться у приватній мережі, на відміну від аналогів які мають прямий доступ до середовища виконання. Як наслідок, підвищується рівень безпеки в роботі CI/CD рішень, оскільки з'являється можливість не розголошувати секрети платформі під час виконання пайплайну, а зберігати їх у власному кластері, що значно зменшує ризики їх несанкціонованого поширення.

2. Подальшого розвитку отримав метод зчитування вихідних даних контейнерів, який відрізняється від відомих у можливості роботи з багатьма контейнерами одночасно й створенню мультиплексованого потоку даних, що дозволяє протоколювати вихідні дані контейнерів, запущених за допомогою команди docker-compose, в один вихідний потік.

Практична цінність отриманих результатів. Практична цінність одержаних результатів полягає в тому, що на основі розробленої в кваліфікаційній роботі методу інтегрування програмних продуктів у середовищі Kubernetes було розроблено програмне забезпечення безпечного виконання пайплайнів.

Особистий внесок здобувача. Усі наукові результати, викладені у магістерській кваліфікаційній роботі, отримані автором особисто. У роботі автору належить ідея та реалізації CI/CD платформи з використанням Kubernetes як середовища виконання пайплайнів, що дозволить покращити продуктивність, безпеку та моніторинг процесів що відбуваються під час виконання пайплайну.

Апробація результатів. Основні положення та результати кваліфікаційної роботи обговорювались на семінарі «Комп'ютерні інформаційні технології», що проходив 29 листопада 2022 року в м. Тернопіль.

Публікації.

Шевчук Р.П., Юзефович І.М. Підвищення безпеки процесів безперервної інтеграції та безперервної доставки коду у середовищі Kubernetes // Матеріали школі-семінару молодих вчених і студентів «Комп'ютерні інформаційні технології» . — Тернопіль : ФО-П Шпак В.Б., 2022.

РОЗДІЛ І
АНАЛІЗ СУЧАСНОГО СТАНУ ПИТАННЯ ТА ОБҐРУНТУВАННЯ ЗАДАЧІ

1.1. Аналіз предметної області

В даний час у розробці веб-додатків як підтипу програмного забезпечення існує чітка тенденція до прискорення циклу розробки: кодування - тестування - збирання - розгортання. Кінцевим результатом циклу є новий випуск програмного забезпечення. Дуже часто команді розробників доводиться випускати нові релізи кілька разів на день. У свою чергу, всі описані в циклі процеси займають відносно багато часу та є досить рутинними. Крім того, ми можемо згадати людський фактор, і незначне відхилення від процедури може призвести до помилок у новому релізі програмного продукту, на виявлення причини якого згодом буде потрачено ще більше часу.

Ця проблема може бути вирішена шляхом введення безперервної інтеграції (CI) у процес розробки. Це практика автоматичного виконання етапів збирання й розгортування необмежену кількість раз, що прагне до однакових результатів з періодичним виконанням.

На сьогоднішній день CI / CD є найбільш широко використовуваної технологією в розробці програмного забезпечення, і розробники прагнуть застосовувати її практично у всіх завданнях. Все, що потрібно – це довести існування статусу «золотого стандарту» і «водоспаду».

Методологія почала активно використовуватися паралельно з повсюдним поширенням гаджетів, коли кількість всіх видів програмного забезпечення для них значно збільшилася. Кожен день, якщо не кожну годину, створюються різні клієнтські програми. Подивіться на динаміку росту кількості додатків на платформах для дистрибутиву за останні кілька років. В 2018 році кількість додатків в Google Play збільшилося на третину,

склавши 3,6 мільйона. Все це не могло не привести до зміни глобальних критеріїв швидкості створення програмного забезпечення як такого.

Наразі вже існує ціла низка рішень для впровадження методології постійної інтеграції. Їх є багато видів, починаючи від найпримітивніших і закінчуючи прогресивними сучасними платформами з великою кількістю можливостей. На ринку CI/CD рішень наразі панує величезна конкуренція, велика кількість застарілих рішень намагаються втримати клієнтів й не надавати їм причин дивитися в сторону більш сучасних платформ. Останні роки є особливо визначними для побільшої долі такого роду програмних засобів, адже стандартом «де-факто» для production оточення стає запуск додатків в контейнерах. А контейнери в свою чергу повинні існувати в межах інструменту для їх оркеструванням, найпопулярніший на сьогоднішній день є Kubernetes[3][4].

Що важливо для користувача й для чого це взагалі йому потрібно? Йому важливо щоб використання інструменту було зручним і не поступалося у функціональності вже доступним рішенням. А необхідність в такого роду рішень набуло попиту з розвитком контейнеризації та CI/CD.

Є багато готових рішень (CircleCI, TravisCI, DroneIO, Gitlab, Codefresh), які можуть покрити низку проблем даної галузі, але їх найбільша проблема – це ціна. Тому розробники все частіше намагаються створити щось «своє» з метою заміни, вже існуючих рішень. Також деякі з них є досить повільними через використання вже застарівших віртуальних машин в якості ізольованого середовища виконання кроків.

Створення такого роду платформи це не легка задача, це ціла низка надзвичайно складних програмних компонент, які мають працювати разом. Одною із найважчих в реалізації компонент, є логування вихідних даних контейнера. На разі не було знайдено компоненти/модуля який би зміг усунути цю складність під час розробки.

Розроблювана програмна система зможе усунути цю дірку, покращити швидкість виконання пайплайнів та надати ентузіастам зручний шаблон й

інтуїтивно зрозумілий інтерфейс для подальшого покращення й створення більш складніших рішень.

Головною перевагою програмної системи буде швидкість в роботі, можливість запускати пайплайни, прослуховувати вихідних дані контейнерів, та показувати їх користувачу.

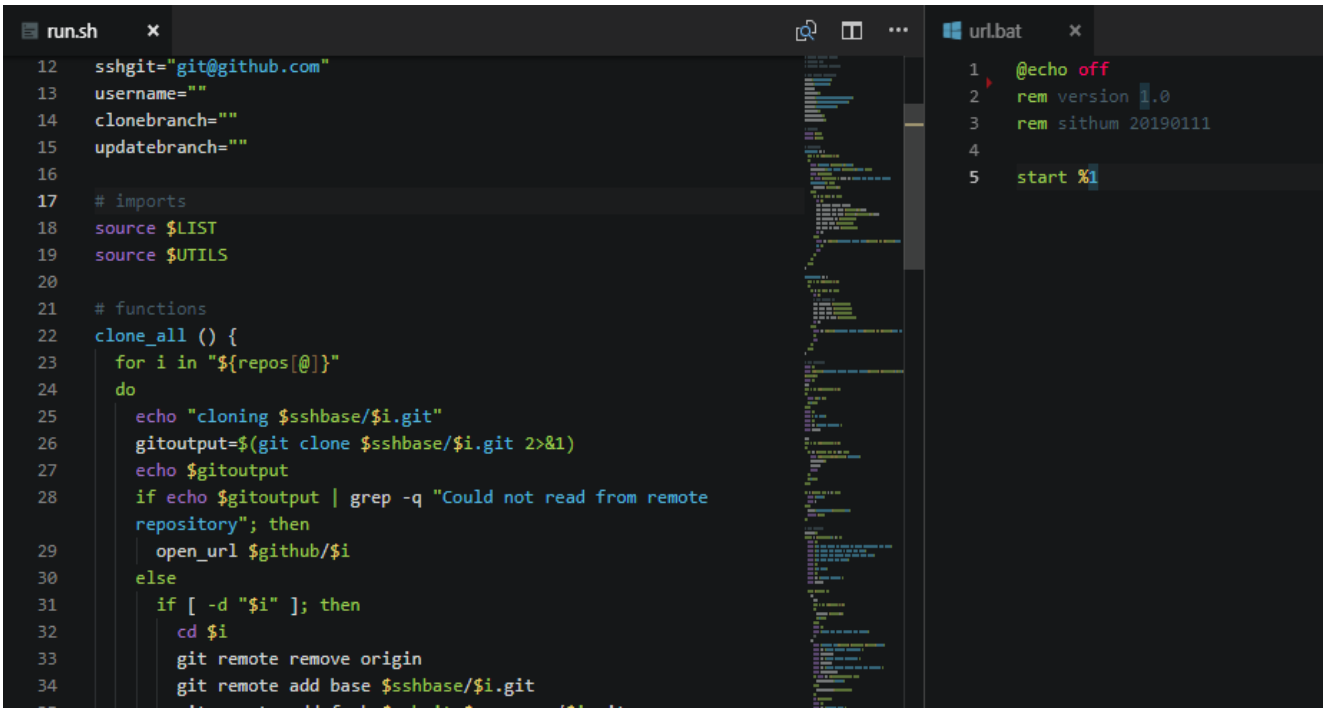
1.2. Аналіз інструментів для автоматизації процесу CI/CD

Розглянемо декілька інструментів інтеграції програмного продукту в режимі реального часу, які були, або залишилися популярними серед розробників.

Автоматизація шляхом написання скриптів (рисунок 1.1) – найпримітивніший з перечислених способів й інструментів. Раніше автоматизація процесів розгортування була виконана у досить прямий спосіб. Програміст писав код, який в якійсь мірі міг автоматизувати процес розгортування, але дане рішення залишалося досить проблемним довгий час.

На це є декілька причин:

- зав’язаність на спеціалісті, який автоматизував процес розгортування. Часто така автоматизація є не легкою задачею й стає проблемно розібратися у всіх рішеннях прийнятих попереднім програмістом. Іншим словом це можна перефразувати у складність в імплемінтації;
- необхідність адаптації скриптів відповідну до проекту й оточення в якому відбувається його розгортування. У випадку з написання скриптів автоматизації власноруч, дуже важко уникнути ситуації коли доведеться модифікувати й адаптувати його під певну ситуацію;
- відсутність екосистеми з низкою додаткових функцій, які надаються «з коробки», як у випадку з повноцінними CI/CD платформами. Мова йдеться про такі життєво необхідні на сьогоднішній день механізми як відмовостійкість, безпечність, інтеграції з зовнішніми сервісами.



.

Рисунок 1.1 – Приклад власноруч написаного скрипта автоматизації розгортування

Hudson (рисунок 1.2) – це система для реалізації безперервної інтеграцією (CI), написана на Java, яка працює на сервлет контейнерах, таких як Apache Tomcat або Glassfish. Він підтримує інструменти SCM, включаючи CVS, Subversion, Git, Perforce, Clear Case і RTC, і може реалізовувати проекти на основі Apache Ant і Apache Maven, а також довільні сценарії оболонки і пакетні команди Windows [5].

Hudson може бути розширений за допомогою архітектури плагінів. Багато плагінів стали загальнодоступними, розширивши його за межі просто інструменту побудови проектів Java. Плагіни доступні для інтеграції з Hudson в більшій мірі реалізують інтеграції з системами контролю версій та базами даних. Багато інструментів побудови підтримуються за допомогою відповідних плагінів. Плагіни також можуть змінювати вигляд та поведінку Хадсона або додавати нові функції.

Hudson є більш сучаснішим інструментом, але не позбавленого проблем:

- Налаштування має бути більш зручним для користувачів. На даний момент користувачі, які вперше користуються системою не розуміють як розпочати роботу.
- Відсутність підтримки від розробників

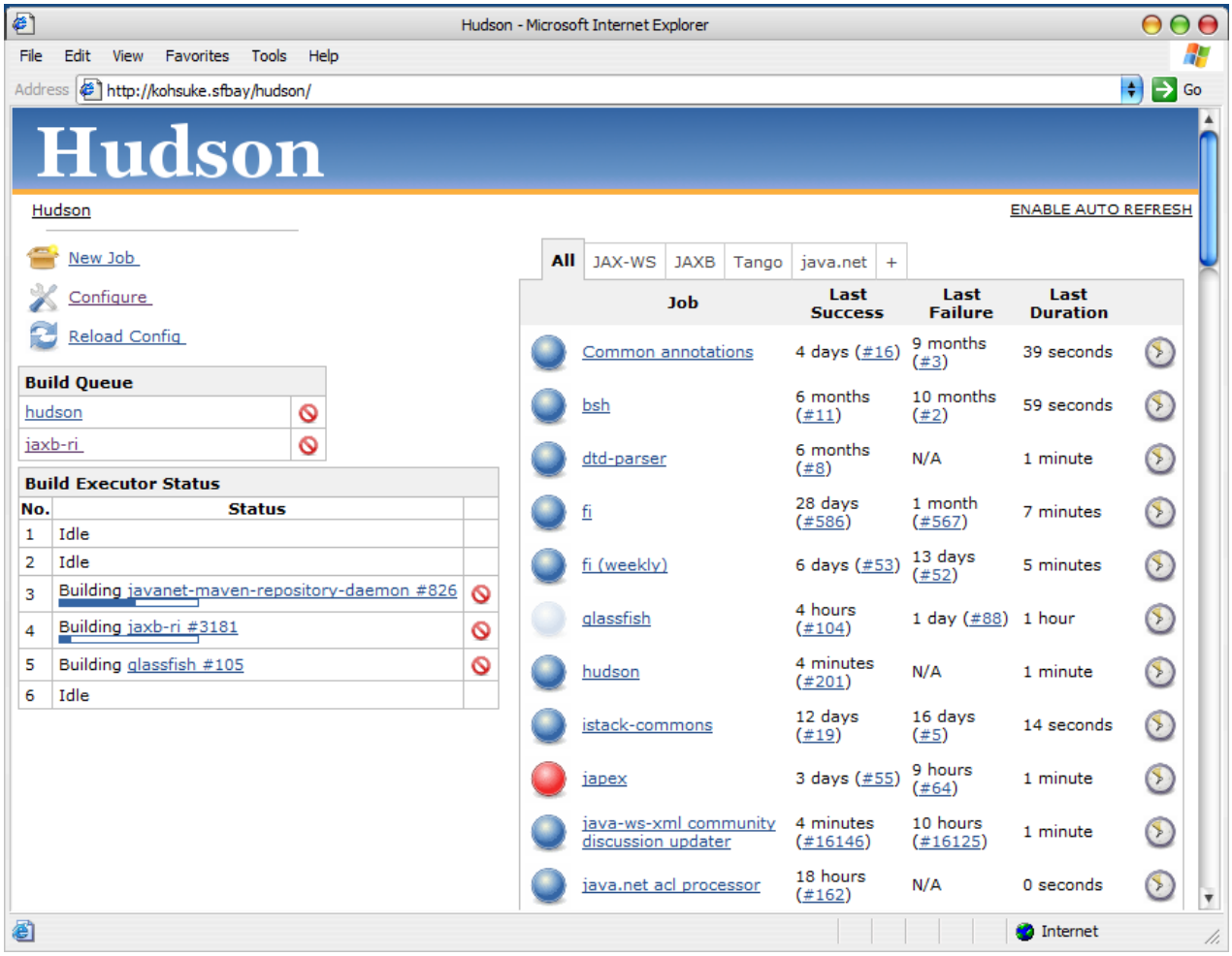


Рисунок 1.2 – Приклад використання Hudson

Jenkins (рисунок 1.3) – це сервер автоматизації з відкритим кодом з неперевершеною екосистемою плагінів, який підтримує практично кожен інструмент як частину пайплайнів [6]. Він є нащадком Hudson.

Jenkins пропонує простий спосіб налаштування середовища безперервної інтеграції або безперервної розгортки (CI / CD) для майже будь-якої комбінації мов та сховищ вихідного коду за допомогою конвеєрів, а також автоматизацію інших завдань рутинної розробки. У той час як Jenkins не усуває необхідності створювати сценарії для окремих кроків, це дає

швидший і надійніший спосіб інтегрувати весь ланцюжок інструментів побудови, тестування та розгортання, що ви можете легко створити самостійно.

Даний інструмент поширюється як архів WAR і як інсталяційний пакет для основних операційних систем: як пакет Homebrew, зображення Docker та вихідний код. Вихідний код здебільшого Java, з кількома файлами Groovy, Ruby та Antlr.

Jenkins досягає високої якості та зручності безперервної інтеграції за допомогою плагінів. Плагіни дозволяють інтегрувати різні етапи DevOps. Якщо ви хочете інтегрувати певний інструмент, вам потрібно встановити їх плагіни. Наприклад: Git, Maven, Amazon EC2, середовища виконання тестів, тощо.

Недоліки Jenkins:

- проблеми з повідомленнями про помилки;
- важке початкове налаштування;
- незручний інтерфейс;
- незручна документація.

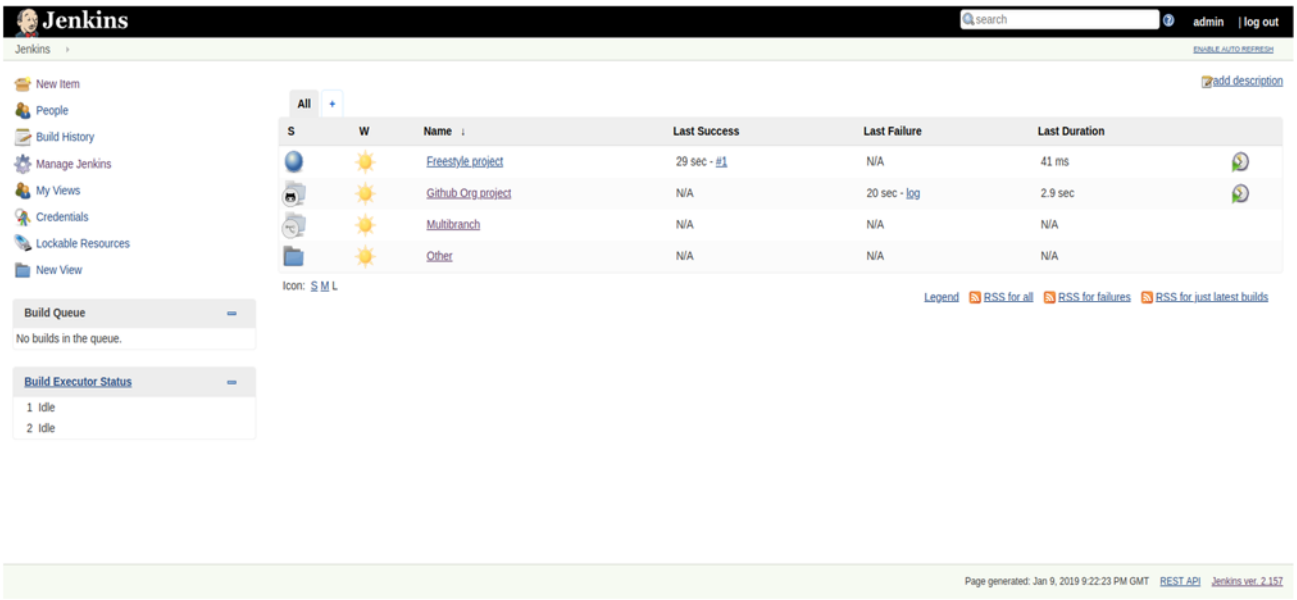


Рисунок 1.3 – Приклад використання Jenkins

CircleCI (рисунок 1.4) – це потужний CI-продукт на основі SaaS, який дозволяє інтеграцію та розгортання у хмарі. Система конфігурації на основі YAML дозволяє окремим розробникам випробовувати свої зміни та ефективно й швидко просувати їх далі у процесі інтеграції. Різноманітність підтримуваних платформ дозволяє декільком людям централізуватися на одному рішенні, щоб уникнути поширення на інші продукти.

Велика свобода конфігурації для користувача робить його придатним для будь-яких видів програмних продуктів. Покроковий перегляд виконання задач є дуже корисним інтерфейсом для відстеження загального стану процесу. Має потужні функції, такі як паралельне виконання кроків.

Недоліки CircleCI:

- незручна конфігурація пайплайнів за допомогою YAML синтаксису;
- швидкість роботи;
- ціна.

Рисунок 1.4 – Приклад використання CircleCI

Проаналізувавши усі аналоги, визначено їхні можливості та недоліки, які враховувались при створенні власного прототипа програмної системи інтеграції програмного продукту в режимі реального часу (табл. 1.1).

Таблиця 1.1 –Порівняльні характеристики доступних рішень

Критерій	Скрипти	Hudson	Jenkins	CircleCI	Власний прототип
Швидкість виконання процесу інтеграції	+	-	+	+	+
Екосистема плагінів	-	+	+	+	+
Можливість виконання в межах власної інфраструктурі	+	-	+	-	+
Вибір місця збереження зчитаних вихідних даних контейнерів під час виконання процесу інтеграції	+	+	-	+	+
Можливість роботи з Docker контейнерами	+	-	+	+	+
Зрозумілий формат YAML файлу	-	-	-	-	+

Проаналізувавши аналоги доступних рішень було вирішено зробити акцент на швидкість виконання пайплайнів, зручність їх опису в YAML форматі та можливість подальшої кастомізації та розширення.

Таблиця порівняльних характеристик показала, що розробка програмної системи інтеграції програмного продукту в режимі реального часу є доцільною. В результаті отримаємо систему, що покриває недоліки існуючих рішень й може слугувати заміною більшості з них.

1.3. Аналіз методів виконання кроків у середовищі Kubernetes

У якості середовища виконання кроків будо обрано Kubernetes так як воно ідеально підходить для вирішення наступних проблем:

- жорстке управління доступом;
- стійкість до відмов;
- гнучкість у виборі додаткових інструментів (наприклад сховище даних);
- горизонтальна та вертикальна масштабованість;
- багато-кластерна конфігурація;
- управління конфігурація та секретами.

Кожен крок буде представлений конфігурацією, обов’язковим полем якої є image. На основі цієї інформації потрібно буде створити контейнер, в якому будуть виконуватися команди користувача. В Kubernetes це робиться за допомогою Pod.

Pods — це найменші, найпростіші об’єкти, які можна розгортати в Kubernetes. Pod являє собою один екземпляр запущеного процесу у вашому кластері [7].

Pod — це група з одного або кількох контейнерів із спільним сховищем та мережевими ресурсами, а також специфікацією для запуску контейнерів. Вміст Pod завжди розміщено разом і планується спільно, і працює в спільному контексті. Pod моделює специфічний для програми «логічний хост»: він містить один або кілька контейнерів програми, які відносно тісно пов’язані. У нехмарному контексті програми, що виконуються на одній фізичній або віртуальній машині, аналогічні хмарним додаткам, що виконуються на тому ж логічному хості.

Pod в кластері Kubernetes використовуються 2 способами:

- Pod, які запускають один контейнер. Модель «один контейнер на Pod» є найпоширенішим випадком використання Kubernetes; у цьому випадку ви можете уявити Pod як обгортку навколо окремого

контейнера; Kubernetes керує Pods, а не безпосередньо керує контейнерами.

- Поди, які запускають кілька контейнерів, які повинні працювати разом. Pod може інкапсулювати програму, що складається з кількох спільно розташованих контейнерів, які тісно пов’язані між собою та потребують спільного використання ресурсів. Ці спільно розташовані контейнери утворюють єдину цілісну одиницю сервісу — наприклад, один контейнер обслуговує дані, що зберігаються в спільному томі, для загального доступу, тоді як окремий контейнер оновлює або оновлює ці файли. Pod об'єднує ці контейнери, ресурси сховища та недовговічний мережевий ідентифікатор разом як єдиний блок.

Pods призначені для підтримки кількох взаємодіючих процесів (як контейнери), які утворюють єдину одиницю обслуговування. Контейнери в Pod автоматично розташовуються і плануються спільно на одній фізичній або віртуальній машині в кластері. Контейнери можуть ділитися ресурсами та залежностями, спілкуватися один з одним і координувати, коли і як вони припиняються.

У одному поді може бути запущено декілька контейнерів, що виконуються на одному Docker демоні. Це дозволяє контейнерам взаємодіяти з Docker API. На основі цього було прийнято рішення реалізувати прослуховування вихідних даних контейнерів шляхом створення Pod таким чином, що поруч з основним контейнером (кроком) буде виконуватися контейнер слухач, задачею якого буде отримання логів контейнера за допомогою Docker API та надсилання їх до відповідного сховища.

Прикладом такого Pod’а може буде наступна конфігурація (рис. 1.5):

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: step-name
spec:
  containers:
    - name: step-name
      image: user-provided-image
  containers:
    - name: listener
      image: logs-listener
      volumeMounts:
        - name: dockersock
          mountPath: "/var/run/docker.sock"

  volumes:
  - name: dockersock
    hostPath:
      path: /var/run/docker.sock

```

Рисунок 1.5 – Мінімальна конфігурація коду

1.4. Аналіз методів зчитування вихідних даних кроків

Методів розв’язання зчитування вихідних даних не так багато, це через те що ми обмежені програмним інтерфейсом, який надає Docker. Виходячи з цього потрібно вивчити документацію й зрозуміти, які інструменти варто використати для вирішення задачі зчитування з контейнера.

В основі будь-якої реалізації буде зчитування з стандартних потоків даних контейнера: stdout та stdin. Читання з цих двох потоків надасть повну картину того що відбувається в середині контейнера.

Стандартні потоки введення/виведення в системах типу UNIX (і багато інших) [8] — потоки процесу, що мають номер (дескриптор), зарезервований для виконання деяких «стандартних» функцій. Як правило (хоча і не обов'язково), ці дескриптори вже відкриті в момент запуску завдання.

В системі за замовчуванням завжди відкриті три "файли" - stdin (клавіатура), stdout (екран) і stderr (повідомлення про помилки на екран) [9].

Ці, та будь-які інші відкриті файли, можуть бути перенаправлені. В реалізації програмного модуля Loggio буде використано stdout та stderr.

stdout – націлений на запис на пристрій виведення (монітор). Командна оболонка UNIX (і оболонки інших систем) дозволяють скерувати цей потік за допомогою символу «>». Для виконання програм у фоновому режимі цей потік зазвичай переводять у файл.

stderr – зарезервовано для виведення діагностики та повідомлень налаштування в текстовому вигляді. Частіше за все вивід цього потоку збігається з stdout, однак, на відміну від нього, місце призначення потоку stderr не змінюється при перепризначенні «>» і створенні конвеєрів ("|").

Часто ці потоки перенаправляються в різні місця, для того щоб розділити помилки від інформації про виконання програми, але у нашому випадку це не доцільно, так як метою є створення єдиного потоку вихідної інформації контейнера. Вище приведені потоки будуть зчитуватися й зливатися в єдиний. Таким чином матимемо єдиний потік із всією вихідною інформацією контейнера. Це надає змогу без проблем реалізувати запис в вибране користувачем сховище, що є одною з ключових вимог до розроблюваного програмного модуля.

Docker API надає такі методи для роботи з вихідними даними контейнерів:

logs – отримує вихідні дані, присутні на час виконання stdout та strerr;

attach – прикріплюється до контейнера й відкриває потік в який потрапляє все що записується в stdout та strerr потоки.

Оскільки планується отримання всіх вихідних даних, а не тільки тих які присутні під час виклику метода, буде використано attach метод, що дозволяє прослуховувати стандартні покоти виводу контейнера, а також отримувати попередньо записані логи використовуючи опцію logs.

1.5. Постановка задачі дослідження

Після аналізу питання створення програмної системи інтеграції в режимі реального часу та модуля для зчитування вихідних даних контейнерів, було визначено наступні завдання, які необхідно виконати для розробки програмного продукту:

- провести аналіз існуючих рішень і засобів з автоматизації інтеграції та розгортання програмного коду;
- розробити розгалужену архітектуру програмної системи;
- розробити алгоритм взаємодії сервера й Kubernetes кластера;
- розробити алгоритм виконання пайплайну в середовищі Kubernetes;
- розробити алгоритм збереження вихідних даних кроків;
- розробити інтерфейс програмного продукту;
- розробити систему інтеграції програмного продукту в режимі реально часу.

Висновки до розділу I

Під час аналізу стану питання і постановки задачі було розглянуто стан рішень для реалізації інтеграції в режимі реального часу на сьогоднішній день. Також було проаналізовано стан даного питання шляхом розгляду аналогів (Скрипти автоматизації, Hudson, Jenkins, CircleCI), та їх порівняння між собою, та розроблюваною програмною системою інтеграції програмного продукту в режимі реального часу. В результаті порівняння було відображено доцільність розробки програмної системи та проаналізовано можливі методи по вирішенню питання. Було обґрунтовано вибір мов програмування, які будуть використовуватися при розробці програмного модуля та наведено основні їх переваги.

РОЗДІЛ ІІ
РОЗРОБКА АРХІТЕКТУРИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА МЕТОДУ ІНТЕГРУВАННЯ ПРОГРАМНИХ ПРОДУКТІВ У СЕРЕДОВИЩІ KUBERNETES

2.1. Аналіз інформаційного забезпечення

Структура інформаційної системи досить складна і включає різні комбінації інформаційних структур, що мають ієрархічну структуру побудови.

Для реалізації системи необхідно обрати технології, які в повній мірі зможуть задовольнити технічні та функціональні вимоги модулів.

Інформаційне забезпечення - це сукупність видів документів, нормативної бази та втілених рішень щодо обсягу, місця розташування та форм організації інформації, що циркулює в системі автоматизованої обробки інформації або в інформаційній системі [10].

Основними вимогами до інформаційного забезпечення є: цілісність, надійність, контроль, захист від несанкціонованого доступу, єдність та гнучкість, стандартизація та уніфікація, адаптованість, мінімізація помилок введення та виведення інформації.

Інформаційне забезпечення складається з таких частин: методичні та шаблонні матеріали, система розподілу та кодування, інформаційна база.

Основою інформаційного забезпечення є інформаційна база (ІБ). За складом та змістом вона повинена відповідати вимогам тих завдань, проектувати ті системи, які вирішуються на її основі. Відповідно до сфери експлуатації існують позамашинні та внутрішньомашинні ІБ.

Внутрішня інформаційна база - це частина інформаційної бази, яка представляє собою сукупність даних про машинні носії, що використовуються в інформаційній системі.

Інформаційна база даних зовнішніх машин - це частина, яка представляє собою набір повідомлень, сигналів та документів, призначених для безпосереднього сприйняття людиною без використання комп'ютерних технологій. Він складається з вхідних, вихідних та регуляторних повідомлень.

Програми пишуться для вирішення проблем. Щоб вирішити проблему, програмі потрібні вхідні дані.

Дані можна вводити різними способами:

- написати у коді програми. Це називається жорстким кодуванням;
- користувачем, під час роботи програми;
- з файлу чи іншого джерела, під час роботи програми.

Програмна система для інтеграції програмного продукту в режимі реального часу отримуватиме такі вхідні дані:

- адресу Kubernetes кластера;
- опис пайплайну у форматі yaml;
- тип сховища логів контейнерів.

Відповідно до переданого тексту у форматі yaml почнеться виконання пайплайну. Вихідними даними є результати виконання пайплайну.

2.2. Розробка розподіленої архітектури програмного забезпечення

Архітектура програмного забезпечення є основною структурою програмної системи та критерієм створення таких структур та систем. Кожна структура містить програмні елементи, зв'язки між ними та атрибути елементів та відносин.

Основна мета створення якісної архітектури – виявити вимоги, які впливають на структуру програми. Добре закладена архітектура зменшує бізнес-ризики, пов'язані зі створенням технічного рішення та створює міст між бізнесом та технічними вимогами[11].

Узагальнена модель роботи системи представлена на рисунку 2.1.

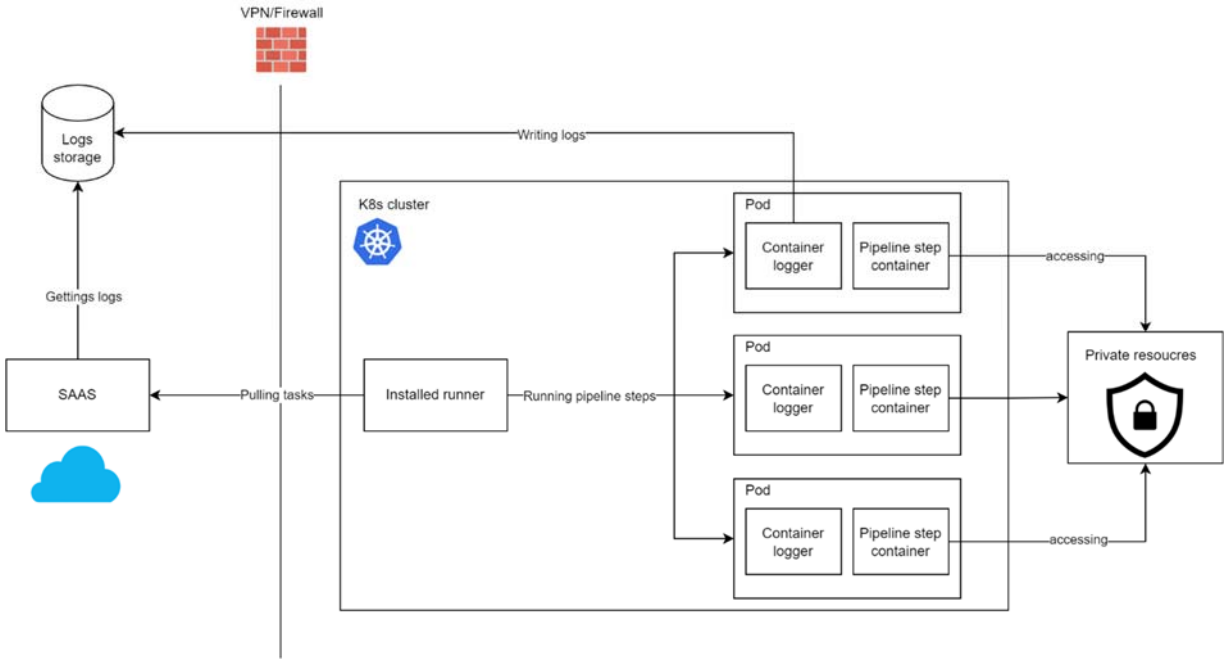


Рисунок 2.1 – Узагальнена модель роботи системи

Для коректної роботи програмної системи необхідно розгорнути SAAS частину у хмарі, базу даних для збереження вихідних даних контейнерів, та встановити компоненту “Runner” у кластері користувача.

Система складається з 3 основних компонент:

- SAAS;
- Runner;
- Container logger (модуль для логування вихідних даних контейнерів).

Під час запуску системи користувач налаштовує пайплайн за допомогою yaml синтаксису. Під час запуску пайплайну в базі даних буде добавлено запис про задачу, яку необхідно виконати. Runner в свою чергу постійно займається опитуванням серверу на предмет нових задач. Як тільки такі з’являються, він починає їх виконання в кластері користувача. Для кожного кроку пайплайну Runner створює Kubernetes Pod з двома контейнерами:

- крок пайплайну створений на основі імеджу вказаного у конфігурації кроку;
- container logger – компонента, яка прослуховує вихідні дані сусіднього контейнера та записує їх до БД, що знаходиться у хмарі.

Варто вказати що також можливе логування до БД що знаходиться у приватній мережі, але в такому випадку потрібно виконати додаткові налаштування для роботи в такий спосіб.

Роль даної компоненти в прослуховуванні Docker демона на вказаній користувачем адресі, фільтрування подій, які з нього надходять на предмет старту контейнера, та додаткове фільтрування відповідно до фільтрів які надав користувач. Якщо стартуючий контейнер успішно проходить фільтрацію, його ідентифікатор передається в Container Logger – компоненту відповідальну за зчитування вихідних даних контейнера. Як тільки зчитування відбулося, вихідні данні передаються в StorageManager – компоненту яка займається записом логів в вибране користувачем сховище.

Потрібно зазначити що зчитування вихідних даних це не одноразовий процес, це буде відбуватися за допомогою потоків даних, тобто відбуватися поступово. Чекаємо надходження порції даних – записуємо.

Через те що пайплайни виконуються в мережі користувача, у них є повний доступ до ресурсів, які знаходяться у цій мережі. Цими ресурсами можуть бути Kubernetes маніфести такі як ConfigMap та Secret, бази даних, захищені API сервіси та все інше, що може бути використано як вхідні дані для кроків.

2.3. Розробка алгоритму взаємодії сервера й Kubernetes кластера

Так як виконання кроків буде здійснено в Kubernetes кластері, буде доречною встановити агент, як частину кластеру. Він буде розгорнутий у вигляді Pod. Поруч з ним буде створено ConfigMap, з налаштуваннями агенту. Такими як інтервал опитування серверу, адреса серверу, максимальна кількість паралельно виконуваних задач.

Оскільки відповідно до розробленої архітектури агент, який відповідає за виконання пайплайнів, знаходиться у приватній мережі користувача, сервер не має доступу до нього. Таким чином потрібно відштовхуватися від

того що зв’язок повинен бути одностороннім. Лише агент має доступ до серверу, але не навпаки. Отже задачею агенту буде постійне опитування сервера на предмет нових задач, які повинні бути виконані у кластері користувача.

Окрім цього агент повинен мати API токен, для виконання авторизованих запитів до серверу. Токен надається від час встановлення агенту у вигляді ConfigMap маніфесту, що знаходиться у кластері клієнта. У випадку коли токен не є валідним, Pod агенту не зможе виконувати опитування серверу. В такому випадку агент не зможе розпочати свою роботу й у його логах буде повідомлення про помилку.

Алгоритм взаємодії агента з сервером (рис. 2.2):

1. Ініціалізація агенту.
2. Отримання конфігурації з ConfigMap, що містить у собі API токен, інтервал опитування серверу, налаштування можливої кількості одночасно виконуваних кроків та пайплайнів.
3. Виконання аутентифікації за допомого API токену.
4. Якщо аутентифікація завершилася помилкою, закінчуємо роботу агенту та повідомляємо про необхідність оновлення токену.
5. У випадку успіху починаємо нескінченний цикл опитування сервера з інтервалом отриманим під час ініціалізації.
6. Виконується запиті на отримання нових задач. Агент вказує їх бажану кількість на основі кількості активних задач та налаштувань одночасного виконання кроків/пайплайнів.
7. Виконання отриманих задач..
8. Витримка інтервалу.
9. Повторення циклу.

Рисунок 2.2 – Алгоритм взаємодії агента з сервером

2.4. Розробка алгоритму виконання пайплайну в середовищі Kubernetes

Пайплайн – це послідовність кроків, які необхідно виконати. CI/CD пайплайни – це практика, спрямована на покращення розгортання програмного забезпечення з використанням підходу DevOps або інженерної надійності сайту (SRE).

Пайплайн CI/CD впроваджує моніторинг та автоматизацію для покращення процесу розробки додатків, особливо на етапах інтеграції та тестування, а також під час доставки та розгортання. Хоча кожен із кроків пайплайну можна виконати вручну, справжня цінність CI/CD пайплайнів реалізується за допомогою автоматизації.

Наступний алгоритм буде використовуватися агентом для виконання задач, отриманих з серверу. Його принцип роботи досить лінійний і оснований на послідовному виконанні кроків пайплайну (рис. 2.3).

Рисунок 2.3 – Алгоритм виконання задач

2.5. Розробка методу зчитування вихідних даних контейнерів

Алгоритм — набір інструкцій, які описують порядок дій виконавця, щоб досягти результату розв'язання задачі за скінченну кількість дій; система правил виконання дискретного процесу, яка досягає поставленої мети за скінченний час. Для візуалізації алгоритмів часто використовують блок-схеми [12].

Ключовим модулем в розробці системи безперервної інтеграції програмного продукту є модуль для логування вихідних даних контейнерів. Розробка полягає в тому що необхідно розробити алгоритми прослуховування (рисунок 2.4), логування (рисунок 2.5), та загальний алгоритм його роботи (рисунок 2.6).

Алгоритм прослуховування демона:

1. Отримання фільтрів.
2. Ітерування масиву фільтрів і їх обробка відповідно до формату даних, який необхідний для правильного функціонування подальшого коду. У випадку виконання пайплайну, фільтром буде унікальний ідентифікатор пайплайну, який також буде прив’язаний до кожного запущеного контейнера у вигляді LABEL.
3. Використовуючи Docker API почати прослуховування подій на демоні. На цьому етапі можливе налаштування фільтрів які підтримуються Docker API. До таких належить тип події, назва контейнера, image який він використовує.
4. Якщо подія є стартом контейнера, продовжити виконання. Інакше закінчити обробку події.
5. Якщо позначки та ім’я контейнера який спричинив подію співпадають з тими, що ввів користувач, почати зчитування логів контейнеру. Інакше закінчити обробку події.

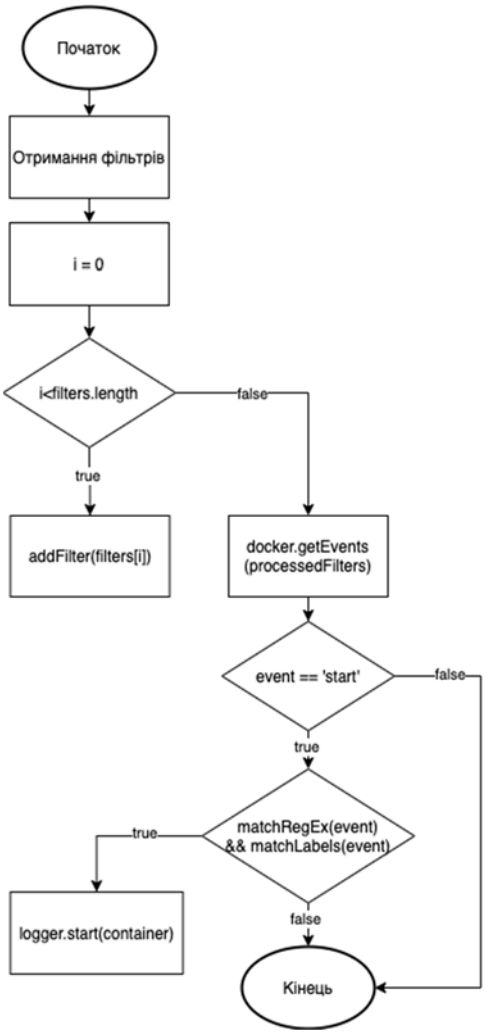


Рисунок 2.4 Блок-схема алгоритму прослуховування Docker демона

Алгоритм логування контейнера:

1. Отримання вибраного сховища та id контейнера.
2. Ініціалізація сховища(наприклад створення з’єднання з базою даних).
3. Створення потоку вихідних даних шляхом під’єднання до контейнера (docker attach).
4. Поєднання стандартних потоків stdout та stderr в один.
5. Передача результуючого потоку в сховище.

Рисунок 2.5 – Блок-схема алгоритму логування вихідних даних контейнера

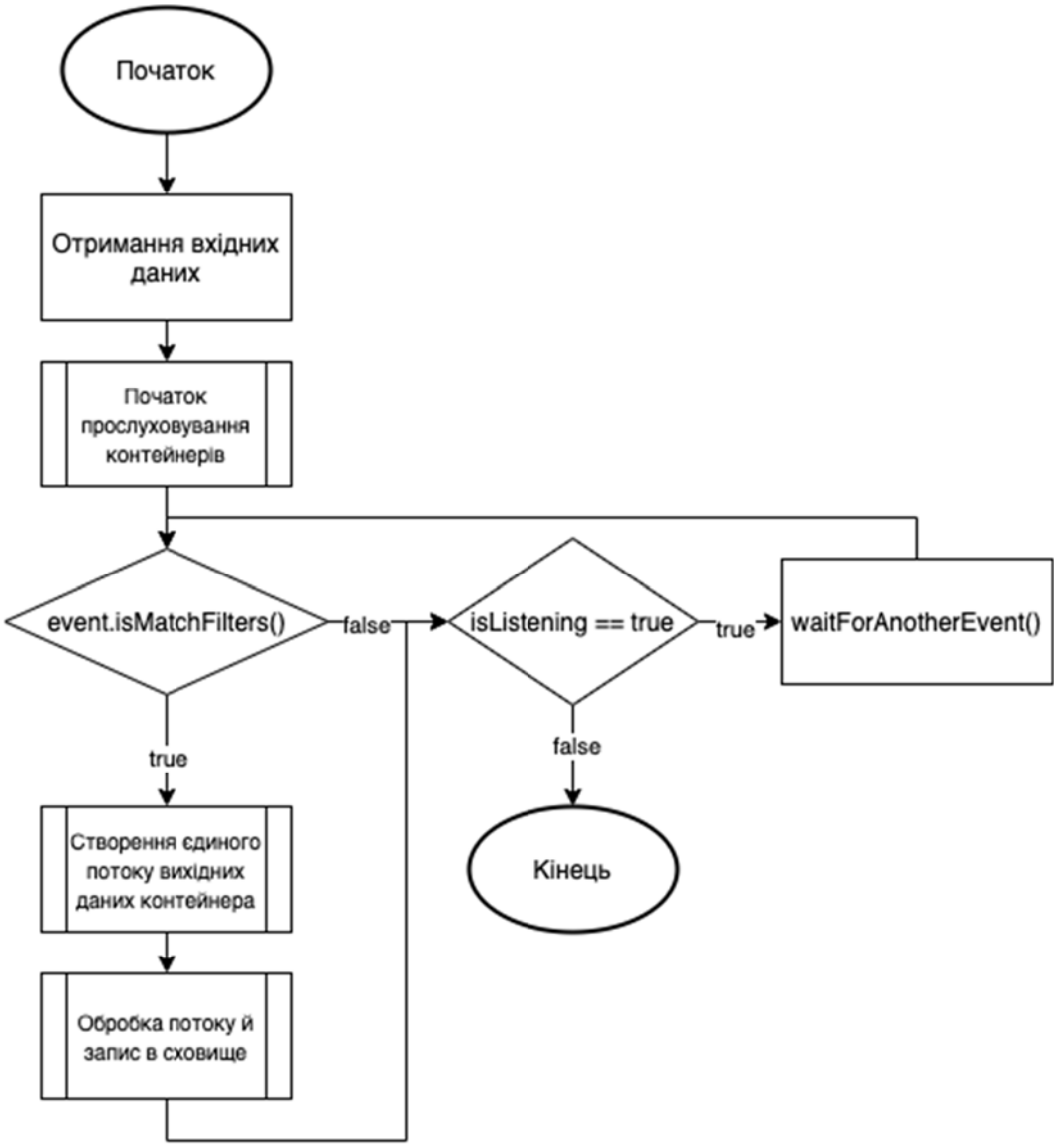


Рисунок 2.6 – Блок-схема загального алгоритму роботи модуля

Для відображення взаємозв’язку між всіма підпроцесами побудовано діаграму компонентів програмного модулю для прослуховування вихідних даних контейнерів (рисунок 2.7).

Діаграма компонент відображає залежності між компонентами програмного забезпечення, включаючи компоненти вихідних кодів, бінарні компоненти, та компоненти, що можуть виконуватись.

Вона дозволяє визначити архітектуру розроблюваної системи, встановив залежності між програмними модулями, компонентами, файлами. Пунктирні лінії, зображені на діаграмі, представляють відношення взаємозалежності.

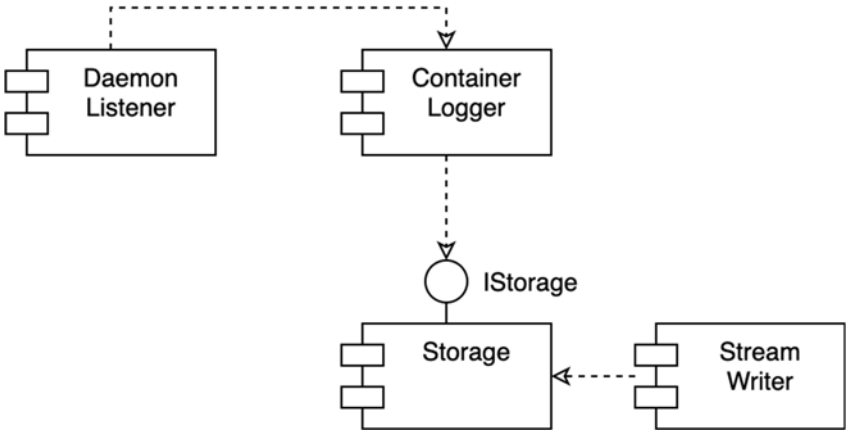


Рисунок 2.7 – Діаграма компонентів програмного модулю

2.6. Розробка структури графічного інтерфейсу

Інтерфейс – це сукупність засобів, методів і правил взаємодії між елементами системи [13].

Розробка візуальної композиції та тимчасової поведінки графічного інтерфейсу є важливою частиною програмування прикладних програм у сфері взаємодії людини та комп'ютера. Його мета — підвищити ефективність і простоту використання для базового логічного дизайну збереженої програми, дисципліни проектування під назвою юзабіліті. Методи дизайну, орієнтованого на користувача, використовуються для того, щоб візуальна мова, введена в дизайн, була добре пристосована до завдань.

Так як графічний інтрефейс є невід'ємною складовою додатків у сучасному світі, його створенню було прикладено багато зусиль. Було вирішено розробити графічний інтерфейс у вигляді веб додатку. Це не змушує користувача виконувати додаткові встановлення на його комп'ютер. Графічний інтерфейс виконаний в мінімалістичному стилі без використання різноманітних бібліотек (рисунок. 2.8). Його задача – показати ефективність програмного модуля логування вихідних даних контейнерів.

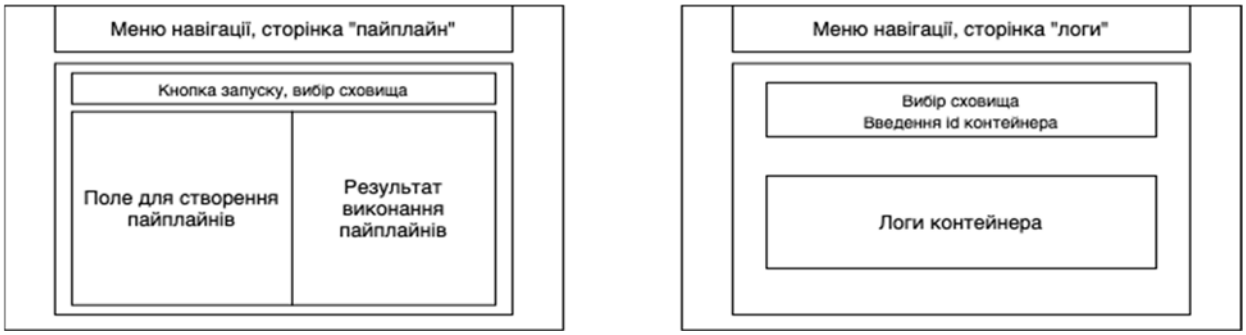


Рисунок 2.8 – Схема графічного інтерфейсу

Висновки до розділу II

Було проаналізовано інформаційне забезпечення необхідне для розробки методу та програмного засобу. Також було розроблено архітектуру програмного додатку. Було розроблено алгоритми взаємодії сервера та Kurbernetes кластера, виконання пайплайнів в середовищі Kubernetes, збереження вихідних даних контейнерів. Для останнього з них було розроблено діаграму компонентів для ілюстрації всіх залежностей між основними компонентами програмного модуля.

Було прийнято рішення про інтерфейс, який буде використовуватися при створенні програмної системи, та розроблено його прототип. Цей інтерфейс буде зручним для користувачів, тому він буде використовуватися в остаточній версії продукту.

РОЗДІЛ ІІІ
РОЗРОБКА ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ДЛЯ ПІДВИЩЕННЯ БЕЗПЕКИ ВИКОНАННЯ CI/CD ПРОЦЕСІВ У СЕРЕДОВИЩІ KUBERNETES

3.1. Варіантний аналіз і обґрунтування вибору засобів реалізації програмного забезпечення

Програмний продукт, який розробляється є не звичайним монолітним додатком, а системою з багатьма компонентами. Тільки працюючи разом вони створюють враження повноцінного продукту. Серед них є клієнтська частина, серверна частина, агент, який повинен бути встановленим у кластер клієнта та логер, який буде використовуватися для зчитування вихідних даних контейнерів. Отже, потрібно обрати мови програмування, яку будуть застосовані до розробки компонентів. Різноманітність задач дає можливість вибору різних мов програмування. Розгортання програмних компонентів теж вимагає дослідження.

Розглянемо технології, за допомогою яких це все було реалізовано.

Найперше що потрібно для функціонування програмного забезпечення – сервер на якому буде виконуватися більша частина бізнес логіки та буде розміщено статичні файли клієнтського застосунку. Одною з основних критерій є підтримка розгортування Docker контейнерів, так як це зробить процес розробки та розгортування значно простішим та гнучкішим, так як майбутня зміна хостингу буде безболісною. В якості такого серверу було обрано сервер, що працює на операційній системі Linux Apline.

Alpine Linux побудований на базі musl libc і busybox. Це робить його меншим і більш ресурсоефективним, ніж традиційні дистрибутиви GNU/Linux. Контейнер вимагає не більше 8 МБ, а мінімальна установка на диск вимагає близько 130 МБ пам’яті. Ви отримуєте не тільки повноцінне середовище Linux, але й великий вибір додаткових утиліт із репозиторію.

Бінарні пакети зменшуються та розбиваються, що дає вам ще більше контролю над тим, що ви встановлюєте, що, у свою чергу, робить ваше середовище якомога меншим та ефективним.

Для того, щоб розгортування Docker контейнерів стало можливим, потрібно також встановити виконавчий файл, за допомогою якого буде запущено Docker daemon та інтерфейс командного рядка, що надасть змогу взаємодіяти з демоном. Це можна зробити використовуючи утиліту apk.

apk - менеджер пакунків Alpine Linux. Ви використовуєте команду apk для видалення, встановлення, оновлення або списку програмного забезпечення на запущеній системі на базі Alpine Linux. Як і більшість сучасних дистрибутивів Linux, усі програмні пакети для Alpine Linux мають цифровий підпис, щоб уникнути проблем із безпекою. Ви можете інсталювати пакунки з локального диска (наприклад, CDROM або USB-накопичувач) або з Інтернет-архіву (сховища), наприклад http://dl-cdn.alpinelinux.org/alpine/v3.5/main. Список сховищ зберігається у файлі конфігурації /etc/apk/repositories.

Вибір мови програмування дуже важливий для розробки будь-якого програмного продукту, оскільки від неї залежить не тільки простота реалізації алгоритму, а також ефективності та швидкості роботи програми. Для реалізації цього програмного модуля потрібно обрати мову програмування, яка вміє чудово працювати з потоками даних. Це означає що вона повинна мати можливість підтримувати й обробляти велику кількість одночасних з’єднань, час життя яких значно більший ніж при звичайних HTTP запитах. На даний момент чудовим рішенням є використання подійно-орієнтованих мов програмування [14].

Ключовим моментом, чому ми обираємо подійно-орієнтовані мови програмування, є уникнення породження обслуговуючих процесів. Окрім того що їх створення є досить ресурсоємкою операцією, самі процеси будуть досить довго використовуватися, так як будуть працювати з потоками. Це зробить програмний модуль повільним на системах з малою кількістю

потоків процесора й навіть на більш потужних, якщо кількість прослуховуваних контейнерів буде великою [15].

Популярні подійно-орієнтовані рішення:

- JavaScript (Node.js);
- Python(Twisted);
- C++(libsigc++).

Node.js – платформа, заснована на движку V8 (здійснює трансляцію JavaScript в машинний код), що перетворює JavaScript з вузькоспеціалізованого мови в мову загального призначення. Node.js додає можливість JavaScript взаємодіяти з пристроями введення-виведення через свій API (написаний на C++), підключати інші зовнішні бібліотеки, написані на різних мовах, забезпечуючи виклики до них з JavaScript-коду. Node.js застосовується переважно на сервері, виконуючи роль веб-сервера, але є можливість розробляти на Node.js і десктопні віконні додатки (за допомогою NW.js, AppJS або Electron для Linux, Windows і macOS) і навіть програмувати мікроконтролери (наприклад, tessel і espruino). В основі Node.js лежить подійно-орієнтоване і асинхронне (або реактивне) програмування з неблокуючим вводом/виводом [16].

Node.js в основному використовується для створення мережевих програм, таких як веб-сервери. Найбільш істотна відмінність між Node.js і PHP полягає в тому, що більшість функцій у PHP блокують до завершення (команди виконуються лише після завершення попередніх команд), тоді як функції Node.js не блокують (команди виконуються одночасно або навіть паралельно, і використовувати зворотні виклики, щоб сигналізувати про завершення або невдачу).

libsigc++ - це бібліотека C ++ для зворотних викликів. Вона реалізує систему зворотного виклику для використання в абстрактних інтерфейсах та загальному програмуванні. libsigc++ - одна з найбільш ранніх реалізацій концепції сигналів і слотів, реалізованої за допомогою метапрограмування

шаблонів C++. Він був створений як альтернатива використанню мета-компілятора, такого як в реалізації сигналів і слотів у Qt [17].

Таблиця 3.1

Порівняння мов програмування

Мова програмування	JS(Node.js)	Python(Twisted)	C++(libsigc++)
Динамічна типізація	+	+	-
Event loop «з коробки»	+	-	-
Stream API «з коробки»	+	+	+
Швидкі операції з пам'яттю	+	-	+
Загалом	4	2	2

Згідно з результатами порівняння мова JavaScript має перевагу над іншими мовами, тому для реалізації серверної частини було обрано саме її.

Для розробки агента встановленого в кластер користувача було обрано мову Golang.

Go (a.k.a. Golang) — мова програмування, вперше розроблена в Google [18]. Це статично типізована мова з синтаксисом, похідним від C, але з додатковими функціями, такими як збір сміття, безпека типів, деякі можливості динамічного введення, додаткові вбудовані типи (наприклад, масиви змінної довжини та карти ключ-значення) , і велика стандартна бібліотека.

Go виразний, лаконічний, чистий та ефективний. Його механізми паралельності дозволяють легко писати програми, які отримують

максимальну віддачу від багатоядерних і мережевих машин, в той час як його нова система типів забезпечує гнучку і модульну конструювання програм. Go швидко компілюється в машинний код, але має зручність збирання сміття та можливість відображення під час виконання. Це швидка, статично типізована, скомпільована мова, яка схожа на динамічно типізовану, інтерпретовану мову.

Вибір впав саме на цю мову через її адаптованість під хмарну розробку. Оскільки Kubernetes було створено за допомогою цієї мови програмування, вона стала стандартом де-факто у світі хмарних технологій. Вона активно просувається проектом CNCF й є основною в більшості його проектів.

The Cloud Native Computing Foundation (CNCF) — це проект Linux Foundation, який був заснований у 2015 році, щоб допомогти просунути технологію контейнерів та налаштувати технологічну галузь навколо її еволюції.

Golang має величезну екосистему бібліотек створену навколо Kubernetes. За допомогою цієї мови створюються розширення таких компонентів Kubernetes:

- controllers;
- scheduler;
- kubectl;
- network plugins;
- storage plugins.

На сьогоднішній день найбільш зручними та сучасними середовищами для розробки веб додатків за допомогою мови програмування JavaScript є Visual Studio Code та WebStorm.

Visual Studio Code - це легкий, але потужний редактор вихідного коду, який доступний для Windows, MacOS та Linux. Він оснащений вбудованою підтримкою JavaScript, TypeScript та Node.js і має багату екосистему

розширень для інших мов (таких як C ++, C #, Java, Python, PHP та Go) та режимів виконання (таких як .NET та Unity).

Основною його перевагою є можливість розширення функціоналу середовища розробки за допомогою розширень. У багатьох аналогів VSCode вони теж присутні, але саме у цьому редакторі їх буквально «безліч», що дозволяє його кастомізацію та використання для майже будь-яких цілей: програмування на будь-якій високорівневій мові, використання інструментів розробки.

VS Code постачається під стандартну ліцензію на продукт Microsoft, оскільки він має невеликий відсоток налаштувань, характерних для Microsoft. Це безкоштовно, незважаючи на комерційну ліцензію.

Інтерфейс редактору Visual Studio Code зображено на рисунку 3.1.

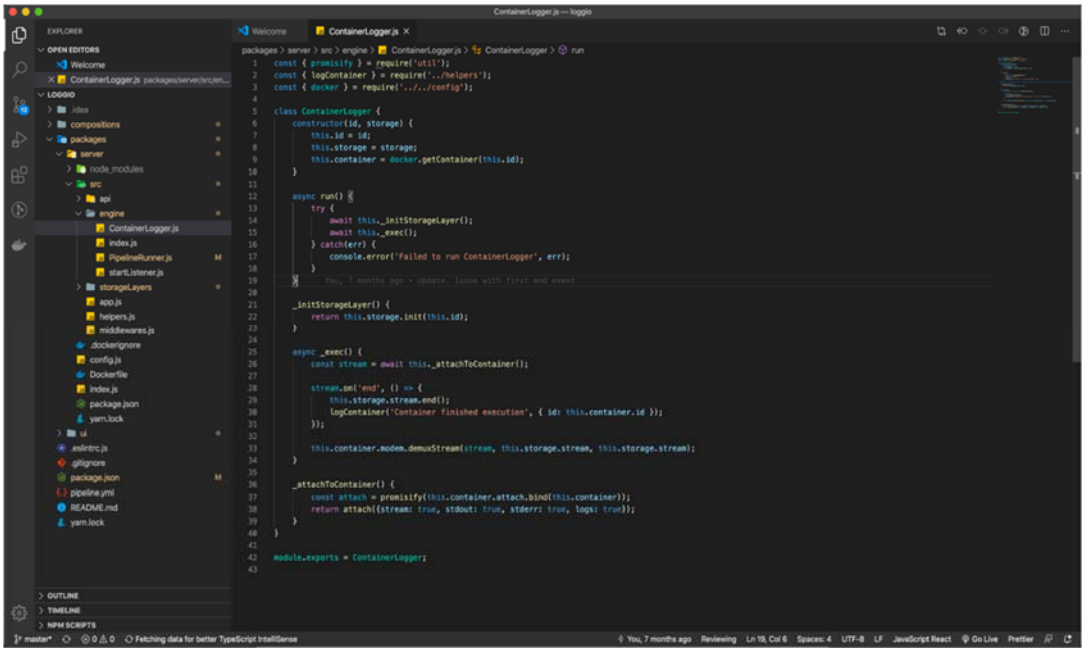


Рисунок 3.1 – Інтерфейс редактору Visual Studio Code

WebStorm - це інтегроване середовище розробки (IDE) від JetBrains для мови програмування JavaScript. Крім того, WebStorm підтримує HTML5, Node.js, Bootstrap, Angular / AngularJS, TypeScript, PhoneGap / Cordova, Dart та багато інших технологій, тому в основному використовується для

розробки веб-мобільних додатків. WebStorm заснований на IntelliJ IDEA від JetBrains, але це версія, спеціалізована на JavaScript.

Діапазон функцій можна розширити за допомогою плагінів, які частково розроблені JetBrains, а частково громадою.

WebStorm містить інструменти для рефакторингу, контролю версій, виділення коду та синтаксису, тестування та для часткової генерації коду. Він пропонує вам інтелектуальну допомогу в програмуванні на JavaScript та мовах які компілюються в нього, Node.js, HTML та CSS. Надає потужні рішення для авто-завершенням коду, навігації між файлами та проектами, виявленням помилок на ходу та рефакторинга для всіх перелічених мов програмування. IDE аналізує проект, щоб забезпечити найкращі результати заповнення коду для всіх підтримуваних мов. Сотні вбудованих інспекцій повідомляють про всі можливі проблеми під час написання коду та пропонують їх рішення.

Окрім цього дана IDE має виключну підтримку тестування, оскільки WebStorm інтегрується з такими рушіями тестів як Karma, Mocha, Jest і Protractor. Це дозволяє запускати й відлагоджувати тести прямо в IDE, переглядати рчезультати їх виконання у хорошому візуальному форматі з можливістю перейти до будь-якого виконаного тесту.

Інтерфейс інтегрованого середовища Webstorm зображено на рисунку 3.2.

База даних — це організована сукупність даних, які зберігаються та доступні в електронному вигляді з комп’ютерної системи. Там, де бази даних є складнішими, вони часто розробляються з використанням формальних методів проектування та моделювання.

При виборі сучасної бази даних одним із найважливіших рішень є вибір реляційної (SQL) або нереляційної (NoSQL) структури даних. Хоча обидва варіанти є життєздатними, між ними є ключові відмінності, які користувачі повинні мати на увазі, приймаючи рішення.

На самому базовому рівні найбільша відмінність між цими двома технологіями полягає в тому, що бази даних SQL є реляційними, а бази даних NoSQL нереляційними [19].

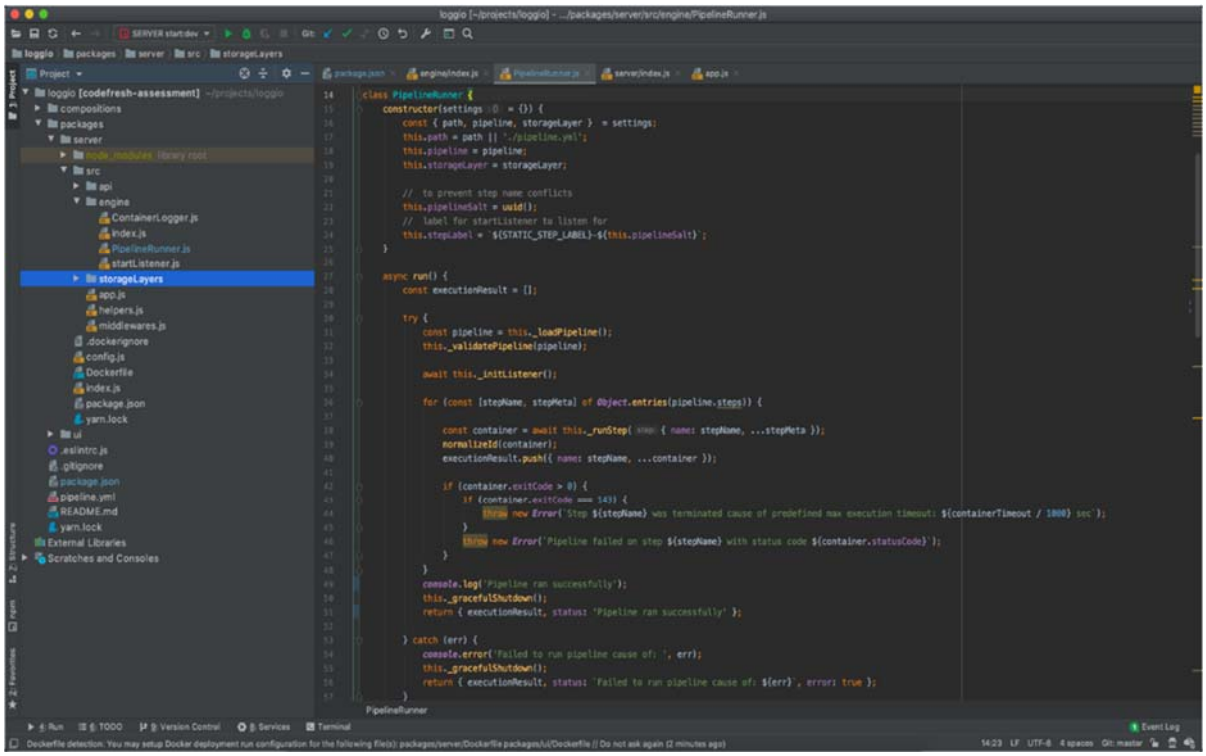


Рисунок 3.2 – Інтерфейс IDE Webstorm

Бази даних SQL використовують структуровану мову запитів і мають попередньо визначену схему для визначення та маніпулювання даними. SQL є однією з найбільш універсальних і широко використовуваних мов запитів, що робить його безпечним вибором для багатьох випадків використання. Він ідеально підходить для складних запитів. Однак SQL може бути занадто обмежуючим. Перш ніж працювати з нею, потрібно використовувати попередньо визначені схеми, щоб визначити структуру даних. Усі ваші дані повинні мати однакову структуру. Цей процес вимагає серйозної попередньої підготовки. Якби ви коли-небудь захотіли змінити структуру даних, це було б складно і зашкодило б всій вашій системі.

Бази даних NoSQL мають динамічні схеми для неструктурованих даних, і дані зберігаються багатьма способами. Ви можете використовувати

для своїх даних сховище, стовпчико-орієнтоване, документо-орієнтоване, графіко орієнтоване або KeyValue. Ця гнучкість означає, що:

- ви можете створювати документи без попереднього визначення їх структури;
- кожен документ може мати свою унікальну структуру;
- синтаксис може відрізнятися від бази даних до бази даних;
- ви можете додавати поля по ходу.

Бази даних SQL у більшості ситуацій вертикально масштабовані. Ви можете збільшити навантаження на один сервер, додавши більше ємності CPU, RAM або SSD. Бази даних NoSQL мають горизонтальне масштабування. Ви можете обробляти більший трафік за допомогою сегментування, що додає більше серверів до вашої бази даних NoSQL. Горизонтальне масштабування має більшу загальну ємність, ніж вертикальне масштабування, що робить бази даних NoSQL кращим вибором для великих і часто змінних наборів даних.

Підсумовуючи, п’ять основних відмінностей між SQL і NoSQL:

- бази даних SQL є реляційними, бази даних NoSQL нереляційними.
- бази даних SQL використовують структуровану мову запитів і мають попередньо визначену схему.
- бази даних NoSQL мають динамічні схеми для неструктурованих даних.
- бази даних SQL мають вертикальне масштабування, а бази даних NoSQL горизонтально.
- бази даних SQL засновані на таблицях, тоді як бази даних NoSQL є сховищами документів, ключів і значень, графіків або широких стовпців.
- бази даних SQL краще для багаторядкових транзакцій, тоді як NoSQL краще для неструктурованих даних, таких як документи або JSON.

Оцінивши всі переваги та недоліки вказані вище було прийнято рішення використовувавти NoSQL базу даних MongoDB. Це БД з відкритим вихідним кодом, що дає змогу безкоштовно її використовувати та розгортувати на власній інфраструктурі.

Зворотний проксі-сервер — це тип проксі-сервера, який розгортається між клієнтами та внутрішніми/початковими серверами, наприклад, HTTP-сервером, таким як NGINX, Apache тощо, або серверами програм, написаними на Nodejs, Python, Java, Ruby , PHP та багато інших мов програмування.

Це шлюз або проміжний сервер, який приймає запит клієнта, передає його одному або кільком внутрішнім серверам і згодом отримує відповідь від сервера і доставляє її назад клієнту, таким чином створюючи враження, ніби вміст походить із самого зворотного проксі-сервера.

Як правило, зворотний проксі-сервер — це внутрішній проксі-сервер, який використовується як «фронт-енд» для контролю та захисту доступу до внутрішніх серверів у приватній мережі: зазвичай він розгортається за мережевим брандмауером.

Це допомагає внутрішнім серверам досягти анонімності для підвищення їх безпеки. В IT-інфраструктурі зворотний проксі-сервер також може функціонувати як брандмауер програми, балансувальник навантаження, термінатор TLS, веб-прискорювач (за допомогою кешування статичного та динамічного вмісту) та багато іншого.

В якості зворотнього проксі серверу було обрано NGINX.

Це безкоштовний, високопродуктивний і дуже популярний HTTP-сервер і зворотний проксі-сервер з відкритим вихідним кодом. NGINX добре відомий своєю високою продуктивністю, стабільністю, багатим набором функцій, простою та гнучкою конфігурацією та низьким споживанням ресурсів (особливо малим обсягом пам’яті).

NGINX підтримує прискорене зворотне проксі-сервер із кешуванням за допомогою модуля ngx_http_proxy_module, який дозволяє передавати запити

на інший сервер через протоколи, відмінні від HTTP, наприклад FastCGI, uwsgi, SCGI та memcached.

Важливо, що він підтримує балансування навантаження та відмовостійкість, які є життєво важливими аспектами великомасштабних розподілених обчислювальних систем. Модуль ngx_http_upstream_module дозволяє визначати групи серверних серверів для розподілу запитів, що надходять від клієнтів. Це робить ваші програми більш надійними, доступними та надійними, високомасштабованими, з часом відгуку та пропускною здатністю. Крім того, що стосується безпеки, він підтримує припинення SSL/TLS та багато інших функцій безпеки.

3.2. Розробка графічного інтерфейсу користувача

Для розробки клієнтського інтерфейсу було вирішено використовувати технологію Single Page Application.

Односторінкові сайти допомагають підтримувати користувача в єдиному, зручному веб-просторі, де вміст представляється користувачеві просто, легко та зручно.

Плюси використання SPA:

- Легке відлагодження в Chrome, оскільки є можливість контролювати мережеві операції, досліджувати елементи сторінки та пов’язані з нею дані.
- Швидкість. Оскільки більшість ресурсів (HTML + CSS + скрипти) завантажуються лише один раз протягом усього життя програми. Повторно можуть бути завантажені лише дані.
- SPA дозволяє ефективно кешувати дані. Додаток надсилає лише один запит й зберігає всі дані, тоді він може використовувати ці дані та працювати навіть офлайн.

Однією із найзручніших технологій для реалізації SPA є бібліотека для JavaScript – React.js.

React.js - це бібліотека JavaScript, створена для створення швидких та інтерактивних інтерфейсів користувачів для веб- та мобільних додатків. Це бібліотека, що працює на компонентах, з відкритим кодом, на основі компонентів, що відповідає лише за шар перегляду програми. У архітектурі Model View Controller (MVC) шар перегляду відповідає за те, як виглядає і почувається додаток:

- React використовує підхід, заснований на компонентах. У ньому кожен елемент DOM може бути компонентом. Це забезпечує найкращі умови створення ізольованих фрагментів коду та логіки, які простіше відлагоджувати, підтримувати та використовувати повторно.
- здатність боротися із поширеною помилкою пошукової системи для читання важких JavaScript додатків. Як рішення, React може працювати на сервері, надаючи та повертаючи віртуальний DOM у браузер як звичайну веб-сторінку.

UML-діаграми активності алгоритму запуску пайплайну та отримання результатів виконання відображено на рисунках 3.3 та 4.4 відповідно.

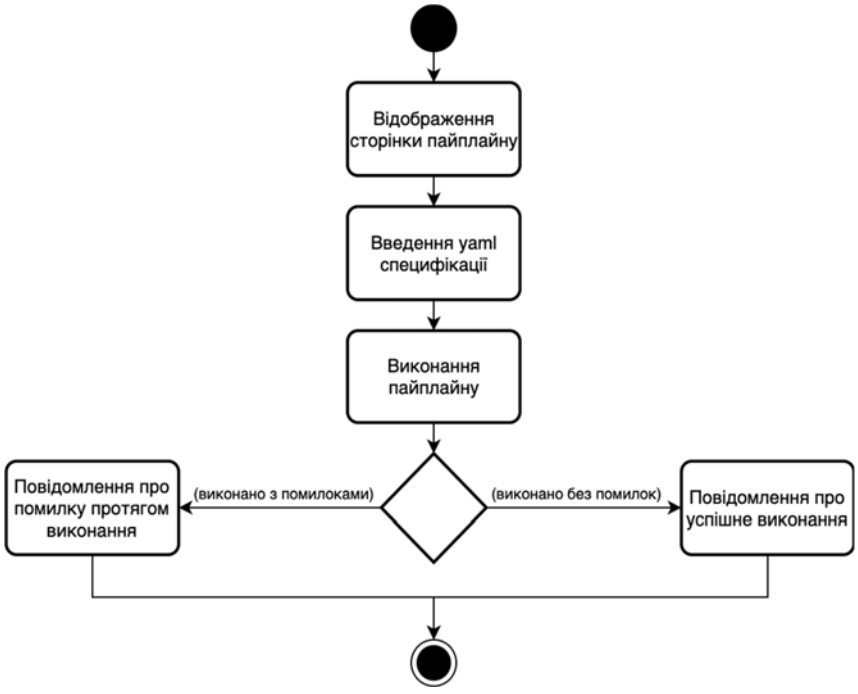


Рисунок 3.3 – Діаграма активності алгоритму запуску пайплайну

Рисунок 3.4 – Діаграма активності алгоритму отримання логів

3.3. Розробка модуля прослуховування та логування вихідних даних кроків

Ключовими програмними компонентами є модулі прослуховування Docker демона та логування вихідних даних контейнера.

Docker надає API для взаємодії з демоном Docker (так званий Docker Engine API), а також пакети SDK для Go і Python. Пакети SDK дозволяють створювати та масштабувати програми та рішення Docker швидко й легко. Якщо Go або Python не працюють для вас, ви можете використовувати Docker Engine API безпосередньо. API Docker Engine — це RESTful API, до якого звертаються HTTP клієнти, наприклад wget або curl, або бібліотека HTTP, яка є частиною більшості сучасних мов програмування.

Більшість CLI команд безпосередньо відповідають кінцевим точкам API (наприклад, docker ps GET /containers/json). Важливим винятком є запуск контейнерів, який складається з кількох викликів API.

Обидва модулі реалізовано за допомогою Docker API, а саме таких HTTP запитів:

- GET /events;
- POST /containers/(id or name)/attach.

GET /events – запит на отримання потоку подій які відбуваються на Docker демоні. Під час його надсилання є можливість вказати тип подій, які ми хочемо отримувати в потоку подій.

POST /containers/(id or name)/attach – запит на отримання потоку вихідних даних контейнера. Він надсилається після того як подія про старт контейнера, яка була отримана з потоку подій, пройшла валідацію відповідно до фільтрів та id контейнера було передано в ContainerLogger модуль.

Код алгоритму прослуховування наведено на рисунку 3.5:

```
const _ = require('lodash');
const { docker } = require('../../config');

class StartListener {
    start(selector, containerHandler) {
        return new Promise((resolve, reject) => {
            const filters = this._getApiFilter(selector);
            docker.getEvents({ filters }, (err, events) => {
                if (err) {
                    reject(err);
                } else {
                    console.log('Listening for starting containers...');
                    this.events = events;
                    events.on('data', (chunk) => {
                        const containerData = JSON.parse(chunk.toString());
                        if (this._matchLocalFilters(containerData, selector)) {
                            containerHandler(containerData)
                        }
                    });
                    resolve();
                }
            });
        });
    }
```

Рисунок 3.5 – Код алгоритму прослуховування

Код алгоритму логування наведено на рисунку 3.6:

```
const { promisify } = require('util');
const { logContainer } = require('../helpers');
const { docker } = require('../../config');

class ContainerLogger {
    constructor(id, storage) {
        this.id = id;
        this.storage = storage;
        this.container = docker.getContainer(this.id);
    }

    async run() {
        try {
            await this._initStorageLayer();
            await this._exec();
        } catch(err) {
            console.error('Failed to run ContainerLogger', err);
        }
    }

    _initStorageLayer() {
        return this.storage.init(this.id);
    }

    async _exec() {
        const stream = await this._attachToContainer();

        stream.on('end', () => {
            this.storage.stream.end();
            logContainer('Container finished execution', { id: this.container.id });
        });

        this.container.modem.demuxStream(stream, this.storage.stream, this.storage.stream);
    }

    _attachToContainer() {
        const attach = promisify(this.container.attach.bind(this.container));
        return attach({stream: true, stdout: true, stderr: true, logs: true});
    }
}

module.exports = ContainerLogger;
```

Рисунок 3.6 – Код алгоритму логування

В початковій версії програмної системи передбачино 2 способи збереження вихідних даних контейнерів:

- Mongo DB;
- файлова система.

MongoDB — документо-орієнтована система керування базами даних (СКБД) з відкритим вихідним кодом, яка не потребує опису схеми таблиць. MongoDB займає нішу між швидкими і масштабованими системами, що оперують даними у форматі ключ/значення, і реляційними СКБД, функціональними і зручними у формуванні запитів. При запуску програмного модуля користувач може вказати адресу запущеного MongoDB

демона через змінну оточення MONGO_URI. По замовчуванню вона дорінює «mongodb://mongo/docker-logger».

При збереженні в файлову систему для кожного контейнера будуть створені файли, де ім’я файла – id контейнера. По замовчуванню вони буду створені за шляхом «/<os_home_dir>/<current_user>/fs-logs-storage».

На рисунку 3.7 наведений код базових абстрактних класів модуля збереження даних.

```
const WriterStream = require('./WriterStream');

class StorageLayer {
    constructor(container) {
        this.container = container
    }

    /**
     * Will be called before writing to stream
     */
    async init() {}
}

class StorageLayerReader extends StorageLayer {
    async read() {}
}

class StorageLayerWriter extends StorageLayer {
    constructor(container) {
        super(container);
        this.stream = new WriterStream({
            onData: this.write.bind(this),
            onFinish: this.onFinish.bind(this),
        });
    }

    /**
     * Will be called every time stream receive data
     */
    write(log) {
        console.log(log);
    }

    /**
     * Will be called on stream 'finish' event
     */
    onFinish() {}
}

module.exports = {
    Writer: StorageLayerWriter,
    Reader: StorageLayerReader
};
```

Рисунок 3.7 – Код модуля збереження даних

На рисунку 3.8 та 3.9 наведений код реалізації класів модуля збереження даних для MongoDB.

```
const mongoose = require('mongoose');

const BaseStorageLayer = require('../Base');
const ContainerLog = require('./Log');
const { mongo } = require('../../../config');
const { notFoundMessage } = require('../constants');

class MongoStorageRead extends BaseStorageLayer.Reader {
    async init() {
        if (mongoose.connection.readyState === 0) {
            return this._initMongo();
        }
    }

    async read() {
        const { id } = this.container;
        const res = await ContainerLog.findById(id);
        if (res)  {
            return res.data;
        } else {
            throw new Error(notFoundMessage);
        }
    }

    _initMongo() {
        return mongoose.connect(mongo.uri, { useNewUrlParser: true, useUnifiedTopology: true });
    }
}

module.exports = MongoStorageRead;
```

Рисунок 3.8 – Код зчитування даних з MongoDB

```
const mongoose = require('mongoose');

const BaseStorageLayer = require('../Base');
const ContainerLog = require('./Log');
const { mongo } = require('../../../config');
const { getImageAndName } = require('../../helpers');

class MongoStorageWrite extends BaseStorageLayer.Writer {
    onFinish() {
        this.containerLog.save().then(() => console.log('Saved successfully!')).catch(console.error);
    }

    async init() {
        this.containerLog = new ContainerLog({
            _id: this.container.id,
            data: '',
            ...getImageAndName(this.container)
        });

        if (mongoose.connection.readyState === 0) {
            return this._initMongo();
        }
    }

    write(log) {
        this.containerLog.data += log;
    }

    _initMongo() {
        return mongoose.connect(mongo.uri, { useNewUrlParser: true, useUnifiedTopology: true });
    }
}

module.exports = MongoStorageWrite;
```

Рисунок 3.9 – Код збереження даних у MongoDB

Так як у якості мови програмування для модуля виконання пайплайну було обрано Golang, розпочнемо з ініціалізації проекта. Створимо main.go файл та імпліментуємо функцію main (рис. 3.10).

```go
package main

import (
	"github.com/zubroide/go-api-boilerplate/command"
	"github.com/zubroide/go-api-boilerplate/dic"
	"fmt"
	"github.com/getsentry/raven-go"
	"github.com/spf13/viper"
	"os"
)

func readConfig() {
	var err error

	viper.SetConfigFile("base.env")
	viper.SetConfigType("props")
	err = viper.ReadInConfig()
	if err != nil {
		fmt.Println(err)
		return
	}

	if _, err := os.Stat(".env"); os.IsNotExist(err) {
		fmt.Println("WARNING: file .env not found")
	} else {
		viper.SetConfigFile(".env")
		viper.SetConfigType("props")
		err = viper.MergeInConfig()
		if err != nil {
			fmt.Println(err)
			return
		}
	}

	// Override config parameters from environment variables if specified
	for _, key := range viper.AllKeys() {
		viper.BindEnv(key)
	}
}

func main() {
	readConfig()

	dic.InitContainer()

	client := dic.Container.Get(dic.RavenClient).(*raven.Client)
	if client != nil {
		client.CapturePanicAndWait(func() {
			command.Execute()
		}, nil)
	} else {
		command.Execute()
	}
}
```

Рисунок 3.10 – main.go

Одною з основних задач агента є опитування сервера на предмет нових задач. Але це не єдина задача агенту, так що виконуватися опитувач повинен

в окремому потоці. Для цього необхідно використати go рутину. Код ініціалізації агенту зображено на рисунку 3.11.

```go
// New creates a new Agent instance
func New(opt *Options) (*Agent, error) {
    if err := checkOptions(opt); err != nil {
        return nil, err
    }
    id := opt.ID
    cf := opt.Codefresh
    runtimes := opt.Runtimes
    log := opt.Logger
    taskPullingInterval := defaultTaskPullingInterval
    if opt.TaskPullingSecondsInterval != time.Duration(0) {
        taskPullingInterval = opt.TaskPullingSecondsInterval
    }
    statusReportingInterval := defaultStatusReportingInterval
    if opt.StatusReportingSecondsInterval != time.Duration(0) {
        statusReportingInterval = opt.StatusReportingSecondsInterval
    }
    taskPullerTicker := time.NewTicker(taskPullingInterval)
    reportStatusTicker := time.NewTicker(statusReportingInterval)
    wg := &sync.WaitGroup{}

    if opt.Monitor == nil {
        opt.Monitor = monitoring.NewEmpty()
    }
    httpClient.HTTPClient.Transport = opt.Monitor.NewRoundTripper(httpClient.HTTPClient.Transport)

    return &Agent{
        id,
        cf,
        runtimes,
        log,
        taskPullerTicker,
        reportStatusTicker,
        false,
        Status{},
        wg,
        opt.Monitor,
    }, nil
}

// Start starting the agent process
func (a *Agent) Start(ctx context.Context) error {
    if a.running {
        return errAlreadyRunning
    }
    a.running = true
    a.log.Info("Starting agent")

    go a.startTaskPullerRoutine(ctx)
    go a.startStatusReporterRoutine(ctx)

    reportStatus(ctx, a.cf, codefresh.AgentStatus{
        Message: "All good",
    }, a.log)

    return nil
}
```

Рисунок 3.11 –agent.go – ініціалізація агенту

Код циклу опитування зображено на рисунку 3.12.

```go
func (a *Agent) startTaskPullerRoutine(ctx context.Context) {
    for {
        select {
        case <-ctx.Done():
            return
        case <-a.taskPullerTicker.C:
            a.wg.Add(1)
            go func(runtimes map[string]runtime.Runtime, wg *sync.WaitGroup, logger logger.Logger, monitor monitoring.Monitor) {
                tasks := pullTasks(ctx, client, logger)
                startTasks(ctx, tasks, runtimes, logger, monitor)
                time.Sleep(time.Second * 10)
                wg.Done()
            }(a.cf, a.runtimes, a.wg, a.log, a.monitor)
        }
    }
}
```

Рисунок 3.12 –agent.go – виконання опитування серверу

Результатом успішного опитування буде список задач, які необхідно виконати. Вони передаються до функції startTasks, імпліментація якої зображена на рисунку 3.13.

```go
func startTasks(ctx context.Context, tasks []task.Task, runtimes map[string]runtime.Runtime, logger logger.Logger, monitor monitoring.Monitor) {
    creationTasks := []task.Task{}
    deletionTasks := []task.Task{}
    agentTasks := []task.Task{}

    // divide tasks by types
    for _, t := range tasks {
        logger.Debug("Received task", "type", t.Type, "tid", t.Metadata.Workflow, "runtime", t.Metadata.ReName)
        switch t.Type {
        case task.TypeCreatePod, task.TypeCreatePVC:
            creationTasks = append(creationTasks, t)
        case task.TypeDeletePod, task.TypeDeletePVC:
            deletionTasks = append(deletionTasks, t)
        case task.TypeAgentTask:
            agentTasks = append(agentTasks, t)
        default:
            logger.Error("unrecognized task type", "type", t.Type, "tid", t.Metadata.Workflow, "runtime", t.Metadata.ReName)
        }
    }

    // process agent tasks
    for i := range agentTasks {
        t := agentTasks[i]
        logger.Info("executing agent task", "tid", t.Metadata.Workflow)
        txn := newTransaction(monitor, t.Type, t.Metadata.Workflow, t.Metadata.ReName)
        go func(tid string) {
            if err := executeAgentTask(&t, logger); err != nil {
                logger.Error(err.Error())
                txn.NoticeError(err)
            }
            txn.End()
            logger.Info("finished agent task", "tid", t.Metadata.Workflow)
        }(t.Metadata.Workflow)
    }

    // process creation tasks
    for _, tasks := range groupTasks(creationTasks) {
        reName := tasks[0].Metadata.ReName
        runtime, ok := runtimes[reName]
        txn := newTransaction(monitor, "start-workflow", tasks[0].Metadata.Workflow, reName)

        if !ok {
            logger.Error("Runtime not found", "workflow", tasks[0].Metadata.Workflow, "runtime", reName)
            txn.NoticeError(errRuntimeNotFound)
            txn.End()
            continue
        }
        logger.Info("Starting workflow", "workflow", tasks[0].Metadata.Workflow, "runtime", reName)
        if err := runtime.StartWorkflow(ctx, tasks); err != nil {
            logger.Error(err.Error())
            txn.NoticeError(err)
        }
        txn.End()
    }

    // process deletion tasks
    for _, tasks := range groupTasks(deletionTasks) {
        reName := tasks[0].Metadata.ReName
        runtime, ok := runtimes[reName]
        txn := newTransaction(monitor, "terminate-workflow", tasks[0].Metadata.Workflow, reName)

        if !ok {
            logger.Error("Runtime not found", "workflow", tasks[0].Metadata.Workflow, "runtime", reName)
```

Рисунок 3.13 –agent.go – виконання задач

Для спрощення створення та видалення Kubernetes ресурсів було створено невеликий набір утилітарних функцій. Вони були реалізовані як частина пакету Kubernetes у файлі kubernetes.go код якого зображено на рисунках 3.14 та 3.15.

```
84  func (k kube) CreateResource(ctx context.Context, spec interface{}) error {
85
86      bytes, err := json.Marshal(spec)
87      if err != nil {
88          return err
89      }
90
91      obj, _, err := kubeDecode(bytes, nil, nil)
92      if err != nil {
93          return err
94      }
95
96      var namespace string
97      switch obj := obj.(type) {
98      case *v1.PersistentVolumeClaim:
99          namespace = obj.ObjectMeta.Namespace
100         _, err = k.client.CoreV1().PersistentVolumeClaims(namespace).Create(ctx, obj, metav1.CreateOptions{})
101         if err != nil {
102             return err
103         }
104         k.logger.Info("PersistentVolumeClaim has been created")
105
106     case *v1.Pod:
107         namespace = obj.ObjectMeta.Namespace
108         _, err = k.client.CoreV1().Pods(namespace).Create(ctx, obj, metav1.CreateOptions{})
109         if err != nil {
110             return err
111         }
112         k.logger.Info("Pod has been created")
113
114     }
115     return err
116 }
```

Рисунок 3.14 –kubernetes.go – створення ресурсу

```
118 func (k kube) DeleteResource(ctx context.Context, opt DeleteOptions) error {
119     switch opt.Kind {
120     case task.TypeDeletePVC:
121         err := k.client.CoreV1().PersistentVolumeClaims(opt.Namespace).Delete(ctx, opt.Name, metav1.DeleteOptions{})
122         if err != nil {
123             return err
124         }
125         k.logger.Info("PersistentVolumeClaim has been deleted")
126
127     case task.TypeDeletePod:
128         err := k.client.CoreV1().Pods(opt.Namespace).Delete(ctx, opt.Name, metav1.DeleteOptions{})
129         if err != nil {
130             return err
131         }
132         k.logger.Info("Pod has been deleted")
133
134     }
135
136     return nil
137 }
138
```

Рисунок 3.15 –kubernetes.go – видалення ресурсу

У процесі виконання створюється Kubernetes ресурс типу Workflow. Цей ресурс не підтримується з коробки, тому його підтримку потрібно оголосити під час встановлення агенту. Ресурси які не постачаються з коробки прийнято називати CRD – Custom Resource Definition. CRD – це розширення API Kubernetes.

Ресурс — це endpoint в Kubernetes API, який зберігає набір об’єктів API певного типу; наприклад, вбудований ресурс pods містить колекцію об'єктів Pod.

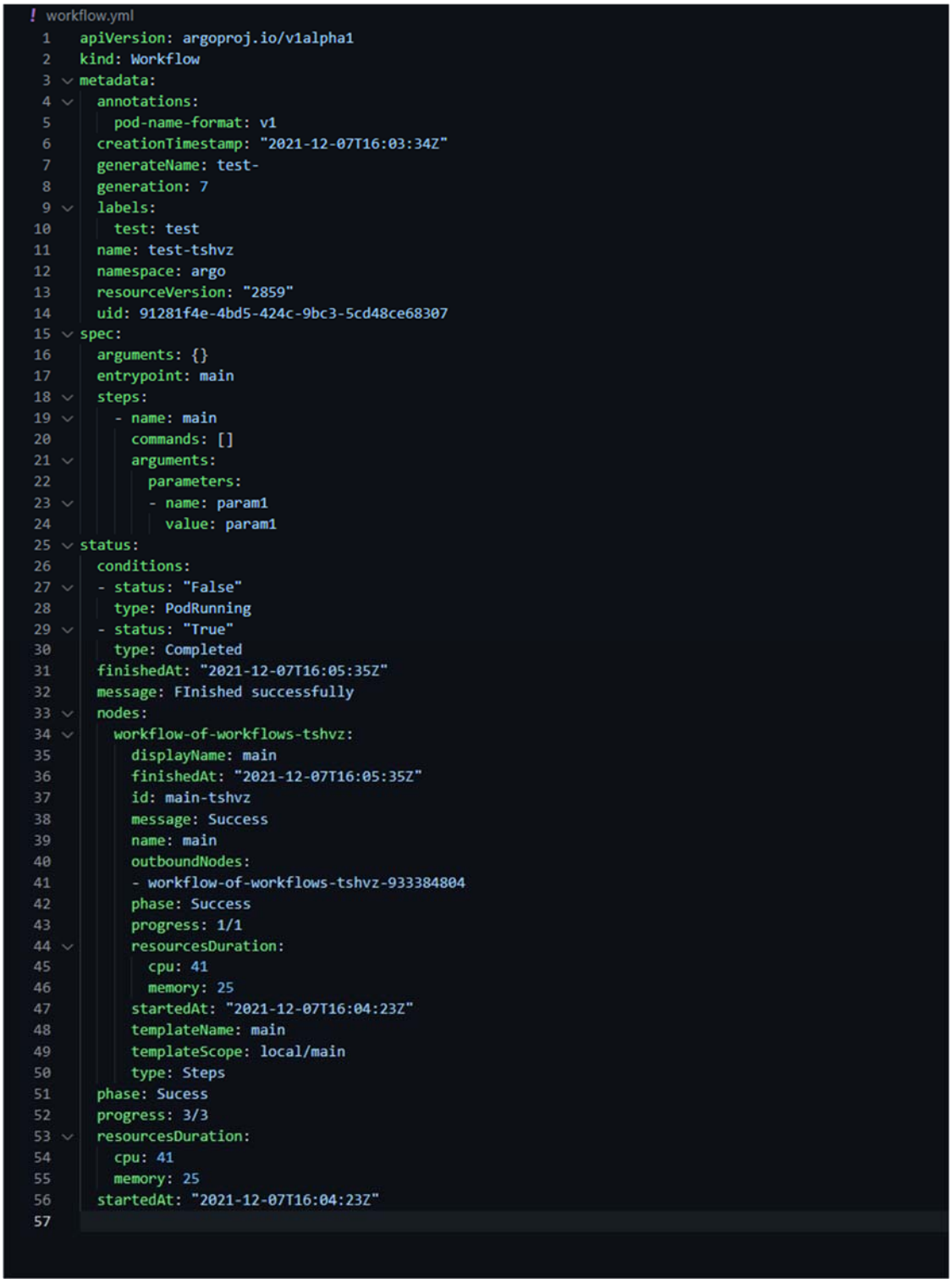
Спеціальний ресурс — це розширення Kubernetes API, яке не обов’язково доступне в стандартній установці Kubernetes. Він являє собою налаштування конкретної інсталяції Kubernetes. Однак багато основних функцій Kubernetes тепер створені за допомогою спеціальних ресурсів, що робить Kubernetes більш модульним. Спеціальні ресурси можуть з’являтися і зникати в запущеному кластері завдяки динамічній реєстрації, а адміністратори кластеру можуть оновлювати власні ресурси незалежно від самого кластера.

Після встановлення спеціального ресурсу користувачі можуть створювати його об’єкти та отримувати доступ до нього за допомогою kubectl, так само, як і для вбудованих ресурсів, таких як Pods.

Самі по собі спеціальні ресурси дозволяють зберігати та отримувати структуровані дані. Коли ви поєднуєте користувацький ресурс із користувацьким контролером, користувацькі ресурси забезпечують справжній декларативний API. Декларативний API Kubernetes забезпечує поділ відповідальності. Ви оголошуєте потрібний стан свого ресурсу. Контролер Kubernetes синхронізує поточний стан об’єктів Kubernetes із вашим оголошеним бажаним станом. Це на відміну від імперативного API, де ви інструктуєте сервер, що робити.

Метою створення Workflow ресурсу у кластері є збереження прогресу виконання пайплайну у випадку припинення його роботи з будь-яких причин. Приклад ресурсу наведено на рисунку 3.16.

Workflow містить інформацію про задачу та прогрес її виконання.



workflow.yml

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  annotations:
    pod-name-format: v1
  creationTimestamp: "2021-12-07T16:03:34Z"
  generateName: test-
  generation: 7
  labels:
    test: test
  name: test-tshvz
  namespace: argo
  resourceVersion: "2859"
  uid: 91281f4e-4bd5-424c-9bc3-5cd48ce68307
spec:
  arguments: {}
  entrypoint: main
  steps:
    - name: main
      commands: []
      arguments:
        parameters:
        - name: param1
          value: param1
status:
  conditions:
  - status: "False"
    type: PodRunning
  - status: "True"
    type: Completed
  finishedAt: "2021-12-07T16:05:35Z"
  message: FInished successfully
  nodes:
    workflow-of-workflows-tshvz:
      displayName: main
      finishedAt: "2021-12-07T16:05:35Z"
      id: main-tshvz
      message: Success
      name: main
      outboundNodes:
      - workflow-of-workflows-tshvz-933384804
      phase: Success
      progress: 1/1
      resourcesDuration:
        cpu: 41
        memory: 25
      startedAt: "2021-12-07T16:04:23Z"
      templateName: main
      templateScope: local/main
      type: Steps
  phase: Sucess
  progress: 3/3
  resourcesDuration:
    cpu: 41
    memory: 25
  startedAt: "2021-12-07T16:04:23Z"
```

Рисунок 3.16 – приклад створеного ресурсу Workflow

Під час розробки серверного додатку було використано TypeScript, що надає можливості статичної типізації коду, який в результаті буде компільовано в NodeJS.

В якості серверного фреймворку було використано NestJS.

Nest (NestJS) — це платформа для створення ефективних, масштабованих додатків Node.js на стороні сервера. Він використовує прогресивний JavaScript, побудований на основі TypeScript і повністю підтримує його (проте все ще дозволяє розробникам писати на чистому JavaScript) і поєднує елементи ООП (об’єктно-орієнтоване програмування), ФП (функціональне програмування) і ФРП (функціональне реактивне програмування).

Ініцціалізацію NestJS дотаку зображено на рисунку 3.17:

```
import { ValidationPipe } from '@nestjs/common';
import { NestFactory } from '@nestjs/core';
import { useContainer } from 'class-validator';
import { TrimStringsPipe } from './modules/common/transformer/trim-strings.pipe';
import { AppModule } from './modules/main/app.module';
import { setupSwagger } from './swagger';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  setupSwagger(app);
  app.enableCors();
  app.useGlobalPipes(new TrimStringsPipe(), new ValidationPipe());
  useContainer(app.select(AppModule), { fallbackOnErrors: true });
  await app.listen(3000);
}
bootstrap();
```

Рисунок 3.17 – Ініціалізація додатку, старт серверу

При роботі з NestJS для автентифікації запитів використовуються Guards. Guards - це класи які імпліментують метод canActivate. Приклад створеного Guard продемонстровано на рисунку 3.18

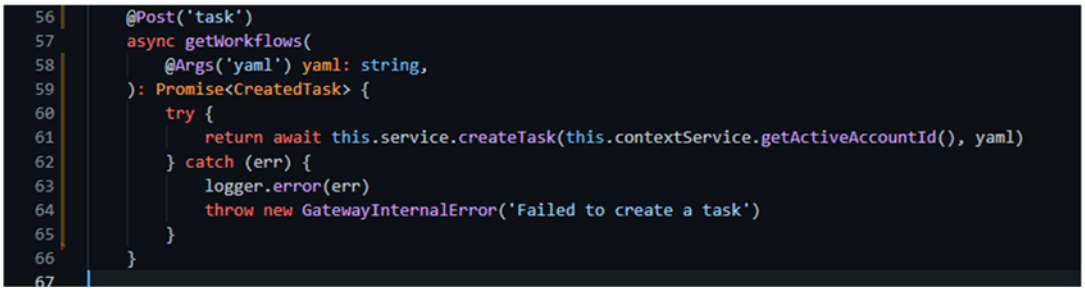
```
@Injectable()
export class RolesGuard implements CanActivate {
    constructor(
        private reflector: Reflector
    ) {}

    private static isUserPermitted(permittedRoles: string[] = [], userRoles: string[] = []): boolean {
        return userRoles.some(role => permittedRoles.includes(role))
    }

    canActivate(context: ExecutionContext): boolean {
        const permittedRoles = this.reflector.get<string[]>('roles', context.getHandler())
        if (isEmpty(permittedRoles)) return true

        const authEntity = AuthenticatedEntity.extractAuthEntityFromRequestContext(context)
        const userRoles = authEntity.isAccountAdmin() ? [ Roles.ADMIN, Roles.USER ] : [ Roles.USER ]

        return RolesGuard.isUserPermitted(permittedRoles, userRoles)
    }
}
```

Рисунок 3.18 – Реалізація Guard в NestJS

В рамках серверу було створено 2 ендпоінти для створення задач (рисунок 3.19), та зчитування їх результатів (рисунок 3.20).

```
@Get('tasks')
async getTasks(
    @Args ('runtime') runtime: string,
): Promise<DbTask | undefined> {
    try {
        const res = await this.service.findMany({
            filter: {
                account: this.contextService.getActiveAccountId(),
                runtime: runtime,
                phase: "WAITING_FOR_EXECUTION"
            },
            sort: {
                //  old one first
                created_at: 1
            }
        })
    } catch (err) {
        logger.error(err)
        throw new GatewayInternalError('Failed to fetch tasks for runtime ' + runtime)
    }
}
```

Рисунок 3.19 – Обробка запиту на отримування задач

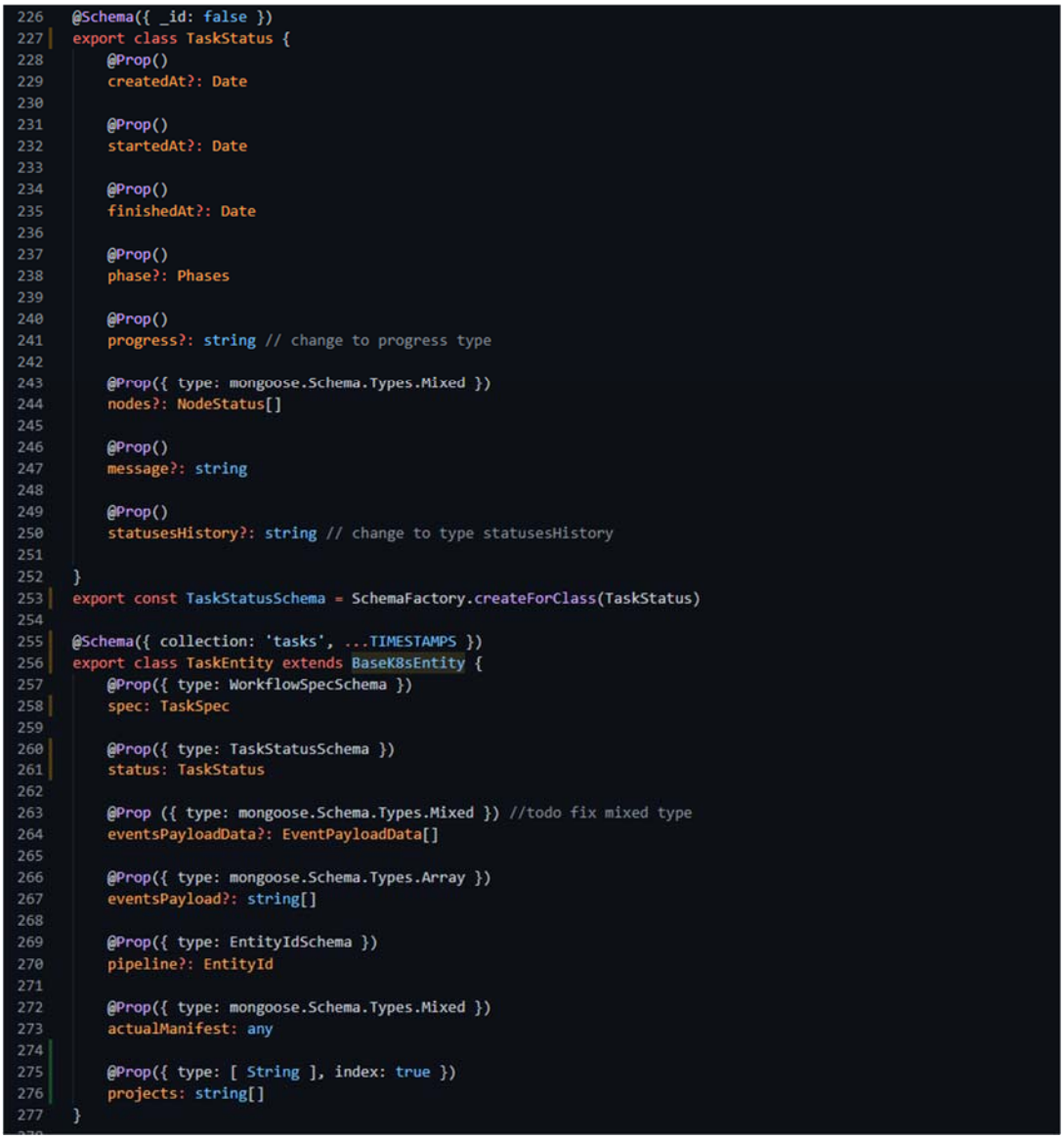


```
@Post('task')
async getWorkflows(
    @Args('yaml') yaml: string,
): Promise<CreatedTask> {
    try {
        return await this.service.createTask(this.contextService.getActiveAccountId(), yaml)
    } catch (err) {
        logger.error(err)
        throw new GatewayInternalError('Failed to create a task')
    }
}
```

Рисунок 3.20 – Обробка запиту на створення задачі

Працювати з БД можна і напряму, за допомогою нативного драйвера MongoDB, проте набагато краще використовувати ODM.

ODM (англ. Object-Document Mapping, укр. Об'єктно-документне відображення, або перетворення) - технологія програмування, яка дозволяє зв'язати БД з концепціями об'єктно-орієнтованих мов програмування, створюючи «віртуальну об'єктну базу даних». Готове та популярне рішення для Node.js та NestJS це Mongoose. Приклад задачі описаної за допомогою Mongoose схеми зображено на рисунку 3.21:



```
@Schema({ _id: false })
export class TaskStatus {
    @Prop()
    createdAt?: Date

    @Prop()
    startedAt?: Date

    @Prop()
    finishedAt?: Date

    @Prop()
    phase?: Phases

    @Prop()
    progress?: string // change to progress type

    @Prop({ type: mongoose.Schema.Types.Mixed })
    nodes?: NodeStatus[]

    @Prop()
    message?: string

    @Prop()
    statusesHistory?: string // change to type statusesHistory

}
export const TaskStatusSchema = SchemaFactory.createForClass(TaskStatus)

@Schema({ collection: 'tasks', ...TIMESTAMPS })
export class TaskEntity extends BaseK8sEntity {
    @Prop({ type: WorkflowSpecSchema })
    spec: TaskSpec

    @Prop({ type: TaskStatusSchema })
    status: TaskStatus

    @Prop ({ type: mongoose.Schema.Types.Mixed }) //todo fix mixed type
    eventsPayloadData?: EventPayloadData[]

    @Prop({ type: mongoose.Schema.Types.Array })
    eventsPayload?: string[]

    @Prop({ type: EntityIdSchema })
    pipeline?: EntityId

    @Prop({ type: mongoose.Schema.Types.Mixed })
    actualManifest: any

    @Prop({ type: [ String ], index: true })
    projects: string[]
}
```

Рисунок 3.21 – Mongoose схема задачі

Для успішного розгортування серверу необхідно описати Dockerfile й створити імедж, на основі якого в майбутньому будуть створені контейнери серверу. Приклад Dockerfile наведено на рисунку 3.22:

```
1   FROM node:14.17.0-alpine as base
2   WORKDIR /code
3
4   FROM base as build
5
6   COPY lerna.json .
7   COPY tsconfig.json .
8   COPY package.json .
9   COPY yarn.lock .
10
11  COPY libs libs
12
13  ARG PACKAGE
14  COPY ${PACKAGE} ${PACKAGE}
15
16  # Install with dev dependencies
17  RUN yarn install
18  RUN yarn build
19  # Clean up dev dependencies
20  RUN yarn install --production
21
22  FROM base
23  ARG PACKAGE
24
25  COPY --from=build /code/libs /code/libs
26  COPY --from=build /code/node_modules /code/node_modules
27  COPY --from=build /code/${PACKAGE} /code/${PACKAGE}
28
29  WORKDIR /code/${PACKAGE}
30  CMD [ "yarn", "start:prod" ]
31
```

Рисунок 3.22 – Dockerfile серверу

Як веб сервер було обрано Nginx. Використання проксі-сервера, Nginx, дозволяє рівномірно розподілити навантаження, зручно та правильно працювати зі статичними ресурсами і реалізувати протокол Transport Layer Security (TLS).

NGINX прискорює доставку вмісту та додатків, покращує безпеку, полегшує доступність і масштабованість для найбільш завантажених веб-сайтів в Інтернеті.

Оскільки ми використовуємо контейнеризацію, принцип дій буде приблизно аналогічний, проте засоби відрізнятимуться. Nginx буде встановлений за допомогою готового image для Docker.

3.4. Тестування програмного забезпечення

Тестування програмного забезпечення - процес технічних досліджень, призначений для отримання інформації про якість програмних продуктів; методи контролю якості, які використовують остаточний набір тестів, вибраних певним чином, для перевірки відповідності між фактичною та очікуваною поведінкою програми [20].

При тестуванні програмних систем застосовуються методи "білої" та "чорної скриньки". Ці методи – динамічні методи тестування програмного забезпечення.

Метод «білої скриньки» тестує внутрішні структури або роботу програми, на відміну від її функціональності (тобто тестування в чорному ящику). Для створення тестів використовуються внутрішні перспективи системи, а також навички програмування.

Зазвичай тестування «білої скриньки» засноване на аналізі керуючої структури програми. Програма вважається повністю перевіреною, якщо проведено вичерпне тестування маршрутів (шляхів) її графа управління [21].

Методи тестування тестування в білій коробці включають такі критерії покриття коду:

- тестування контрольного потоку;
- тестування потоку даних;
- гілкове тестування;
- покриття виразів;
- покриття умов;
- змінена умова/покриття рішення;
- тестування основного шляху;

- тестування побічних шляхів.

Тестування «чорної скриньки» - це метод тестування програмного забезпечення, який вивчає функціональність програми, не заглядаючи у її внутрішні структури чи роботи. Цей метод тестування може бути застосований практично до кожного рівня тестування програмного забезпечення: блочного, інтеграційного, системного. Іноді це називається тестуванням на основі специфікацій.

Тестування «чорної скриньки» забезпечує пошук наступних категорій помилок:

- некоректних чи відсутніх функцій;
- помилок у зовнішніх структурах даних або в доступі до зовнішньої бази даних;
- помилок характеристик.

Розроблений програмний продукт перевірятиметься на відповідність технічним вимогам, безпечність, коректність роботи у середовищі Windows 10 за допомогою браузера Google Chrome.

Першим кроком для роботи з програмним продуктом є встановлення агенту у кластері користувача. Це можна зробити за допомогою маніфестів зображених на рисунках 3.23 та 3.24.

```
50  apiVersion: v1
51  kind: ConfigMap
52  metadata:
53    name: agent-config
54    namespace: default
55    resourceVersion: "516"
56    uid: b4952dc3-d670-11e5-8cd0-68f728db1985
57  data:
58    apiToken: <api-token>
59    interval: <interval>
60
```

Рисунок 3.23 – ConfigMap агента

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    version: 1.0
  name: runner-agent
  namespace: default
spec:
  replicas: 1
  revisionHistoryLimit: 5
  strategy:
    rollingUpdate:
      maxSurge: 50%
      maxUnavailable: 50%
    type: RollingUpdate
  template:
    spec:
      volumes:
        - name: runnerconf
          secret:
            secretName: runnerconf
      containers:
        image: melden19/runner-agent
        ports:
        - containerPort: 8080
          protocol: TCP
        readinessProbe:
          httpGet:
            path: /health
            port: 8080
          periodSeconds: 5
          timeoutSeconds: 5
          successThreshold: 1
          failureThreshold: 5
        volumeMounts:
        - name: runnerconf
          mountPath: "/etc/secrets"
          readOnly: true
        imagePullPolicy: Always
        name: {{ .AppName }}
        resources:
{{ toYaml .Runner.Resources | indent 10 }}
      securityContext:
        runAsUser: 10001
        runAsGroup: 10001
        fsGroup: 10001
```

Рисунок 3.24 –Deployment агента

Протестуємо відкриття вебсайту у браузері. (рисунок 3.25).

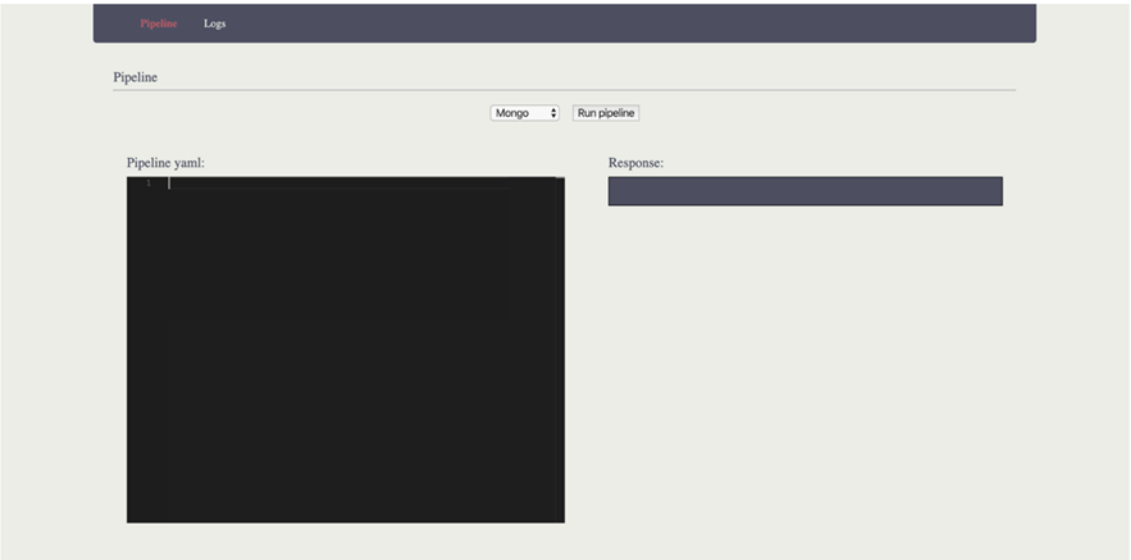


Рисунок 3.25 – Сторінка Pipeline

В блоці Pipeline yaml потрібно ввести інформацію про контейнери(steps) які бажаємо запустити у форматі yaml й, вибравши бажане сховище з випадаючого списку, натиснути «Run pipeline» (рисунок 3.26).

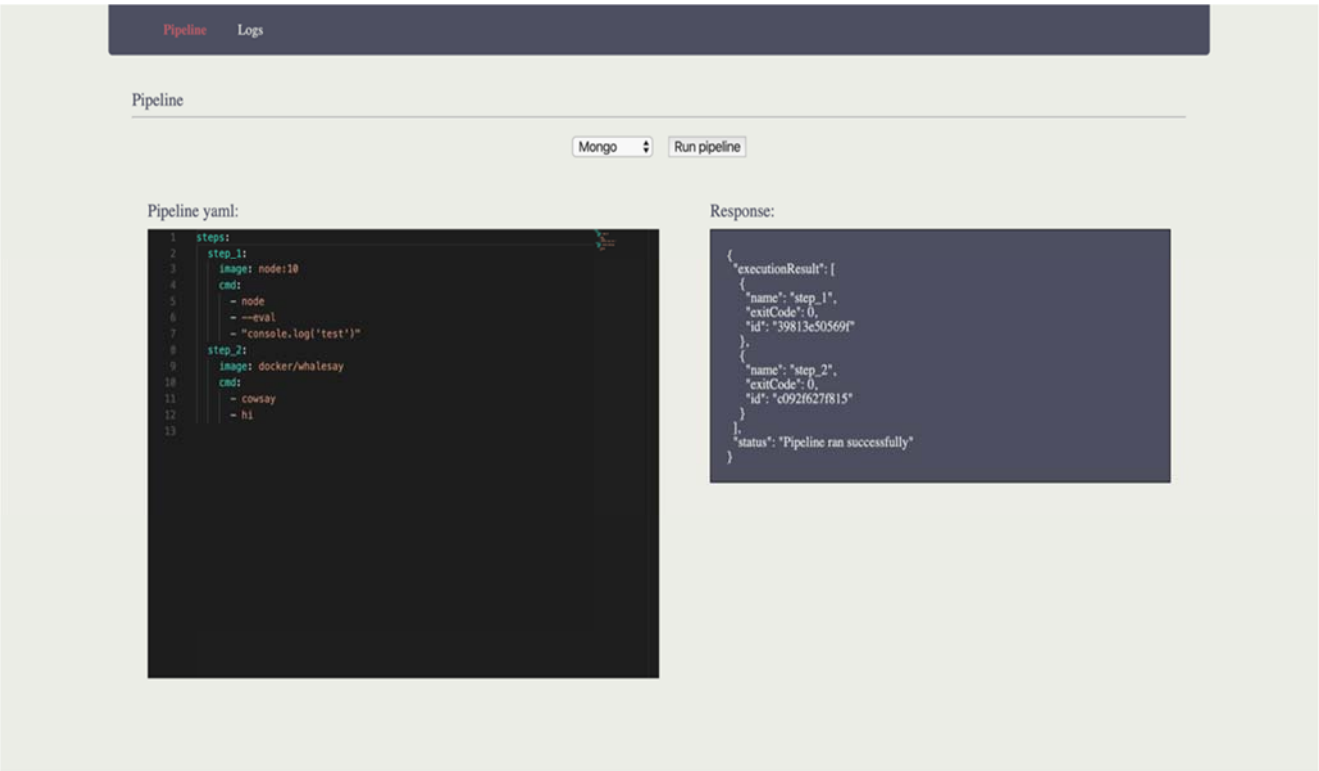


Рисунок 3.26 – Сторінка Pipeline після виконання пайплайну

Після цього було створено задачу у базі даних MongoDB. Далі встановлений агент повинен буде отримати цю задачу і розпочати її виконувати. На рисунку 3.27 зображено логи робочого агенту. По них видно, що він з фіксованим інтервалом намагається отримати задачу з серверу.

Після отримання задачі буде розпочато її виконання. Під час виконання задачі буде створено CRD Workflow з описом пайплайну. Відповідно до нього агент буде послідовно виконувати кроки пайплайну. Протягом циклу виконання кроків агент буде оновлювати ресурс workflow з метою збереження прогресу виконання пайплайну на випадок несподіваного припинення його роботи й початку роботи з початку.

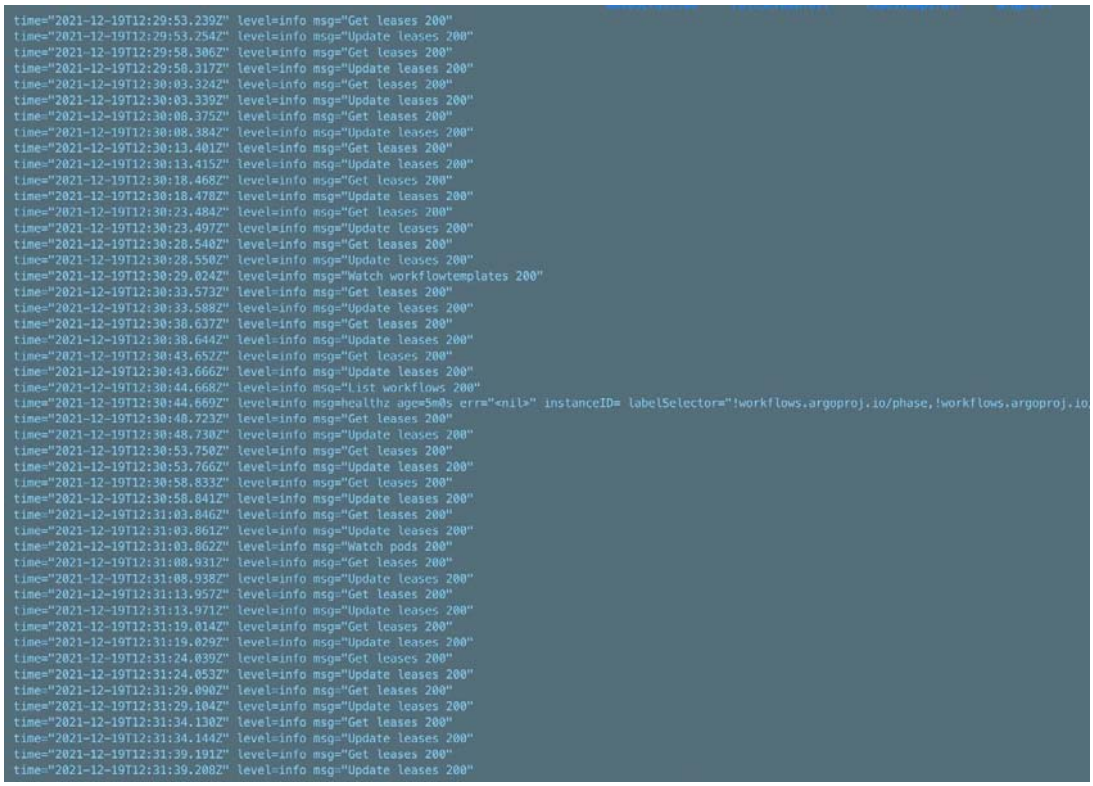


Рисунок 3.27 – Логи роботи агенту

Переконатися, що ресурс було створено без помилок й агент розпочав його виконання можна за допомогою команд зображених на рисунку 3.28 та 3.29.

Рисунок 3.28 – отримання ресурсу workflow у фазі Running

Далі для кожного кроку буде запущено Pod, в якому буде виконуватися 2 контейнера:

- контейнер крок;
- контейнер слухач логів.

Перевірити чи кроки було виконано успішно можна за допомогою команди kubectl get po | grep pipeline-1, результат виконання якої зображено на рисунку 3.30.

```yaml
apiVersion: v1alpha1
kind: Workflow
metadata:
  name: pipeline-1
spec:
  entrypoint: steps
  templates:
  - name: steps
    steps:
    - - name: step-1
        template: step-1
    - - name: step-2
        template: step-2
  - name: step-1
    container:
      image: node:10
      command: [node]
      args: ["--eval", "console.log('test')"]
      resources:
        limits:
          memory: 32Mi
          cpu: 100m
  - name: step-2
    container:
      image: docker/whalesay
      command: [cowsay]
      args: ["hello world"]
      resources:
        limits:
          memory: 32Mi
          cpu: 100m
```

Рисунок 3.29 – Створений ресурс Workflow

```
~ ❯ k get pods | grep pipeline-1
pipeline-1-3893458983                    0/2     Completed   0          4m50s
pipeline-1-809788873                     0/2     Completed   0          5m
~ ❯
```

Рисунок 3.30 – Отримання списку завершених кроків (Pods)

Отримавши маніфест одного з Pods за допомогою команди “kubectl get po/pipeline-1-809788873 -o yaml “ (рис. 3.31) можна побачити, що було успішно виконано 2 контейнера.

```yaml
apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubernetes.io/psp: eks.privileged
    steps-name: step-1
  creationTimestamp: "2021-12-19T11:56:18Z"
  name: pipeline-1-809788873
  namespace: aws-local-runtime
  resourceVersion: "814562"
  uid: 880bc800-470d-4d51-8aa3-f668ce43a8f6
spec:
  containers:
  - command:
    - listen
    image: melden19/container-logger
    name: logger
    volumeMounts:
    - mountPath: /var/run/docker.sock
      name: dockersock
  - args:
    - --eval
    - console.log('test')
    command:
    - node
    image: node:10
    imagePullPolicy: IfNotPresent
    name: step-1
    volumeMounts:
      - mountPath: /data
        name: data
        subPath: stepo
  volumes:
  - name: data
    persistentVolumeClaim:
      claimName: my-pvc
  - name: dockersock
    hostPath:
      path: /var/run/docker.sock
```

Рисунок 3.31 – Отримання маніфесту завершеного кроку (Pod)

Тепер повернувшись до клієнта й перейшовши на сторінку «Logs» у поле «Step name» вводимо ім’я кроку й вибираємо сховище, у цьому випадку «Mongo» (рисунок 3.32).

Рисунок 3.32 – Сторінка Logs

Натиснувши кнопку «Get logs» можемо отримати вихідні дані контейнері відповідно до сказаного id та сховища (рисунок 3.33).

Рисунок 3.33 – Відображення вихідних даних кроку

З метою перевірки правильності роботи програмного модуля намагаємося вручну запустити такий самий контейнер й звірити вихідні дані. Після виконання цього отримуємо такий результат (рисунок 3.34).

Рисунок 3.34 – Відображення вихідних даних контейнера під час його запуска з командної стрічки

Також варто перевірити чи дійсно вихідні дані кроку було збережено в вибране сховище. Шукаємо вихідні дані в базі MongoDB (рисунок 3.35).

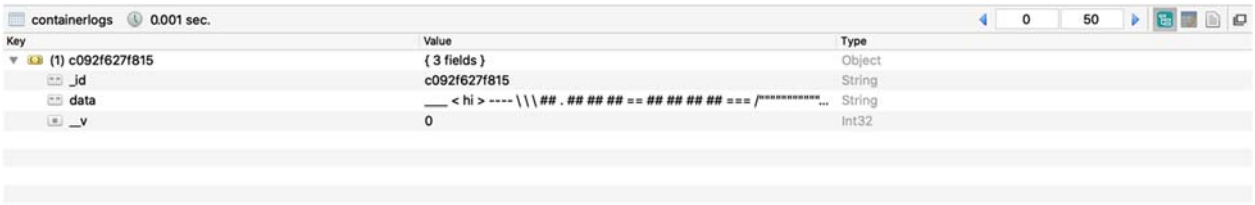


Рисунок 3.35 – Запис в базі який відповідає вихіним даним кроку

Отже проаналізувавши вище наведені рисунки можна зробити висновок що програмний продукт працює коректно. Тестування програмного продукту показало повну відповідність поставленому технічному завданню.

Оскільки підбір мови програмування опирався на швидкість й невимогливість в ресурсах, програмний модуль не є вимогливим потужної технічної конфігурації комп’ютера. Найбільше на швидкість роботи буде впливати частота роботи процесора. Так як Node.js є подійно-орієнтованою мовою програмування й працює лише з одним потоком, важлива саме максимальна частота ядра процесора.

Протестувавши програмну систему можна виділити таку послідовність її використання:

1. Реєстрація на платформі.
2. Встановлення агенту на власний Kubernetes кластер.
3. Опис пайплайну, вибір типу сховища.
4. Запуск пайплайну.
5. Очікування завершення пайплайну.
6. Отримання id запущених контейнерів.
7. Введення назви кроку, вибір типу сховища.
8. Отримання логів.

У таблиці 3.1 та 3.2 наведено мінімальну та рекомендовану конфігурацію персонального комп’ютера для запуску програми.

Таблиця 3.1

Мінімальна конфігурація:

Тип процесора	32-розрядний (x86) або 64-розрядний (x64) процесор з тактовою частотою 1 ГГц
Об’єм оперативної пам’яті	1 ГБ для 32-розрядної системи і 2 ГБ для 64-разрядної системи
Розмір жорсткого диску	16 ГБ для 32-розрядної системи і 22 ГБ для 64-розрядної системи

Таблиця 3.2

Рекомендована конфігурація:

Тип процесора	32-розрядний (x86) або 64-розрядний (x64) процесор з тактовою частотою 2 ГГц
Об’єм оперативної пам’яті	2 ГБ для 32-розрядної системи і 4 ГБ для 64-разрядної системи
Розмір жорсткого диску	20 ГБ для 32-розрядної системи і 25 ГБ для 64-розрядної системи

Програма повинна працювати під управлінням сімейства операційних систем Linux та MacOS. Для роботи з програмою потрібно встановити NodeJS та npm пакети вказані у файлі package.json. Після цього користувач може запускати і працювати з програмним модулем.

Висновки до розділу III

В цьому розділі було здійснено варіантний аналіз і обґрунтування вибору засобів реалізації програмного продукту. В наслідок чого було обрано мови програмування NodeJS та Golang, середовище розробки Visual Studio Code, база даних MongoDB, nginx в якості зворотнього проксі серверу.

Також було розроблено такі модулі програмного додатку як: прослуховування та логування вихідних даних кроків, виконання пайплайну в середовищі Kubernetes та серверну частину додатку.

Тестування програми показало повну її працездатність та відповідність поставленому технічному завданню. Розроблено інструкцію користувача по використаню й встановленню програмного продукту.

ВИСНОВКИ ТА ПРОПОЗИЦІЇ

Під час виконання кваліфікаційної роботи було розроблено програмне забезпечення у середовищі Kubernetes для інтегрування програмних продуктів.

Було проаналізовано стан даної проблеми на сьогоднішній день. Розглянуто основні аналоги, визначено їх особливості та недоліки і розроблено порівняння з власною програмною системою.

Для серверної частини було обрано мову програмування Nodejs. Для агенту встановленого в Kubernetes кластер користувача було обрано мову Go.

Були вирішенні наступні задачі:

- проведено аналіз існуючих рішень і засобів з автоматизації інтеграції та розгортання програмного коду;
- розроблено розгалужену архітектуру програмної системи;
- розроблено алгоритм взаємодії сервера й Kubernetes кластера;
- розроблено алгоритм виконання пайплайну в середовищі Kubernetes;
- розроблено алгоритм збереження вихідних даних кроків;
- розроблено інтерфейс програмного продукту;
- розроблено систему інтеграції програмного продукту в режимі реально часу.

Було розроблено архітектуру програмної системи та модель виконання пайплайну у приватному кластері користувача, які дозволили запобігти поширенню конфіденційних даних з сервером, у випадку коли пайплайн повинен мати до них доступ. Це було реалізовано за допомогою агента, який повинен бути встановленим у кластері користувача.

Тестування програмної системи довело повну її працездатність та відповідність поставленому технічному завданню. Розроблено інструкцію користувача.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Belmont J. M. Hands-On Continuous Integration and Delivery/J. M. Belmont – New York: Packt, 2018, 416c.
2. CI/CD Pipeline. [Електронний ресурс] – Режим доступу: https://semaphoreci.com/blog/cicd-pipeline
3. Kubernetes. [Електронний ресурс] – Режим доступу: https://kubernetes.io/
4. Kubernetes wiki. [Електронний ресурс] – Режим доступу: https://en.wikipedia.org/wiki/Kubernetes
5. Hudson (Software). [Електронний ресурс] – Режим доступу: https://en.wikipedia.org/wiki/Hudson_(software)
6. Jenkins [Електронний ресурс] – Режим доступу: https://www.jenkins.io/
7. An Introduction to Kubernetes [Електронний ресурс] – Режим доступу: https://github.com/Leverege/kubernetes-book/blob/master/An%20Introduction%20to%20Kubernetes%20%5BFeb%202019%5D.pdf
8. Страндартні потоки [Електронний ресурс] – Режим доступу: https://uk.wikipedia.org/wiki/%D0%A1%D1%82%D0%B0%D0%BD%D0%B4%D0%B0%D1%80%D1%82%D0%BD%D1%96_%D0%BF%D0%BE%D1%82%D0%BE%D0%BA%D0%B8
9. Daniel P. Understanding the Linux Kernel, Second Edition/P. Daniel, C. Marco – California: O'Reilly Media, 2002. – 950 c.
10. Загальна характеристика інформаційного забезпечення [Електронний ресурс] – Режим доступу: https://library.if.ua/book/80/5658.html
11. Купер А. Интерфейс. Основы проектирования взаимодействия. 4-е изд./А. Купер – Санкт-Петербург:Питер, 2016. – 534 с.

12. Алгоритм [Електронний ресурс] – Режим доступу: https://uk.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC
13. Графічний інтерфейс користувача [Електронний ресурс] – Режим доступу: https://en.wikipedia.org/wiki/Graphical_user_interface
14. Подійно-орієнтоване програмування [Електронний ресурс] – Режим доступу: https://en.wikipedia.org/wiki/Event-driven_programming
15. Event based VS Thread based server sys. [Електронний ресурс] – Режим доступу: https://stackoverflow.com/questions/25280207/what-are-the-differences-between-event-driven-and-thread-based-server-system
16. Node.js [Електронний ресурс] – https://ru.wikipedia.org/wiki/Node.js
17. libsigc++ [Електронний ресурс] – https://developer.gnome.org/libsigc++/stable/
18. Go programming language [Електронний ресурс] – https://en.wikipedia.org/wiki/Go_(programming_language)
19. MongoDB [Електронний ресурс] – https://github.com/night-codes/The-Little-MongoDB-Book-rus
20. Software testing [Електронний ресурс] – https://en.wikipedia.org/wiki/Software_testing

ДОДАТОК А
ДЕКЛАРАЦІЯ ДОБРОЧЕСНОСТІ

Я, Юзефович Ігор Михайлович, підтверджую, що сам написав цю роботу і не використовував жодних інших, окрім цитованих, джерел інформації.

Дослівні вирази або фрази, які цитуються, позначаються як такі; інші недослівні запозичення чи ремінісценції, наведені в тексті цієї роботи, містять актуальну інформацію щодо первинних джерел наведеного контенту. Робота у цій безпосередньо або змістовно аналогічній формі не була раніше публікована чи оприлюднена. Усе зазначене вище посвідчую власноручним підписом.

(підпис) (дата)

ДОДАТОК Б

ЛІСТИНГ ОСНОВНИХ МОДУЛІВ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

PipelineRunner.js

```
const { readFileSync } = require('fs');
const { execSync } = require('child_process');
const _ = require('lodash');
const yaml = require('js-yaml');
const uuid = require('uuid/v4');
const { docker, containerTimeout } = require('../../config');
const startListener = require('./startListener');
const createStorageLayer = require('../storageLayers');
const { normalizeId, logContainer } = require('../helpers');
const Logger = require('./ContainerLogger');

const STATIC_STEP_LABEL = 'codefresh-assessment-pipeline-step';

class PipelineRunner {
  constructor(settings = {}) {
    const { path, pipeline, storageLayer }  = settings;
    this.path = path || './pipeline.yml';
    this.pipeline = pipeline;
    this.storageLayer = storageLayer;

    //  to prevent step name conflicts
    this.pipelineSalt = uuid();
    //  label for startListener to listen for
    this.stepLabel = `${STATIC_STEP_LABEL}-${this.pipelineSalt}`;
  }

  async run() {
    const executionResult = [];

    try {
      const pipeline = this._loadPipeline();
      this._validatePipeline(pipeline);

      await this._initListener();

      for (const [stepName, stepMeta] of Object.entries(pipeline.steps)) {

        const container = await this._runStep({ name: stepName, ...stepMeta });
        normalizeId(container);
        executionResult.push({ name: stepName, ...container });

        if (container.exitCode > 0) {
```

```
            if (container.exitCode === 143) {
              throw new Error(`Step ${stepName} was terminated cause of predefined max
execution timeout: ${containerTimeout / 1000} sec`);
            }
            throw new Error(`Pipeline failed on step ${stepName} with status code
${container.statusCode}`);
          }
        }
        console.log('Pipeline ran successfully');
        this._gracefulShutdown();
        return { executionResult, status: 'Pipeline ran successfully' };

      } catch (err) {
        console.error('Failed to run pipeline cause of: ', err);
        this._gracefulShutdown();
        return { executionResult, status: `Failed to run pipeline cause of: ${err}`, error: true };
      }
    }

    async _runStep(step) {
      const { image: Image, cmd: Cmd, name } = step;

      try {
        const container = await docker.createContainer({
          name: `${this.pipelineSalt}-${name}`,
          Image,
          Cmd,
          Labels: {
            meta: this.stepLabel
          }
        });
        const output = await container.start();
        let running = true;

        if (output.Error) {
          throw output.Error;
        }

        setTimeout(() => {
          if (running) {
            container.stop().catch((err) => {
              console.error('Container timeout termination failed', err);
            });
          }
        }, containerTimeout);

        const res = await container.wait();
        running = false;
```

```
            return {
              exitCode: res.StatusCode,
              id: container.id
            };
          } catch (err) {
            throw new Error(`Failed to run step cause of: ${err}`);
          }
        }

        _loadPipeline() {
          try {
            if (this.pipeline) {
              return this.pipeline;
            }
            const file = readFileSync(this.path);
            return yaml.safeLoad(file);
          } catch (err) {
            throw new Error(`Failed to load pipeline cause of: ${err}`);
          }
        }

        _validatePipeline({ steps }) {
          if (!_.keys(steps)) {
            throw new Error('Validation failed');
          }
        }

        _initListener() {
          return startListener.start({ labels: [`meta=${this.stepLabel}`] }, async (container) => {
            normalizeId(container);
            logContainer('Attaching to container', container);

            const storage = createStorageLayer(this.storageLayer, 'writer', container);

            const logger = new Logger(container.id, storage);
            await logger.run();
          });
        }

        _gracefulShutdown() {
          execSync('yes | docker container prune');
          startListener.stop();
        }
      }

      module.exports = PipelineRunner;
```

StartListener.js

```
const _ = require('lodash');
const { docker } = require('../../config');

class StartListener {
  start(selector, containerHandler) {
    return new Promise((resolve, reject) => {
      const filters = this._getApiFilter(selector);
      docker.getEvents({ filters }, (err, events) => {
        if (err) {
          reject(err);
        } else {
          console.log('Listening for starting containers...');
          this.events = events;
          events.on('data', (chunk) => {
            const containerData = JSON.parse(chunk.toString());
            if (this._matchLocalFilters(containerData, selector)) {
              containerHandler(containerData)
            }
          });
          resolve();
        }
      });
    });
  }

  stop() {
    if (this.events) {
      this.events.destroy();
      this.events = null;
      console.log('No longer listen to docker `start` events');
    } else {
      console.log('StartListener already stopped');
    }
  }



  _matchLocalFilters(eventData, selector) {
    const name = _.get(eventData, 'Actor.Attributes.name');
    if (selector.nameRegex && name) {
      return (new RegExp(selector.nameRegex)).test(name);
    }
    return true;
  }

  _getApiFilter(selector) {
    const filter = {};
```

```
    _.set(filter, 'event', ['start']);

    if (selector.labels) {
      _.set(filter, 'label', _.castArray(selector.labels));
    }

    return JSON.stringify(filter);
  }
}

module.exports = new StartListener();
```

ContainerLogger.js

```
const { promisify } = require('util');
const { logContainer } = require('../helpers');
const { docker } = require('../../config');

class ContainerLogger {
  constructor(id, storage) {
    this.id = id;
    this.storage = storage;
    this.container = docker.getContainer(this.id);
  }

  async run() {
    try {
      await this._initStorageLayer();
      await this._exec();
    } catch(err) {
      console.error('Failed to run ContainerLogger', err);
    }
  }

  _initStorageLayer() {
    return this.storage.init(this.id);
  }

  async _exec() {
    const stream = await this._attachToContainer();

    stream.on('end', () => {
      this.storage.stream.end();
      logContainer('Container finished execution', { id: this.container.id });
    });

    this.container.modem.demuxStream(stream, this.storage.stream, this.storage.stream);
  }
```

```
  _attachToContainer() {
    const attach = promisify(this.container.attach.bind(this.container));
    return attach({stream: true, stdout: true, stderr: true, logs: true});
  }
}

module.exports = ContainerLogger;
```

App.js

```
const http = require('http');
const express = require('express');
const bodyParser = require('body-parser');
const logger = require('morgan');
const mongoose = require('mongoose');
const chalk = require('chalk');

const { mongo } = require('../config');
const index = require('./api/index');

const app = express();
const server = http.createServer(app);

/**
 * Connect to MongoDB.
 */
mongoose.set('useFindAndModify', false);
mongoose.set('useCreateIndex', true);
mongoose.set('useNewUrlParser', true);
mongoose.set('useUnifiedTopology', true);
mongoose.connect(mongo.uri)
  .then(() => console.log(`CONNECTED TO ${mongo.uri}`))
  .catch(err => {
    console.error(err);
    console.log('%s MongoDB connection error. Please make sure MongoDB is running.',
chalk.red('✗'));
    process.exit();
  });

/**
 * Express configuration.
 */
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(function(req, res, next) {
  res.header("Access-Control-Allow-Origin", "*"); // update to match the domain you will
```

```
make the request from
  res.header("Access-Control-Allow-Headers", "Origin, X-Requested-With, Content-Type,
Accept");
  next();
});

app.use('/api', index);

app.use(function(req, res, next) {
  const err = new Error('Not Found');
  err.status = 404;
  next(err);
});

app.use(function(err, req, res, next) {
  res.status(err.status || 500);
  res.json({
    message: err.message,
    error: err
  });
});

server.listen(process.env.PORT || 8080);

module.exports = app;
```

FS.reader.js

```
const { existsSync, readFileSync } = require('fs');
const BaseStorageLayer = require('../Base');
const { fsLayerPath, notFoundMessage } = require('../constants');

class FSStorageRead extends BaseStorageLayer.Reader {
  async read() {
    const logPath = `${fsLayerPath}/${this.container.id}.log`;
    if (existsSync(logPath)) {
      return readFileSync(logPath).toString();
    } else {
      throw new Error(notFoundMessage);
    }
  }
}

module.exports = FSStorageRead;
```

FS.writer.js

```
const fs = require('fs');
```

```
const BaseStorageLayer = require('../Base');
const { delay } = require('../../helpers');
const { fsLayerPath } = require('../constants');

class FSStorageWrite extends BaseStorageLayer.Writer {
  write(log) {
    if (!this.writeStream) {
      const writePath = `${fsLayerPath}/${this.container.id}.log`;
      if (!fs.existsSync(fsLayerPath)) {
        fs.mkdirSync(fsLayerPath, { recursive: true });
      }
      this.writeStream = fs.createWriteStream(writePath);
    }

    this.writeStream.write(log);
  }
}

module.exports = FSStorageWrite;
```

Mongo.reader.js

```
const mongoose = require('mongoose');

const BaseStorageLayer = require('../Base');
const ContainerLog = require('./Log');
const { mongo } = require('../../../config');
const { notFoundMessage } = require('../constants');

class MongoStorageRead extends BaseStorageLayer.Reader {
  async init() {
    if (mongoose.connection.readyState === 0) {
      return this._initMongo();
    }
  }

  async read() {
    const { id } = this.container;
    const res = await ContainerLog.findById(id);
    if (res)  {
      return res.data;
    } else {
      throw new Error(notFoundMessage);
    }
  }

  _initMongo() {
    return mongoose.connect(mongo.uri, { useNewUrlParser: true, useUnifiedTopology: true
});
```

```
  }
}

module.exports = MongoStorageRead;
```

Mongo.writer.js

```
const mongoose = require('mongoose');

const BaseStorageLayer = require('../Base');
const ContainerLog = require('./Log');
const { mongo } = require('../../../config');
const { getImageAndName } = require('../../helpers');

class MongoStorageWrite extends BaseStorageLayer.Writer {
  onFinish() {
    this.containerLog.save().then(() => console.log('Saved successfully!')).catch(console.error);
  }

  async init() {
    this.containerLog = new ContainerLog({
      _id: this.container.id,
      data: '',
      ...getImageAndName(this.container)
    });

    if (mongoose.connection.readyState === 0) {
      return this._initMongo();
    }
  }

  write(log) {
    this.containerLog.data += log;
  }

  _initMongo() {
    return mongoose.connect(mongo.uri, { useNewUrlParser: true, useUnifiedTopology: true
});
  }
}

module.exports = MongoStorageWrite;
```

Pipelines.jsx

```
import React, { useContext, useCallback } from 'react';
import MonacoEditor from 'react-monaco-editor';
```

```
import BaseLayout from '../BaseLayout';
import { Store } from '../../App.store';
import './style.scss';

const Pipeline = () => {
  const store = useContext(Store);
  const { editor, response, error, loading, storageLayer } = store.pipeline;

  const handlerStorageChange = useCallback(e => {
    storageLayer.set(e.target.value);
  }, [storageLayer]);

  const runPipeline = useCallback(() => {
    loading.set(true);

    const base64Pipeline = window.btoa(editor.value);

    fetch('http://localhost:8080/api/pipelines/run', {
      body: JSON.stringify({
        pipeline: base64Pipeline,
        storageLayer: storageLayer.value
      }),
      headers: {
        'Content-Type': 'application/json',
      },
      method: 'POST'
    })
      .then(res => res.json())
      .then(response.set)
      .catch(error.set)
      .finally(() => {
        loading.set(false);
      })
  }, [editor, storageLayer]);

  return (
    <BaseLayout title="Pipeline" className="pipeline-page">
      <div className="controlls">
        <select value={storageLayer.value} onChange={handlerStorageChange}>
          <option value="fs">File system</option>
          <option value="mongo">Mongo</option>
        </select>

        <button onClick={runPipeline}>Run pipeline</button>
      </div>
      <div className="wrapper">
        <div style={{ width: '50%' }}>
          <p className="solid-label">Pipeline yaml:</p>
```

```
            <MonacoEditor
              width="100%"
              height="500"
              language="yaml"
              theme="vs-dark"
              value={editor.value}
              onChange={editor.set}
            />
          </div>
          <div>
            <p className="solid-label">Response:</p>
            <div className="result response">
              {loading.value
                ? 'Running...'
                : error.value
                  ? <div className="error">
                    {error.value.toString()}
                  </div>
                  : <div>
                    {typeof response.value === 'string' ? response.value :
JSON.stringify(response.value, null, 2)}
                  </div>
              }
            </div>
          </div>
        </div>
      </BaseLayout>
    )
}

export default Pipeline;
```

Logs.jsx

```
import React, { useCallback, useContext } from 'react'
import BaseLayout from '../BaseLayout'
import { Store } from '../../App.store';
import './style.scss'

const Logs = () => {
  const store = useContext(Store)
  const { container, error, loading, response, storageLayer } = store.logs;

  const handlerChange = useCallback(e => {
    container.set(e.target.value);
  }, [container]);
  const handlerStorageChange = useCallback(e => {
    storageLayer.set(e.target.value);
  }, [storageLayer]);
```

```
  const handleGetLogs = useCallback(() => {
    loading.set(true);
    error.set(null);
    response.set('');

fetch(`http://localhost:8080/api/containers/${container.value}/logs?storageLayer=${storageL
ayer.value}`)
      .then(res => res.json())
      .then(response.set)
      .catch((err) => {
        console.error(err);
        error.set(err);
      })
      .finally(() => loading.set(false));
  }, [container, storageLayer]);

  return (
    <BaseLayout title="Logs" className="logs-page">
      <div className="controls">
        <div>
          <label htmlFor="logs_select">Storage Layer</label>
          <select id="logs_select" value={storageLayer.value}
onChange={handlerStorageChange}>
            <option value="fs">File system</option>
            <option value="mongo">Mongo</option>
          </select>
        </div>
        <div>
          <label htmlFor="logs_input">Container ID</label>
          <input id="logs_input" type="text" value={container.value}
onChange={handlerChange}/>
        </div>
        <button onClick={handleGetLogs}>{loading.value ? 'Loading...' : 'Get logs'}</button>
      </div>
      <p className="solid-label">Response:</p>
      <div className="response">
        {error.value ? <div className="error">
          {error.value.toString()}
        </div> : <div>
          {typeof response.value === 'string' ? response.value :
JSON.stringify(response.value, null, 2)}
        </div>}
      </div>
    </BaseLayout>
  )
}
```

```
export default Logs;
```

BaseLayout.jsx

```
import React from 'react'

const BaseLayout = ({ title, children, className }) => {
  return (
    <div className={className}>
      <h1>{title}</h1>
      <hr/>
      <section>
        {children}
      </section>
    </div>
  )
}
```

ДОДАТОК В
КОПІЯ ПУБЛІКАЦІЇ