Державний вищий навчальний заклад
“Прикарпатський національний університет імені Василя Стефаника”
Кафедра інформаційних технологій

УДК 004

ДИПЛОМНИЙ ПРОЕКТ

Тема Автоматизація процесів розробки та впровадження програмного забезпечення за допомогою DevOps методології

Спеціальність 121 “Інженерія програмного забезпечення”
код і назва спеціальності

ПОЯСНЮВАЛЬНА ЗАПИСКА

ДП.ІПЗ-08.ПЗ
(позначення)

Рецензент
проф. Кузь М.В.
(посада) (підпис) (дата) (розшифровка підпису)

Студент
ІПЗ-41 Іванків Р.Б.
(шифр групи) (підпис) (дата) (розшифровка підпису)

Нормоконтролер
проф. Кузь М.В.
(посада) (підпис) (дата) (розшифровка підпису)

Керівник дипломного проекту
доц. Лазарович І.М.
(посада) (підпис) (дата) (розшифровка підпису)

Допускається до захисту

Завідувач кафедри
доц. Козленко М.І.
(посада) (підпис) (дата) (розшифровка підпису)

2020
(рік)

Державний вищий навчальний заклад
«Прикарпатський національний університет імені Василя Стефаника»
Факультет математики та інформатики Кафедра інформаційних технологій
Спеціальність 121 Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Завідувач кафедри Козленко М. І.
студенту Іванківу Ростиславу Богдановичу
„_____”_____ 20___р.

ЗАВДАННЯ
НА ВИКОНАННЯ ДИПЛОМНОГО ПРОЕКТУ

Студенту Іванківу Ростиславу Богдановичу
(прізвище, ім’я, по батькові студента)
1. Тема проекту Автоматизація процесів розробки та впровадження програмного забезпечення за допомогою DevOps методології затверджена розпорядженням по факультету математики та інформатики від „ 25 ” жовтня 2019 р. № 7
2. Термін здачі студентом закінченого проекту 22 травня 2020 р.
3. Вихідні дані до дипломного проекту P2675 DevOps – стандарт побудови надійних систем, включаючи їх збірку та розгортання, CI / CD практики, документація GCP, AWS, Docker, Jenkins, Github Actions
4. Зміст пояснювальної записки (перелік питань, що їх належить опрацювати)
1. Аналіз предметної області. Постановка задачі
2. Моделювання. Розробка структури проекту
3. Реалізація
4. Обгрунтування економічної доцільності проекту
5. Перелік графічного матеріалу (з точним забезпеченням обов’язкових креслень) “Мета, актуальність, цілі роботи”, “Освновна ідея”, “Огляд попередньої інфраструктури”, “Розгляд проблем старого підходу”, “Огляд нової інфраструктури”, “Порівняння підходів”, “Технології проекту”, “Git Workflow”, “CI / CD flow”, “Jenkins pipeline”, “Github Actions pipelines”, “Backend CD”, “Frontend CD”, “Висновки”
6. Дата видачі завдання 11.09.2019

Керівник _____ Лазарович І.М.
(підпис) (розшифровка підпису)
Завдання прийняв до виконання _____ Іванків Р.Б.
(підпис) (розшифровка підпису)

КАЛЕНДАРНИЙ ПЛАН

Номер і назва етапів дипломного проекту	Термін виконання етапів проекту	Примітка
1. Дослідження інфраструктури	20.10.2019	Виконав
2. Огляд моделей	21.12.2019	Виконав
3. Вибір інструментів	05.01.2020	Виконав
4. Вдосконалення системи	25.03.2020	Виконав
5. Підведення підсумків	14.04.2020	Виконав
6. Оформлення роботи	16.05.2020	Виконав

Студент _____ Іванків Р.Б.
(підпис) (розшифровка підпису)

Керівник проекту _____ Лазарович І.М.
(підпис) (розшифровка підпису)

РЕФЕРАТ

Пояснювальна записка: 102 сторінки (без додатків), 139 рисунків, 0 таблиць, 35 джерел, 7 додатків на 11 сторінках.

Ключові слова: ПЗ, Автоматизація, DevOps, CI / CD, Jenkins, Github Actions, Pipeline, Docker, AWS, GCP, VM instance, RDS, Git, Github, Dockerhub, Web Application, Spring, Maven, Angular.

Об`єктом дослідження є впровадження CI / CD процесів в існуючу інфраструктуру аплікацій та їх практична реалізація за допомогою готових засобів.

Мета роботи: автоматизувати процеси перевірки та доставки коду до кінцевого користувача.

У даному дипломному проекті, було досліджено та імплементовано автоматизовану систему процесів створення, тестування та випуску програмного забезпечення, яка відбувається за допомогою адаптації різного роду готових Pipeline, що в свою чергу дозволяють описати повторюваний сценарій перевірки та доставки готового продукту до користувачів. При цьому адаптація не займає багато часу, а частини таких сценаріїв можна перевикористати в інших проектах.

ABSTRACT

Explanatory note: 102 pages (without appendix), 139 figures, 0 tables, 35 references, 7 appendix on 11 pages.

Key words: Software, Automation, DevOps, CI / CD, Jenkins, Github Actions, Pipeline, Docker, AWS, GCP, VM instance, RDS, Git, Github, Dockerhub, Web Application, Spring, Maven, Angular.

Object of study: implementation of CI / CD processes in the existing application infrastructure and their practical implementation using ready-made tools.

Brief description of the text of the explanatory note:

A purpose is to automate the processes of verification and delivery of code to the end user.

In this diploma project, was investigated and implemented the process automation system of software creation, testing, releasing, which takes place through the adaptation of various kinds of ready-made Pipelines, which in turn allows describing the repetitive checking and delivery script a ready product to users. However, the adaptation does not take much time, and some of these scenarios can be reused in other projects.

ЗМІСТ

ПЕРЕЛІК ОСНОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ПЗ	програмне забезпечення
DevOps	Development + Operations
CI	Continuous Integration
CD	Continuous Delivery
VM	Virtual Machine
GCP	Google Cloud Platform
AWS	Amazon Web Services
RDS	Relational Database Service
SSH	Secure SHell
HTTP	HyperText Transfer Protocol
WAR	Web Application ARchive
JAR	Java ARchive

ВСТУП

Тенденції розвитку галузі

За оцінками компанії IBM біля 70% IT-бюджету витрачається на підтримку існуючої інфраструктури і тільки 30% на її розширення. І чим далі, тим гостріше бізнес потребує раціоналізації використання IT-ресурсів і автоматизації їх управління. Отож, будь-яка автоматизації в області IT, а особливо процесів розробки ПЗ є явищем необхідним.

Проблеми та завдання, які необхідно вирішити

- Зменшення витрат людських ресурсів
- Автоматизація перевірки якості коду
- Автоматизація розгортання аплікацій

Проблема полягає в тому, що IT-ринок завжди змушений шукати рішення мінімізації своїх витрат в різних його проявах. Існує багато рутинної роботи в межах команд, де це робиться вручну та, відповідно до цього, відбувається оплата роботи працівника. Проте, оплата регулярної роботи людині за годину праці, вочевидь, є більшою за оплату автоматизації цього процесу та подальшої підтримки. В той же момент, потрібно швидко передавати ПЗ в експлуатацію, а також робити це якимось надійним способом, без особливих ризиків.

Оцінка сучасного стану конкретного інженерного завдання

Інженерне завдання в даний момент набуло популярності, оскільки, це вирішує проблеми бізнесу та нормалізації його процесів. На думку багатьох експертів рішення цього завдання постійно буде видозмінюватись і прогресувати і буде в подальшому затребуваним.

Актуальність завдання

При потребі, цим підходом можна вирішити різні види рутинних процесів та запобігти витратам часу на їх повторення. Отже, вирішення цього завдання допоможе як і стороні бізнесу, так і стороні підприємства.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ. ПОСТАНОВКА ЗАДАЧІ

1.1 Аналіз літератури із дослідженням особливостей вирішення ідентичних та аналогічних задач

В літературі йдеться про те, що дана методологія дозволяє ще швидше і надійніше передавати програмне забезпечення в експлуатацію, в значній мірі автоматизує впровадження програмного забезпечення і тим самим зменшує ризики, що виникають при переході на нові версії.

Насправді, найвищим пріоритетом для нас має бути задоволення потреб замовника завдяки регулярній і швидкій доставці програмного забезпечення до кінцевого користувача[1].

На думку авторів, можна виділити наступні плюси даної методології:

- Розгортання здійснюється частіше — аж до декількох разів на день. Це скорочує час введення в дію нового функціоналу.

- Часті розгортання також прискорюють отримання відгуків на нові особливості і зміни в коді. Розробникам не доводиться згадувати, що робилося в минулому місяці.

- Щоб розгортання протікало швидше, створення тестового оточення і власне тестування повинні здійснюватися автоматично, інакше на це буде йти занадто багато часу і сил.

- Автоматизація тестування спрощує відтворення помилок. Оскільки під час кожного тесту виконується одна і та ж послідовність дій.

- Автоматизовані тести можуть виконуватися частіше і не вимагають додаткових зусиль. Відповідно, будь-які виправлення постійно тестуються, і будь-які помилки можуть виявлятися до розгортання нового коду в робочому оточенні.

- Ризики, пов'язані з установкою нових версій, суттєво зменшуються за рахунок такого налаштування робочого оточення, яка дає можливість легко відкотитися до старої версії, якщо це буде потрібно.

1.2 Аналіз основних понять та визначень

1.2.1 Автоматизація

Автоматизація — це створення технології та її застосування з метою контролю та моніторингу виробництва, спрощення доставки різних послуг. Вона виконує завдання, які раніше виконувала людина.

1.2.2 Розробка та впровадження ПЗ

Розробка програмного забезпечення — комплекс заходів з інформаційних технологій, присвячених процесу створення, проектування, розгортання та підтримки програмного забезпечення. Це робота, яку щодня виконують розробники, пишучи код для реалізації необхідних функцій.

Впровадження програмного забезпечення — це процес виходу програмного продукту на ринок. Впровадженням називають доставку вашого продукту до кінцевого користувача.

1.2.3 DevOps

DevOps (акронім від англ. development і operations) — низка практик, призначених для пожвавлення взаємодії розробників із фахівцями

інформаційно-технологічного обслуговування(рис. 1.1) та зближення їхніх робочих процесів одне з одним[2].

Рисунок 1.1 — Основні одиниці DevOps[3]

DevOps базується на тісній залежності між розробкою та використанням ПЗ, розташований на стиці 3-х складових(рис. 1.2), допомагає організаціям швидше створювати і оновлювати програмні продукти та послуги.

Рисунок 1.2 — Складові DevOps[4]

1.2.4 CI/CD процеси

Неперервна інтеграція (англ. Continuous Integration (CI)) — практика розробки програмного забезпечення, яка полягає у виконанні частих автоматизованих складань проекту для якнайшвидшого виявлення та вирішення інтеграційних проблем[5]. Поступових перехід до неперервної інтеграції знижує скалдність інтеграції нових змін і робить їх внесення простішими за рахунок виявлення та усунення більшості помилок на ранньому етапі.

На виділеному сервері або за допомогою використання готових сервісів, організовується служба, до завдань якої входять:

- Отримання початкового коду з репозиторію
- Складання проекту
- Виконання тестів
- Розгортання готового проекту
- Відправлення звітів

Переваги:

- Проблеми інтеграції виявляються і виправляються швидше, що є дешевшим рішенням для багатьох проектів
- Регулярна перевірка справності модульних тестів для останніх змін
- Продукт має постійно має стабільну версію
- Привчає розробників вносити меншу кількість змін, проте робити це частіше, що зменшує можливість нанесення шкоди робочій версії ПЗ

Безперервна доставка (англ. Continuous delivery у (CD)) — підхід у програмній інженерії, суть якого полягає в тому, що команди розробляють програмне забезпечення протягом коротких періодів часу, забезпечуючи надійний випуск версії у будь-який час[6].

Метою такого підходу є швидке та часте створення, тестування та випуск програмного забезпечення[7]. Даний підхід в змозі допомогти зменшити вартість впровадження ПЗ, пришвидшити його, а також знизити ризики доставки змін у робочому вигляді, дозволяючи випуск нових версій. Цей процес, зазвичай, є простим та повторюваним(рис. 1.3).

Переваги:

- Зменшення ризиків
- Стабільність доставки коду
- Частота випуску релізів
- Недоліками неперервної інтеграції та безперервної доставки є:
- Витрати на підтримку інфраструктури

- Потенційна необхідність у виділеному сервері

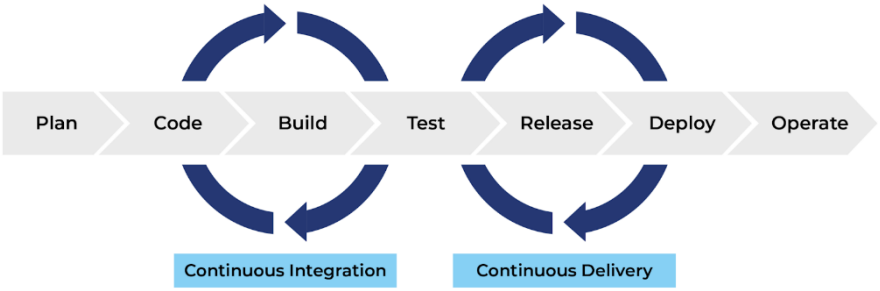


Рисунок 1.3 — Стадії CI / CD процесів[8]

1.2.5 Хмарні платформи та сервіси

Хмара — це велика мережа віддалених серверів по всьому світу, які з'єднані між собою і покликані діяти як єдина екосистема. Ці сервери призначені для зберігання та управління даними, запуску програм чи надання послуг, таких як потокове відео, веб-пошта, різного роду програмне забезпечення. Замість доступу до файлів та даних з локального чи персонального комп’ютера ви отримуєте доступ до них в Інтернеті з будь-якого пристрою через Інтернет, тож інформація буде доступна у будь-якому місці та у будь-який час, коли вам це потрібно. Користувач має доступ до власних даних, але не може управляти і не повинен піклуватися про інфраструктуру, операційну систему і програмне забезпечення, з яким він працює.

1.2.6 Система керування версіями

Система керування версіями або система управління вихідним кодом(SCM) — програмне забезпечення для полегшення роботи зі змінною інформацією[9].

SCM зазвичай використовуються при розробці ПЗ для контролю над змінами у файлах, над змінами яких одночасно працюють декілька людей. Кожна версія має унікальний ідентифікатор, зміни кожного з документів запам’ятовуються. Система керування версіями допускає створення різного роду відгалужень(рис. 1.4), які можуть допомогти при одночасній розробці в команді.

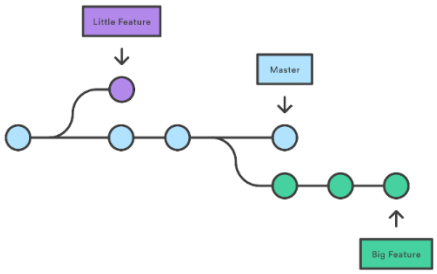


Рисунок 1.4 — Система відгалужень Git[10]

1.2.7 Контейнеризація. Docker

Контейнеризація (віртуалізація на рівні операційної системи, контейнерна віртуалізація) — метод віртуалізації, при якому ядро операційної системи підтримує кілька ізольованих примірників середовища користувача замість одного. Ці екземпляри(контейнери) з точки зору користувача повністю ідентичні окремому екземпляру операційної системи. Для систем на базі Unix ця технологія схожа на поліпшену реалізацію механізму chroot. Ядро забезпечує повну ізольованість контейнерів, тому програми з різних контейнерів не можуть впливати один на одного.

Docker — програмне забезпечення для автоматизації розгортання і управління додатками в середовищах з підтримкою контейнеризації. Дозволяє «упакувати» додаток з усіма його залежностями в контейнер, який може бути перенесений на будь-яку Linux-систему.

Docker Engine — це клієнт-серверна програма із основними компонентами(рис. 1.5):

- dockerd — процес, який запускає docker сервер
- REST API — відкриті API для спілкування з dockerd
- CLI client — сама команда docker з консолі

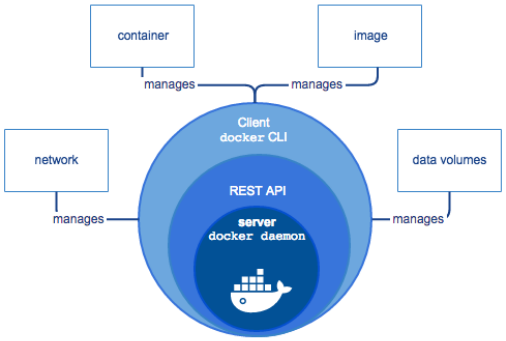


Рисунок 1.5 — Основні компоненти Docker[11]

Переваги:

- Абстрагування додатків від хоста
- Маштабування
- Управління версіями і залежностями
- Ізолювання середовища
- Використання шарів
- Компонування

Недоліки:

- Інколи тяжка настройка
- Зворотна сумісність
- Продуктивність
- Архітектура
- Підтримка

1.3 Огляд існуючих систем та сервісів

1.3.1 CI/CD системи

Jenkins — це інструмент автоматизації з відкритим кодом(рис. 1.6), який є self-hosted, написаний на Java, з системою плагінів, побудованою для цілей постійної інтеграції. Jenkins використовується для безперервного створення та тестування ваших програмних проектів, що полегшує розробникам інтеграцію змін у проект і полегшує випуск нових версій. Це також дозволяє безперервно поставляти програмне забезпечення шляхом інтеграції з великою кількістю технологій тестування та розгортання.

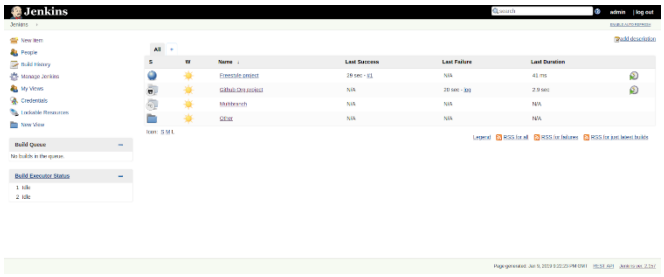


Рисунок 1.6 — Зовнішній вигляд Jenkins

Travis CI — це система безперервної інтеграції та розгортання(рис. 1.7), що надається як сервіс. Цей інструмент створений для проектів з відкритим кодом та орієнтований на CI. Є одним з перших та найбільш вживаним рішенням для відкрити проектів на Github. Для вдосконалення процесу збирання використовується автоматизоване тестування та розроблена система оповіщення. Ви можете швидко протестувати свій код, а Travis буде контролювати всі зміни та повідомить, чи вони були успішними, чи ні. Однак у Travis CI немає безкоштовного плану для приватних репозиторіїв.

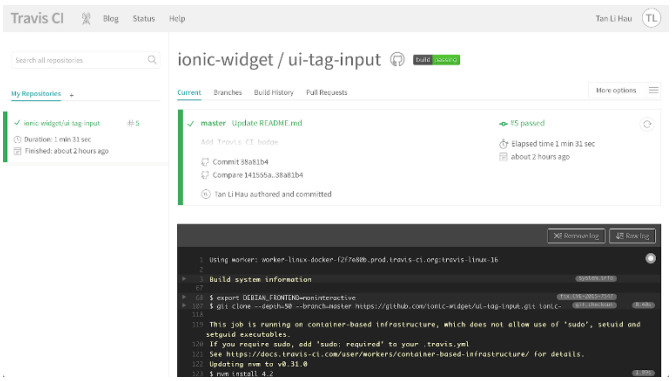


Рисунок 1.7 — Зовнішній вигляд Travis CI

Circle CI — це cloud-based система для безперервної інтеграції(рис. 1.8) та розгортання. Використовуючи unit-тести, інтеграційні тести та функціональні тести, він глибоко зосереджений на тестуванні всіх змін коду. Провідні компанії, такі як Facebook, Kickstarter, Spotify, Lyft, Expedia, успішно ведуть свій процес розвитку з CircleCI. Таким чином їм вдалося прискорити доставку та покращити якість їхньої продукції.

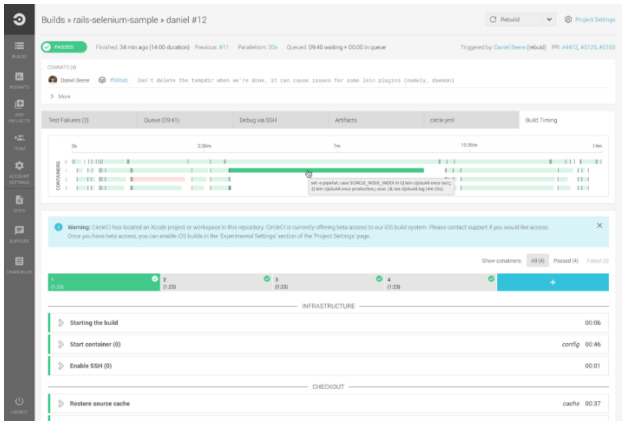


Рисунок 1.8 — Зовнішній вигляд Circle CI

Github actions — це CI / CD система, інтегрована з GitHub(рис. 1.9). Сервіс безкоштовний для відкритих проектів, і навіть для закритих, якщо ваші білди збираються недовго або нечасто. Github Actions як функція, введена недавно в Github, що дозволяє автоматизувати робочий процес. GitHub реагує на різного роду події: оновлення коду, створення релізу, т.п..

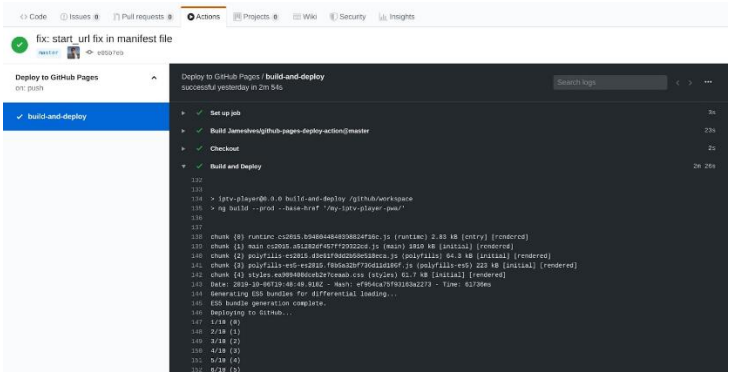


Рисунок 1.9 — Зовнішній вигляд Circle CI

1.3.2 Провайдери готових хмарних рішень

Amazon Web Services або AWS — це хмарна інфраструктура платформи, яка надається компанією Amazon(рис. 1.10). В інфраструктурі представлено безліч сервісів для надання всіляких послуг, наприклад зберігання даних (файл-хостинг), надання обчислювальних потужностей, оренда віртуальних серверів і т.д..

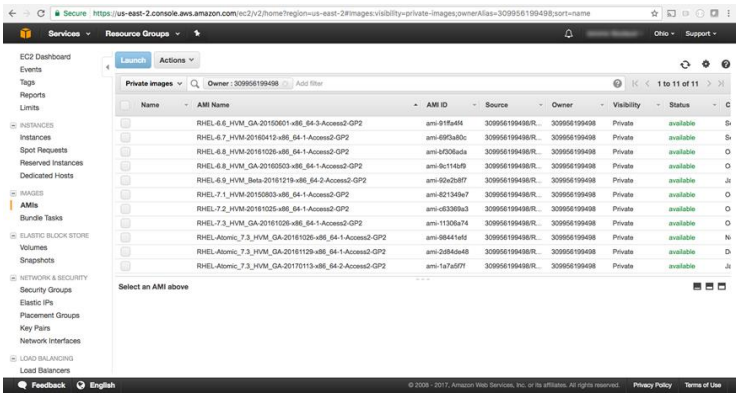


Рисунок 1.10 — Зовнішній вигляд AWS

Google Cloud Platform або GCP — набір хмарних служб, що надається компанією Google(рис. 1.11), які виконуються на тій же самій інфраструктурі, яку Google використовує для своїх продуктів, призначених для кінцевих споживачів, таких як Google Search і YouTube.

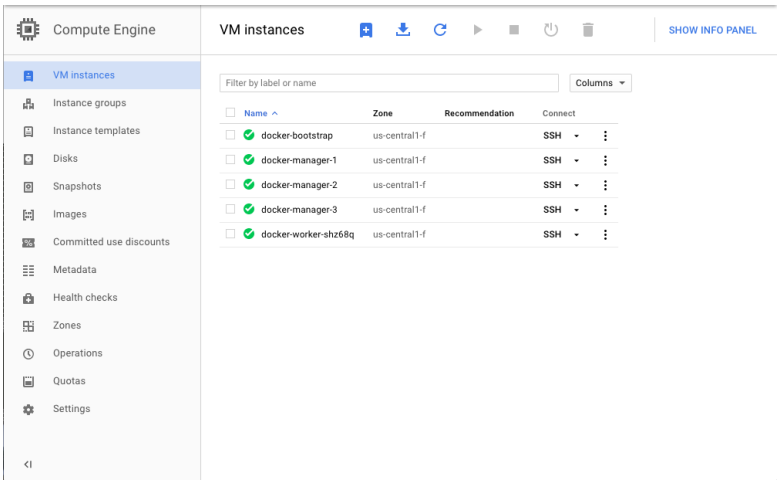


Рисунок 1.11 — Зовнішній вигляд GCP

Microsoft Azure — хмарна платформа компанії Microsoft(рис. 1.12). Надає можливість розробки, виконання додатків і зберігання даних на серверах, розташованих в розподілених дата-центрах.

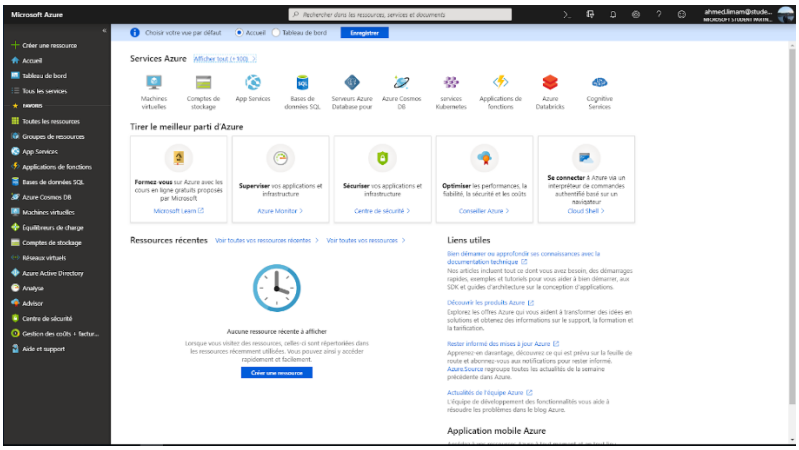


Рисунок 1.12 — Зовнішній вигляд Microsoft Azure

DigitalOcean, Inc. — американський провайдер хмарних інфраструктур(рис. 1.13), з головним офісом в Нью-Йорку і з центрами обробки даних по всьому світу. Це сервіс хмарного хостингу, який робить наголос на швидкість і простоту розгортання та підтримки віртуальних серверів.

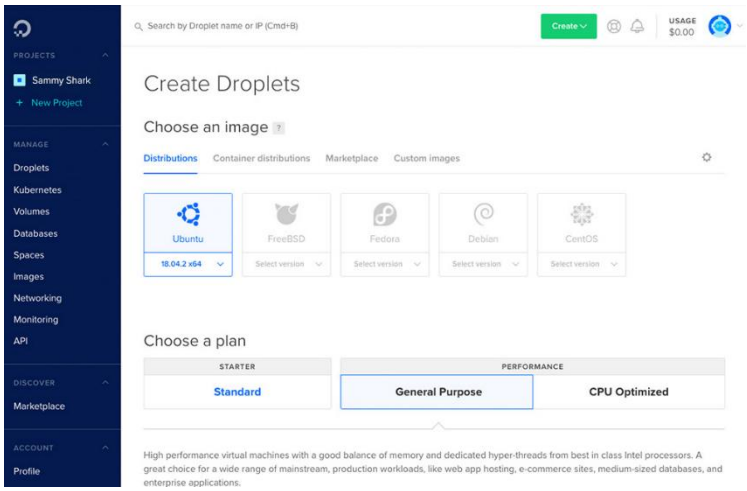


Рисунок 1.13 — Зовнішній вигляд DigitalOcean

1.3.3 Веб-сервіси для спільної розробки програмного забезпечення

GitHub — це платформа, що зберігає різні Git-репозиторії(рис. 1.14) на своїх серверах. Також GitHub називають найбільшим веб-сервісом для хостингу і спільної розробки IT-проектів. Github базується на системі контролю версій Git і розроблений компанією GitHub на Ruby on Rails. Він безкоштовний для тих проектів, які мають відкритий вихідний код. Для великих корпоративних клієнтів доступні платні тарифні плани.

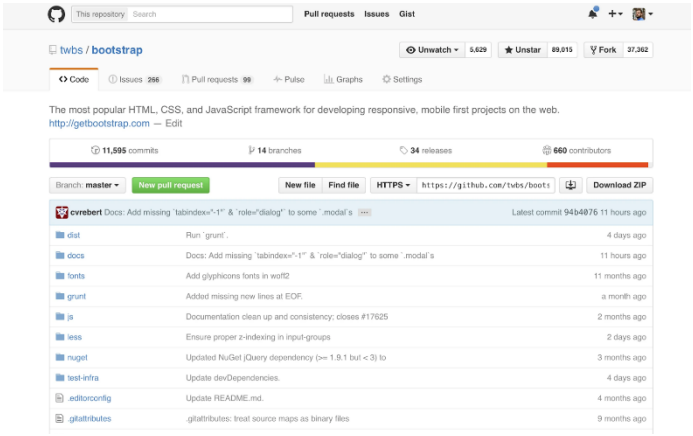


Рисунок 1.14 — Зовнішній вигляд Github

GitLab — це онлайн-сервіс(рис. 1.15), призначений для роботи з git-репозиторіями. Його можна використовувати безпосередньо на офіційному

сайті (gitlab.com), зареєструвавши аккаунт, або встановити і розгорнути на своєму сервері.

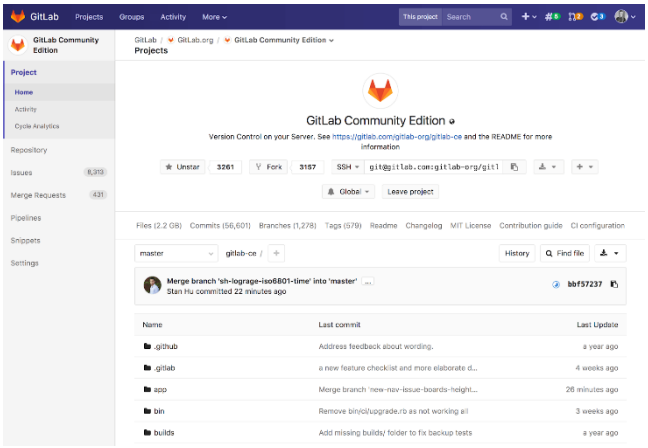


Рисунок 1.15 — Зовнішній вигляд Gitlab

Bitbucket — це веб-сервіс для хостингу проектів і їх спільної розробки(рис. 1.16), на базі системи контролю версій Mercurial і Git. За цілями застосування і функціональності аналогічний GitHub, проте з можливістю мати безкоштовні «закриті» репозиторії для команди.

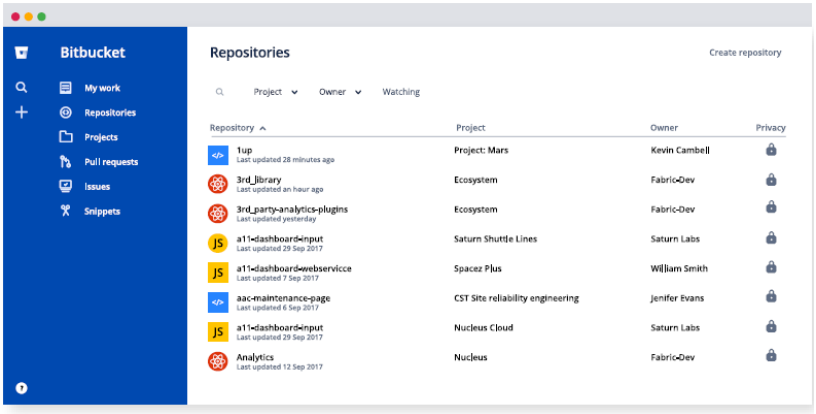


Рисунок 1.16 — Зовнішній вигляд Bitbucket

Gerrit (Gerrit Code Review) — це веб-інструмент для аналізу коду(рис. 1.17), який інтегрований з Git і побудований на основі системи контролю версій Git (допомагає розробникам працювати разом і зберігати історію своєї роботи). Це дозволяє об'єднати зміни в Git-репозиторій, після огляду та перевірки коду.

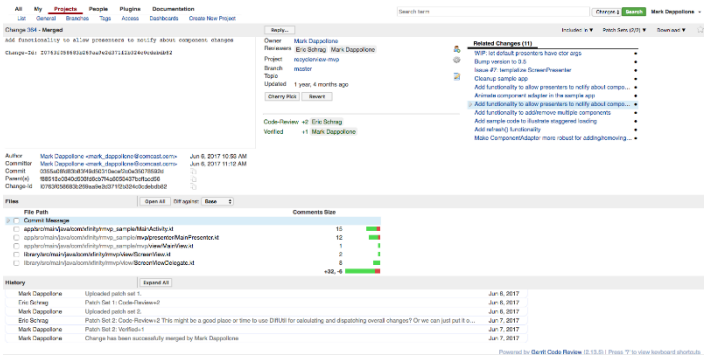


Рисунок 1.17 — Зовнішній вигляд Gerrit

Такі сервіси працюють наступним чином, вони об’єднують локальні репозиторії кожного з розробників в одному місці, який стає віддаленим репозиторієм, зазвичай це виглядає приблизно як зображено на рис. 1.18:

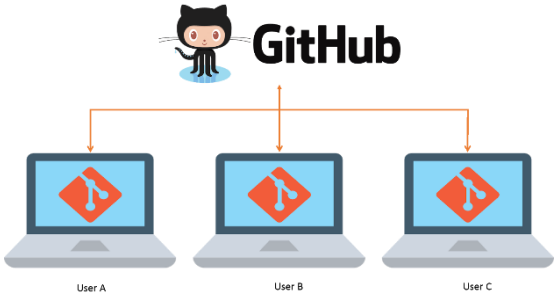


Рисунок 1.18 — Принцип роботи веб-сервісів для спільної розробки програмного забезпечення[12]

1.4 Вирішення аналогічних задач

Перш за все, такі компанії як Google, Netflix, Amazon, Twitter, LinkedIn та ін. базують розробку своїх програм з використанням мікросервісної архітектури. Вони вирішують більш комплексні задачі, які реалізовуваються шляхом впровадження DevOps, оскільки без цього, всі процеси були би набагто складнішими. Унаслідок цього, інженери різних компаній включаючи вищеперераховані, вирішували різного роду аналогічні задачі, які входять до списку тих, що вирішують проблеми підприємства. Відповідно, ці задачі відносяться до роду комерційних, які не підлягають порівнянню.

Проте, існують інші варіанти, які можуть показати сучасні рішення вирішення аналогічних задач.

У книзі автора Rafal Leszko — “Continuous Delivery with Docker and Jenkins: Delivering software at scale”[13] було розглянуто задачі:

- Створення надійних програм за допомогою Docker контейнерів
- Створення повного Pipeline CD за допомогою Jenkins

Також можна розглянути вирішення задачі пов’язаних з Docker у книзі іншого автора — Joseph Muli під назвою “Beginning DevOps with Docker”[14].

Він вирішив завдання які полягали у:

- Створення власних контейнерів Docker
- Оперування контейнерами
- Інтеграція Docker в існуючу інфраструктуру

Певного роду принципи, які використовуються при вирішенні такого роду задач, також було проілюстровано у роботі Gene Kim — “The DevOps Handbook”[15], де згадується про правильні підходи при впровадженні практик та відповідних їх різновидах.

Також Co-Founder & CTO в компанії “Rheo” — Prakash Kumar, проілюстрував деякі аспекти в своїй роботі[16], яка охоплює аспекти:

- Встановлення Jenkins
- Встановлення відповідних плагінів Jenkins
- Налаштування JDK та Maven
- Створення облікових даних Docker Hub
- Створення простого Jenkins Pipeline

Abhinav Dhasmana — Architect компанії “mfine” провів дослідження побудови CI / CD з використанням GitHub Actions[17]. Його метою було:

- Використання Docker замість “bare metal deployment”
- Використання Github Actions для CI
- Імплементація CD для Node.JS аплікацій

1.5 Постановка задачі на дипломний проект

Виходячи з проблем та завдання, які необхідно вирішити, постановка задачі звучить так: мінімізувати витрати підприємства на виконання рутинної роботи, пов’язаної з розробкою та впровадження ПЗ, шляхом її автоматизації. Впровадити CI/CD практики для полегшення процесів інтеграції та доставки коду.

Завдання складається з таких частин:

1) Перенесення проектів(бекенд та фронтенд) в Github

- Створення бекенд, фронтенд проектів
- Налаштування правил для специфічних віток
- Використання Code Review системи

2) Створення 2-х VM(Centos 7) в Google Cloud Platform для подальших потреб

- Створення загального образу на основі Centos 7 для подальшого створення нових VM екземплярів з нього
- Створення інстансу ci-jenkins для хостингу Jenkins CI
- Створення інстансу ims-staging для розгортання наших аплікації
- Встановлення необхідних пакетів

3) Створення клауд БД(RDS) для потреб бекенд аплікації

- Створення RDS на базі MySQL
- Інтеграція з Spring Boot

4) Налаштування Jenkins CI для бекенд частини

- Встановлення Tomcat на VM для подальшого розгортання Jenkins CI як Java Servlet Application
- Налаштування Tomcat для коректної роботи
- Розгортання Jenkins CI
- Зв’язка Jenkins з існуючим Github репозиторієм через webhook
- Створення проекту в CI системі, Multibranch Pipeline для нього

- Створення Jenkinsfile з сценарієм збирання проекту, створенням артефакту та його потенційним розгортанням

5) Налаштування Github actions для фронтенд частини

- Зв’язка з існуючим Github репозиторієм через .github/workflows
- Створення ci.yml, cd.yml з сценаріями збирання та створенням артефакту

6) Інтеграція використання Docker контейнерів в існуючу інфраструктуру

- Додавання Dockerfile до двох проектів
- Створення Docker Hub репозиторіїв для зберігання образів
- Автоматизація розгортання шляхом використання Jenkins

1.6 Підсумки аналізу предметної області

1.6.1 Загальні вимоги

- зменшити витрати людських ресурсів
- добавити автоматичну перевірку коду
- надати можливість легко мігрувати інфраструктуру

1.6.2 Вибір клауд провайдерів

При виборі клауд провайдерів вибір проходив між 2 найбільш популярними варіантами — AWS(рис. 1.19) та Google Cloud Platform(рис. 1.20). Надзвичайно потужні платформи, які можуть надати велику кількість послуг для будь-яких потреб. Для наших потреб для початку знадобиться декілька екземплярів віртуальних машин та сервіс БД.

Екземпляр віртуальної машини на AWS — EC2, а в GCP — Compute Engine VM instance. Сервіс БД на AWS — RDS, GCP — Cloud SQL.

Рисунок 1.19 — Популярні AWS сервіси[18]

Рисунок 1.20 — Популярні GCP сервіси[19]

Оскільки більшість базових сервісів, які нам потрібні є в обох з вищеперерахованих платформ, а стабільність та безперебійність роботи гарантують обидві, то для вибору варто порівняти їхні ціни та можливі додаткові функції.

Порівнюючи ціни на екземпляри VM(див. рис. 1.21), можна зробити висновок, що Google Cloud є трохи дешевшим рішенням по даній категорії. Так як це в нашому випадку тривала інтеграція, то навіть незначна різниця в ціні може згодом бути важливим фактором. Тому для для цих потреб буде використовуватись GCP.

Google Instance Type	CPU Cores	RAM	AWS Instance Type	CPU Cores	RAM	Google New On-Demand (per hour)	AWS On-Demand (per hour)
n1-standard-1	1	3.75	m3.medium	1	3.75	$ 0.070	$ 0.113
n1-standard-2	2	7.5	m3.large	2	7.5	$ 0.140	$ 0.225
n1-standard-4	4	15	m3.xlarge	4	15	$ 0.280	$ 0.450
n1-standard-8	8	30	m3.2xlarge	8	30	$ 0.560	$ 0.900

Рисунок 1.21 — Порівняння цін на екземпляри VM між GCP та AWS

Щодо клауд БД, то тут вибирати було б простіше через те, що питання ціни порівняти складно(рис. 1.22), а залежати тільки від 1-го клауд провайдера не варто, тож буде прийнято рішення використовувати AWS.

Name	Configuration	HA	Disk	Price
Amazon RDS	db.t2.large (2cpu, 8GB RAM)	Multizone A-Z	100GB SSD (provisioned IOPS)	219$
Google SQL 2nd gen	db-n1-standard-2 (2cpu, 7.5GB RAM)	HA failover replica	100GB SSD	220$

Рисунок 1.22 — Порівняння цін на екземпляри клауд БД між GCP та AWS

Для покриття потреб було використано 2 клауд провайдери: GCP, AWS. Серед переваг GCP — продуманий користувацький інтерфейс, набір багатьох основних сервісів, AWS — найбільша кількість сервісів серед всіх постачальників такого роду послуг.

1.6.3 Вибір CI/CD системи

Для використання в даній роботі було обрано Jenkins, як найбільш уживаний серед перевірених часом та Github actions, як якесь нове рішення, яке не вимагає налаштування складних конфігурацій.

1.6.4 Вибір веб-сервісів для спільної розробки

Для зберігання двох проектів було обрано GitHub, так як він являється найкращим варіантом для команди, оскільки кожен був добре ознайомленим з його функціоналом.

1.6.5 Технології проекту

- Back-end technologies: Spring Boot(Java)
- Build tool: Maven
- DBMS: MySQL on RDS(AWS)
- Front-end: Angular(TypeScript)
- Integration tools: Jenkins, GitHub Actions
- Version control system: Git
- Web-container: Tomcat
- Cloud: Google Cloud Platform
- Virtualization tool: Docker

1.6.6 Опис та обґрунтування технологій проекту

Spring Framework (або коротко Spring) — універсальний фреймворк з відкритим вихідним кодом для Java-платформи. Spring Boot дозволяє легко створювати окремі додатки на основі Spring, які можна «просто запустити» завдяки вбудованому Tomcat в jar файл. Більшість програм Spring Boot потребують дуже малої конфігурації Spring. Тому для швидкого старту дуже просто використовувати саме цей фреймворк, який дає гнучку конфігурацію та просте подальше використання.

Apache Maven — фреймворк для автоматизації збирання проектів на основі опису їх структури в файлах на мові POM (англ. Project Object Model), що є підмножиною XML. Зазвичай стандарт у ролі білдера проекту, тому використати його буде простіше, ніж його найближчий аналог — Gradle.

Angular — це відкрита і вільна платформа для розробки веб-додатків, написана на мові TypeScript, що розробляється командою з компанії Google, а також спільнотою розробників з різних компаній. Angular — це повністю переписаний фреймворк від тієї ж команди, яка написала AngularJS. В нашому випадку, як для команди Java розробників, позитивним основною перевагою була саме типізація, яка включена в цей фреймворк.

Jenkins — провідний сервер автоматизації з відкритим кодом, він пропонує сотні плагінів для підтримки побудови, розгортання та автоматизації будь-якого проекту. Через його гнучкість та велику кількість плагінів, глибокий спектр налаштувань було обрано саме його.

GitHub Actions — сервіс, який дозволяє створювати робочі процеси життєвого циклу розробки програмного забезпечення (SDLC) безпосередньо у вашому сховищі GitHub. Він досить зручний в плані інтеграції, коли у вас проекти лежать саме в межах GitHub, в нашому випадку — так і було. Серед його переваг також — простота у використанні.

Git — найчастіше використовується система управління версіями. Git відслідковує зміни, які ви вносите у файли, тому у нас є запис про зроблене, і ми можемо повернутися до конкретних версій, якщо нам це буде потрібно. Також Git полегшує співпрацю, дозволяючи змінам кількох людей об'єднатись в одному місці. Тож незалежно від того, пишете ви код, який бачите лише ви, чи працюєте як частина команди, Git буде корисний. Якщо порівнювати git та його аналоги — svn, mercurial, то найкращим для використання є саме він, оскільки є розподіленою системою керування версіями та найбільш поширеною серед community.

MySQL — вільна реляційна система управління базами даних. Розробка та підтримка сайта MySQL здійснює корпорація Oracle. Є рішенням для малих і середніх додатків. Гнучкість СУБД MySQL забезпечується підтримкою великої кількості типів таблиць. Спільнотою розробників MySQL створені різні відгалуження коду, такі як Drizzle, OurDelta, Percona Server і MariaDB. У використанні з Spring Boot є готові рішення для зв'язки БД та Java моделей, шляхом використання ORM, які підтримуються великою кількістю людей, що є, вочевидь, перевагою, оскільки при появі будь-яких проблем, можна швидко знайти готове їх вирішення.

Amazon Relational Database Service або Amazon RDS — це розподілена реляційна база даних від Amazon. Це хмарний сервіс, який забезпечує користувачів реляційними базами даних для використання в своїх додатках. Amazon RDS дозволяє виробляти швидке розгортання, просте обслуговування і легке масштабування. Такі складні процеси, як оновлення програмного забезпечення БД, проведення резервного копіювання, повернення до ранніх станів (відновлення) проводяться автоматично. Найбільш застосовуваним підходом використання БД в клауді є саме через налаштування RDS, що і було обрано через легкість інтеграції.

Tomcat(в старих версіях — Catalina) — контейнер сервлетів з відкритим вихідним кодом. Написаний на мові Java. Tomcat дозволяє запускати веб-додатки і містить ряд програм для їх конфігурування. Tomcat використовується в якості самостійного веб-сервера, як сервер контенту в поєднанні з веб-сервером Apache HTTP Server, а також в якості контейнера сервлетів в серверах додатків JBoss і GlassFish. Так як він влаштований в Spring Boot jar артефакт, то вибір був очевидним, оскільки, він є стандартним для інфраструктури більшості Spring Boot аплікацій.

Google Cloud Platform — це загальнодоступна хмарна система, послуги якої постачаються клієнтам на постійній основі за допомогою компонентів сервісу. Перевагою є відносно низькі ціни та гарантія стабільності.

Docker — це відкрита платформа, яка дозволяє пакувати, розробляти, запускати та відправляти програми у середовищах, які називаються контейнерами. За останні кілька років Docker повністю змінив індустрію програмного забезпечення. Контейнери є революційними, оскільки вони дозволяють вам як розробнику працювати з додатками без зайвих проблем з встановленням додаткових залежностей. З Docker робоче середовище швидко налаштується. Таким чином, ви можете витратити свій час, зосереджуючись на коді, який справді має значення для вас, вашої команди та вашої організації. Це полегшує всі процеси створення аплікацій та їх подальшого розгортання, підтримки.

Базуючись на даних технологіях та проектних вимогах, можна з упевненістю підрезюмувати, що задача є підсильною для виконання та, вочевидь, є актуальною. Самі технології не створюють жодних проблем для втілення необхідних постановлених задач, можуть позитивно вплинути на подальшу інтеграцію та підтримку.

2 МОДЕЛЮВАННЯ. РОЗРОБКА СТРУКТУРИ ПРОЕКТУ

2.1 Аналіз основних характеристик досліджуваного процесу

2.1.1 Загальний опис проекту

Проект являє собою систему управління запасами, яка складається з деяких додаткових можливостей. Як користувач я можу організувати всі свої товари, запчастини, предмети на своєму складі чи вдома. Є можливість мати базовий аккаунт та преміальний.

Користувач має можливість зареєструватися, увійти, змінити пароль, редагувати інформацію про профіль, створити організацію та запросити людей до організації електронною поштою, редагувати склади та оновити до преміального аккаунт своєї організації.

Отже, є такі ролі користувачів, як:

- Адміністратор
- Робітник.

При цьому, роль адміністратора включає весь функціонал робітника, тобто розширює його.

Запрошений користувач не має можливості створювати організацію і автоматично призначається до запрошеної організації як робітник.

2.1.2 Базова структура підпроектів

2.1.2.1 Структура Spring Boot аплікацій

Більшість Spring Boot аплікації мають наступну архітектуру(рис. 2.1):

- Controller / RestController — класи, які відповідають за прийняття запитів та їх обробку

- Service — класи, які відповідають за бізнес-логіку програми

- Repository — містять CRUD методи, які використовуються у сервісах

- Model — моделі даних, зазвичай пов'язані з БД, або ж звичайні POJO(Plain Old Java Object) чи DTO(Data Transfer Object) класи.

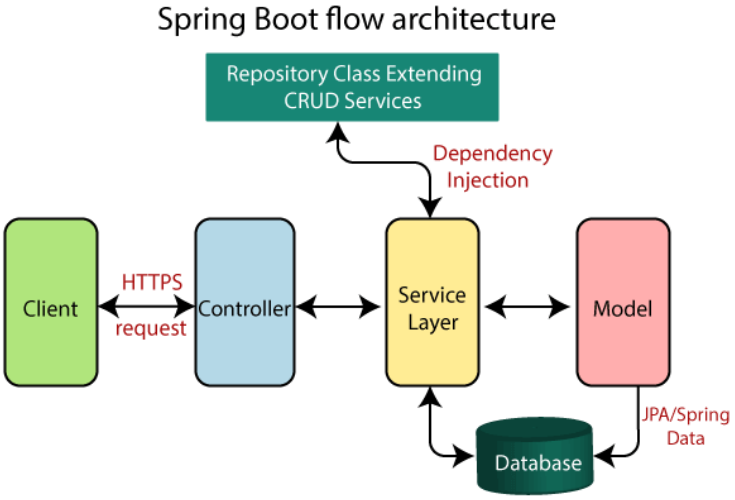


Рисунок 2.1 — Архітектура Spring Boot аплікацій[20]

Spring Boot складається з декількох частин: найбільш популярного Java веб-фреймворку — Spring Framework, влаштованого HTTP сервера — Tomcat, та базової конфігурації(рис. 2.2).

Рисунок 2.2 — Складові Spring Boot аплікацій[21]

2.1.2.2 Структура Angular аплікації

Більшість Angular аплікації мають таку структуру(рис. 2.3):

- Module — пакет який містить шаблони та компоненти.

- Template — шаблон, який зазвичай містить HTML елементи. Директиви забезпечують логіку програми, а розмітка прив'язки з'єднує ваші дані програми та DOM

- Component — певна одиниця UI, наприклад якась частина екрану. Зазвичай обробляє події, містить стан змінних шаблону

- Service — логіка програми або запити в мережу відбувається тут

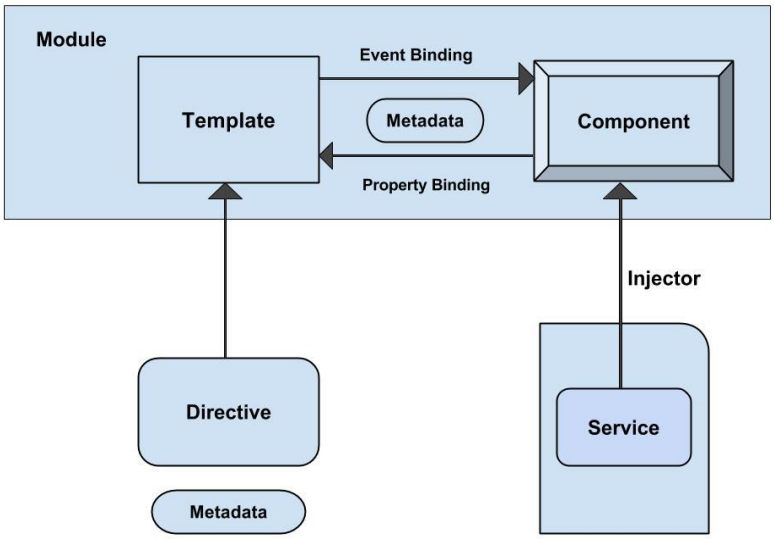


Рисунок 2.3 — Архітектура Angular аплікацій[22]

2.1.2.3 Файлова структура Maven проекту

Java проект, який використовує Maven(рис. 2.4), складається з наступних частин:

1) src: всі вихідні файли

- src/main: вихідні файли власне для проекту

- src/main/java: вихідний текст на Java

- src/main/resources: інші файли, які використовуються при компіляції або виконанні, наприклад properties-файли

- src/test: вихідні файли, необхідні для організації автоматичного тестування

- src/test/java: JUnit-тести для автоматичного тестування

- src/test/resources: файли-ресурси, які використовуються при тестуванні

2) target: всі створювані в процесі роботи Мавена файли

3) pom.xml: інформація для складання проекту, підтримуваного Apache Maven

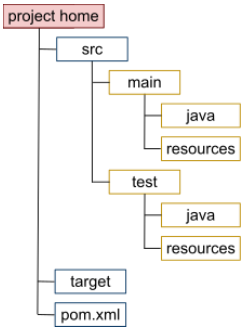


Рисунок 2.4 — Структура Maven проекту[23]

2.1.2.4 Файлова структура Angular проект

Звичайний Angular проект містить наступну файлову структуру(рис. 2.5):

1) src: це папка, яка містить основні файли коду, пов'язані з нашою angular програмою.

- src/app: містить файли, створені для компонентів додатку.

- src/app/app.component.css: містить код CSS у компоненті програми.

- src/app/app.component.html: містить HTML-файл, пов’язаний із його компонентом. Це файл-шаблон, який спеціально використовується для відображення даних.

- src/app/app.component.spec.ts: файл є тестовим, пов'язаним із компонентом. Цей файл використовується для тестів. Він запускається з angular CLI за допомогою командного рядка.

- src/app/app.component.ts: Це найважливіший файл, який включає логіку відображення компоненту.

- src/app/app.module.ts: файл, який складається з усіх залежностей веб-сайту. Дані використовуються для визначення необхідних модулів, які імпортуються, компонентів, які підлягають декларуванню та основного елемента для показу.

2) package.json: файл конфігурації npm. Він містить детальну інформацію про наш веб-сайт та залежність від пакетів.

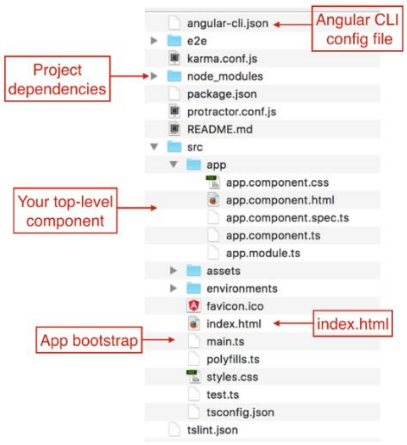


Рисунок 2.5 — Структура Angular проекту[24]

2.1.3 Досліджувані процеси. Їх основні характеристики

2.1.3.1 Процес розробки ПЗ

Більшість комерційних проектів пишуться в межах однієї або навіть декількох команд. Виходячи з цього, розробка ПЗ зводиться до кількох етапів, в яких задіяні багато осіб. Під час імплементації нового функціоналу, або правок старого, деякі функції можуть працювати неправильно, критично впливати на інший функціонал.

Сам процес розробки проходить в рамках певної методології або її видозміни, в межах даного проекту це був — Scrum. Scrum — підхід управління

проектами для гнучкої розробки програмного забезпечення[25]. Scrum чітко робить акцент на якісному контролі процесу розробки.

Ролі:

- Власник Продукту (Product Owner)
- Керівник (Scrum Master)
- Команда розробників (Scrum Team)

Зустрічі:

- Планування спринта
- Щоденна нарада
- Демонстрація
- Ретроспектива

Для трекінгу задач було обрано Trello, оскільки він є безкоштовним для середніх та великих команд та простим у використанні.

Trello — система управління проектами, яка надається безкоштовно(рис. 2.6). Вона використовується при керуванні проектами, використовуючи канбан дошку. Проекти зображені у вигляді дошки, що містять списки. Списки можуть містити картки, які містять задачі. Картка може мати відповідальних за неї користувачів. Картки можна перетягувати відповідно до статусу задачі.

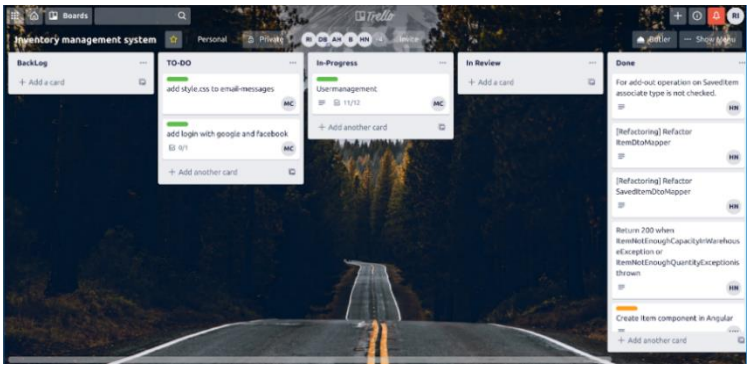


Рисунок 2.6 — Зовнішній вигляд Trello

Після того як задачі розібрані, команда розробників працює в межах декількох репозиторіїв Github, в межах однієї організації. Після того як розробник вносить якісь зміни він записує їх у віддалений репозиторій, тим

самим у кожного з команди є змога постійно мати доступ до останніх змін у коді.

2.1.3.2 Процес тестування аплікації

За ступенем автоматизації тестування буває:

- ручне тестування (manual testing);
- автоматизоване тестування (automated testing);
- напівавтоматизоване тестування (semiautomated testing).

На етапі старту роботи було задіяне тільки ручне тестування, яке займало час і, відповідно, людські ресурси. Ручне тестування повинно обов’язково бути, проте, було б краще, якщо б це відбувалося після того як відбувся етап автоматичного тестування. Для цього потрібно мати автоматизовані тести, бажано з високим покриттям коду.

2.1.3.3 Процес доставки коду до кінцевого користувача

Аплікація розгорталась шляхом виконання певного набору операції, які зводилися до наступного:

- завантаження останніх змін
- білд проекту локально, який призводив до створення артефакту
- локальний запуск, перевірка справності роботи
- перенесення артефакту на сервер
- заміна артефакту новою версією
- перенесення нового артефакту у відповідне місце
- видалення старого артефакту
- запуск jar файлу через screen

2.1.3.4 Загальний вигляд досліджуваних процесів

Процес створення, тестування, розгортання ПЗ можна зобразити наступним чином(рис. 2.7):

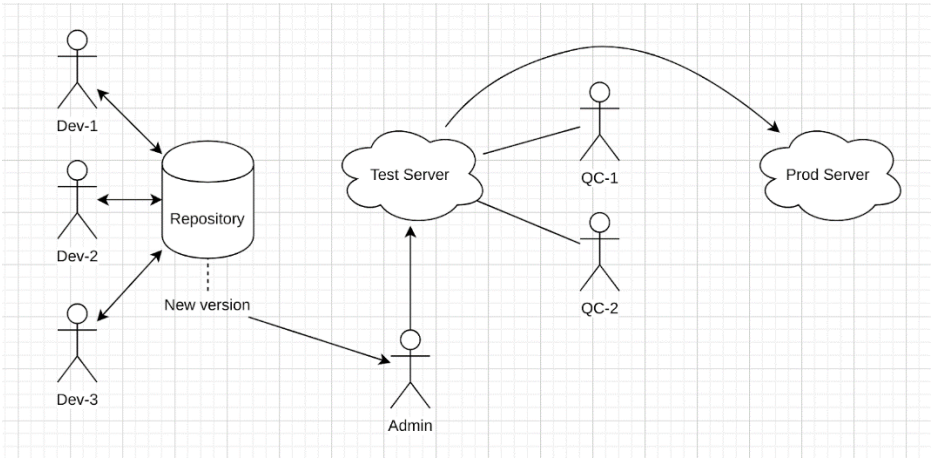


Рисунок 2.7 — CI / CD процеси до виконання автоматизації

Загалом процес розробки вимагає багато розробників, котрі в свою чергу взаємодіють з загальною кодовою базою, яка в певний період часу містить відповідну версію проекту. Ця нова версія може бути перенесена на сервер, тобто відбудеться процес розгортання нашої аплікації. Сервер для початку має бути тестовим, для відлагодження та перевірки програми. Коли нова версія програми потрапляє на тестовий сервер, в той момент відділ тестування випробовує, перевіряє на помилки даний продукт. Після успішної перевірки, якщо не було виявлено жодних дефектів, проект розгортається в межах production середовища, тобто такого, яке використовується реальними користувачами.

Як бачимо, якщо забрати з загальної картини людину яка займається адмініструванням, то виникне проблема, коли цей весь процес стає неможливим. Тобто ми маємо певні ризики щодо якихось непередбачених речей, які можуть статись через велику залежність від цієї однієї людини.

Шляхи вирішення:

- найняти ще одного робітника такого ж типу

- автоматизувати частково ці процеси, якими цей робітник займається

2.2 Вдосконалення наявних методів

2.2.1 Загальний вигляд після вдосконалення

Після вдосконалення наша спрощена система набуде наступного вигляду(рис. 2.8):

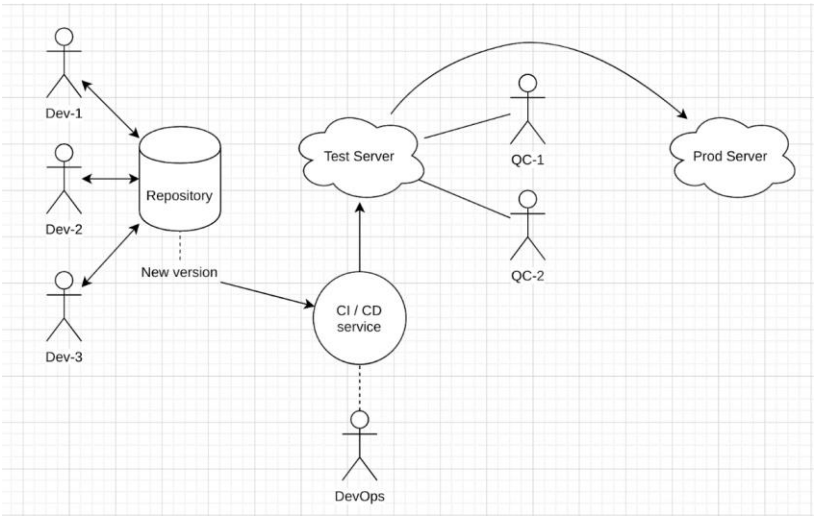


Рисунок 2.8 — CI / CD процеси після автоматизації

В даній ситуації DevOps відповідає тільки за налаштування системи автоматичної інтеграції та розгортання. Відповідно до даної ситуації, ми можемо не перейматись щодо можливості призупинки наших безперервних процесів, адже тепер вони забезпечуються за допомогою окремого сервісу.

2.2.2 Аналіз вимог та характеристик до розроблюваної системи

Система має відповідати наступним вимогам:

- Виконання тестів має проходити автоматично

- Доставка коду має відбуватись у автоматичному або напівавтоматичному режимі

- Система має бути стабільною

- Інфраструктура має відповідати такій, яка б могла легко масштабуватись при потребі

Характеристики розроблюваної системи:

- Першим проектом є бекенд частина, а другим — фронтенд частина клієнт-серверної аплікації. Бекенд взаємодіє з реляційною БД(рис. 2.9), віддаючи дані шляхом передачі через HTTP протокол базуючись на RESTful підході. СКБД — MySQL яка розміщена на AWS з використанням RDS.

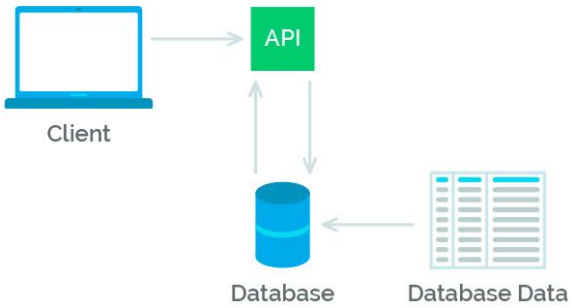


Рисунок 2.9 — Архітектура взаємодії Client – API – DB[26]

- Назва бекенд проекту — ims(скорочення від Inventory Management System), фронтенд — ims-ui. Взаємодія з користувачем(рис. 2.10) відбавається через його взаємодію з фронтенд частиною.

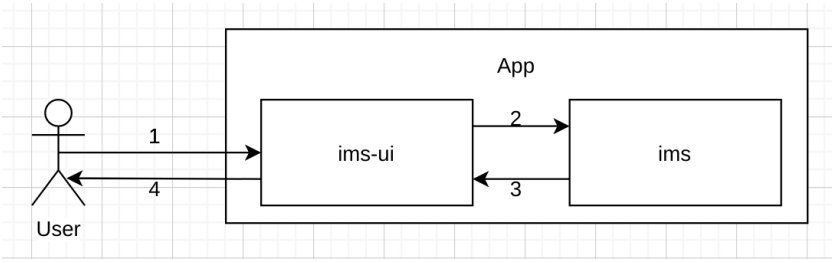


Рисунок 2.10 — Клієнт-серверна архітектура аплікацій ims, ims-ui

- Після тестування відбувається створення Docker образів, які вже можна використовувати при розгортання. Базовий Docker образ(рис. 2.11) для бекенд буде містити — Tomcat, а фронтенд — Nginx.

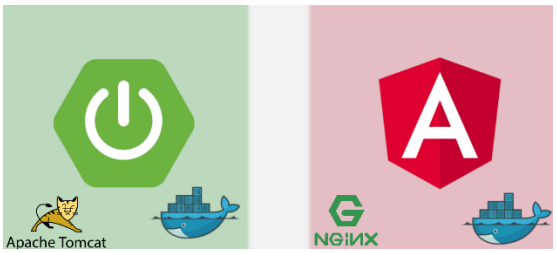


Рисунок 2.11 — Docker образи для проектів

- Складається з 2 проектів, де проведено автоматизацію прогонки тестів, створення артефакту та розгортання, у випадку якщо тести пройшли успішно(рис. 2.12)

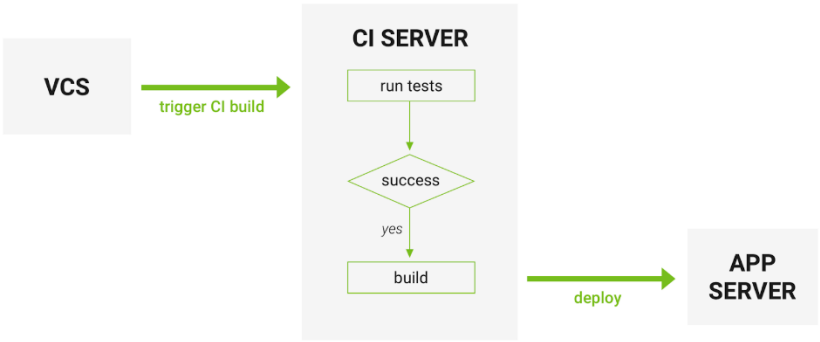


Рисунок 2.12 — Візуалізація роботи CI / CD системи[27]

2.3 Проектування створення VMs на GCP в межах проекту

Екземпляри віртуальних машин на платформі Google Cloud в межах одного проекту, який було створено, можна створити з створених образів, які будуть мати попередньо встановленні пакети для полегшення подальшої роботи.

В даному випадку було вирішено створити базовий образ(рис. 2.13) на базі Centos 7(docker-zsh), який буде включати в себе базове налаштування Linux системи, docker, docker-compose, zsh.

Від цього образу, в хмарі від Google Platform, буде створено 2 наступні екземпляри VM:

- ims-staging: місце для розгортання наших 2 аплікації, саме через його ip-адресу ми будемо мати доступ до програм, які збиралися розгорнути

- jenkins-ci: місце для розгортання нашої CI/CD-системи, тут буде зберігатись історія збірок, інформація про них, конфігурації для розгортання

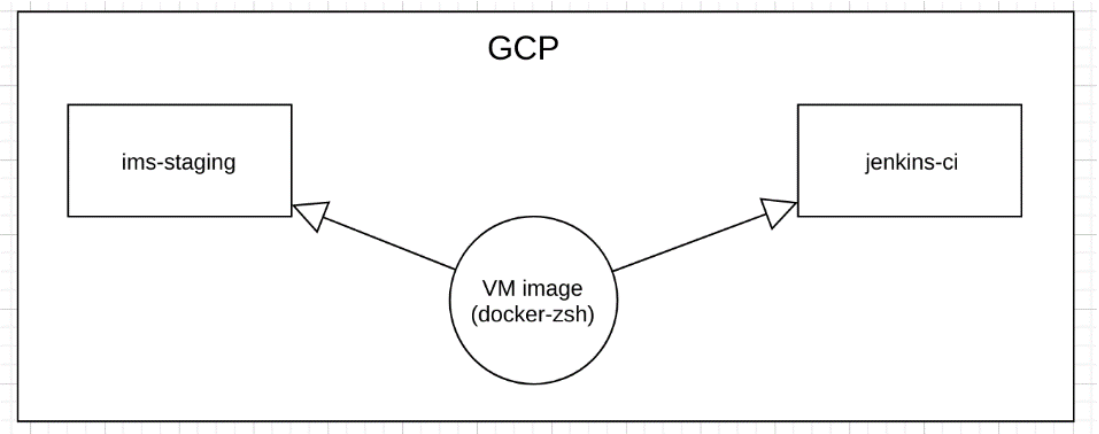


Рисунок 2.13 — Загальний GCP VM образ

2.3.1 Внутрішня структура VMs

Мінімальним набором для виконання CI / CD flow та зберігання нашої аплікації необхідно мати хоча б 2 екземпляри віртуальних машин(рис. 2.14), які ми орендуємо у Google Cloud Platform.

- ims-staging складається з Docker системи з декількома запущеними контейнерами: ims — бекенд проект, ims-ui — фронтенд проект

- jenkins-ci складається з: Tomcat з розгорнутим Jenkins CI всередині

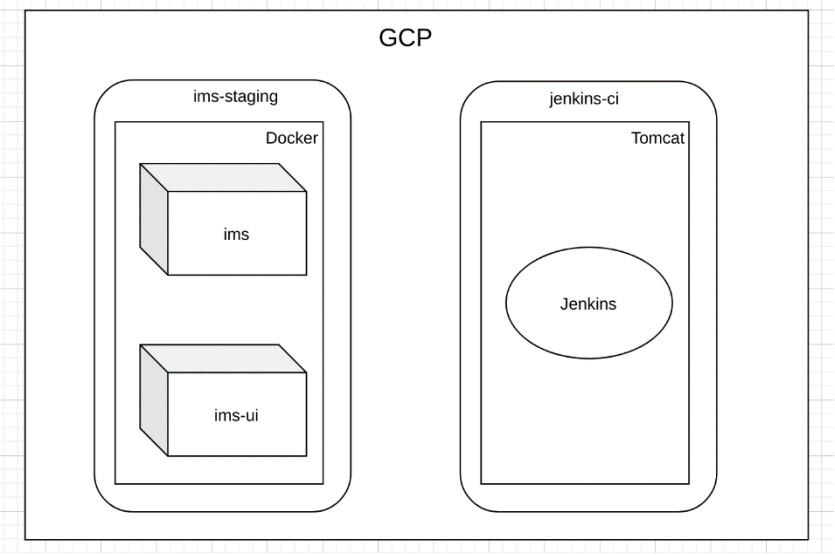


Рисунок 2.14 — Структура GCP VM образів

2.3.2 Сутність ims-stating instance

Екземпляр віртуальної машини ims-staging, з встановленим Docker-ом всередині, може мати певну кількість контейнерів всередині, які можуть бути запущеними на певному порті. На першому етапі у нас може бути тільки 2 контейнери — для бекенд та фронтенд аплікацій. Потенційно 4, тобто 2 бекенд та 2 фронтенд контейнера(рис. 2.15), які будуть запущені на різних портах.

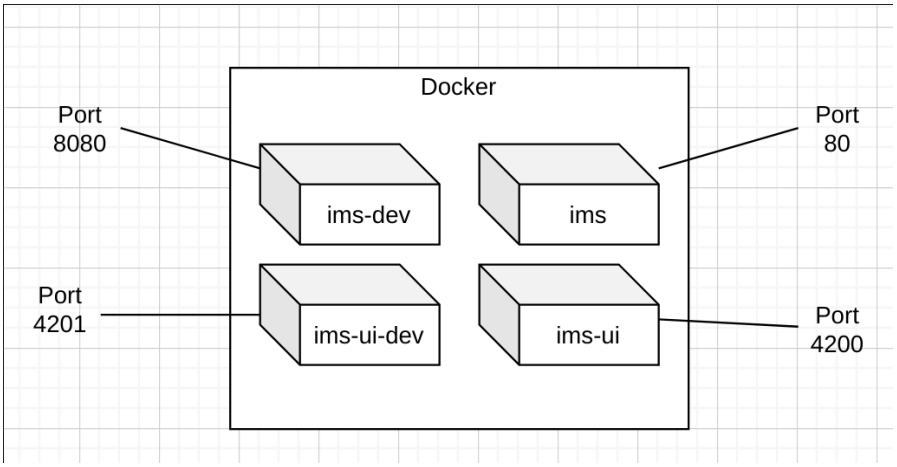


Рисунок 2.15 — Аплікації всередині Docker системи

Потенційно в даній схемі існують користувачі, які будуть бачити ims-ui у своєму браузері, а також інші клієнти, які будуть взаємодіяти з ims контейнером.

Взаємодія у обох випадках буде відбуватись через HTTP-протокол, шляхом використання запитів(рис. 2.16).

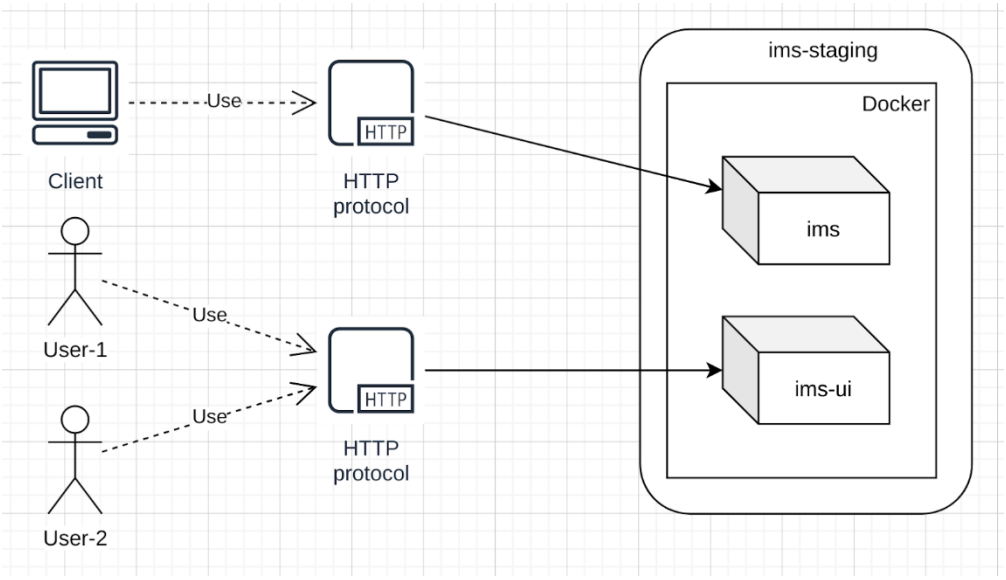


Рисунок 2.16 —Доступ через HTTP до проектів в Docker

2.3.3 Сутність ci-jenkins instance

Екземпляр віртуальної машини jenkins-ci, з встановленим Tomcat всередині, може мати декілька сервлетних аплікацій всередині. Таким чином, систему Jenkins CI буде розгорнуто через Tomcat(рис. 2.17) під ендпоінтом /jenkins.

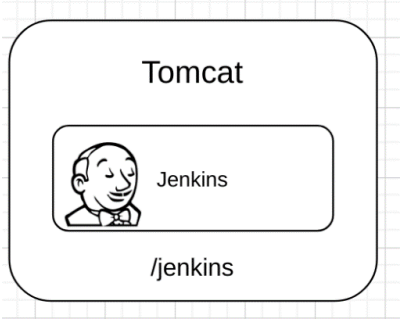


Рисунок 2.17 — Jenkins endpoint всередині Tomcat

При внесенні якихось змін в репозиторій когось з розробників, Github, після певних налаштувань, буде відсилати Webhook, який являє собою звичайний HTTP POST запит з певним корисним навантаженням(рис. 2.18), що і використовує Jenkins CI для початку виконання білда.

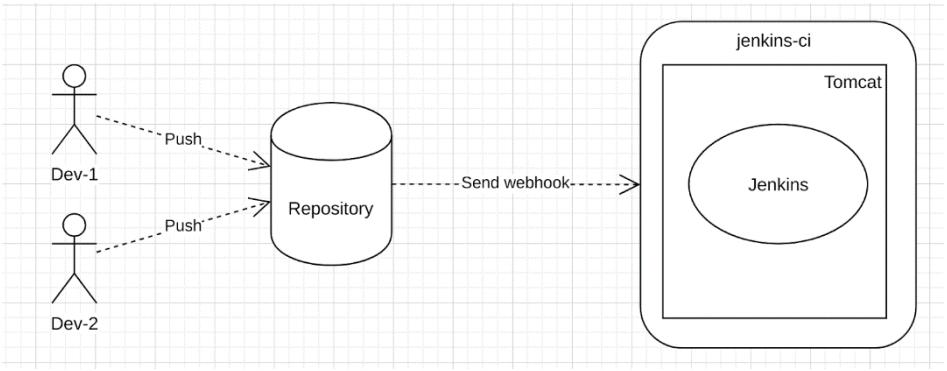


Рисунок 2.18 — Запуск Jenkins збірки через Webhook

Після отримання даних з Webhook-у(рис. 2.19), CI/CD система здатна виконувати її роботу у вигляді збірки проекту та його розгортання.

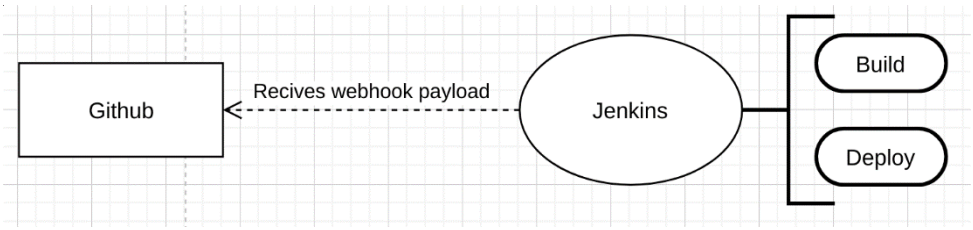


Рисунок 2.19 — Активація збірки Jenkins через Webhook

Збірка проекту включає в себе: компіляцію програмного коду, запуск тестів, створення та збереження артефактів. Розгортання проекту включає в себе: завантаження нової версії та її запуск.

2.4 Сутність Github Actions service

Github Actions — це сервіс, який надається Github як готове CI/CD рішення. Цей сервіс надає ряд готових інструментів, які можна комбінувати для створення та налаштування робочого процесу. В даному випадку цей

сервіс(рис. 2.20) виконує тільки перевірку справності фронтенд аплікації, створення її образу та його відправку в registry.

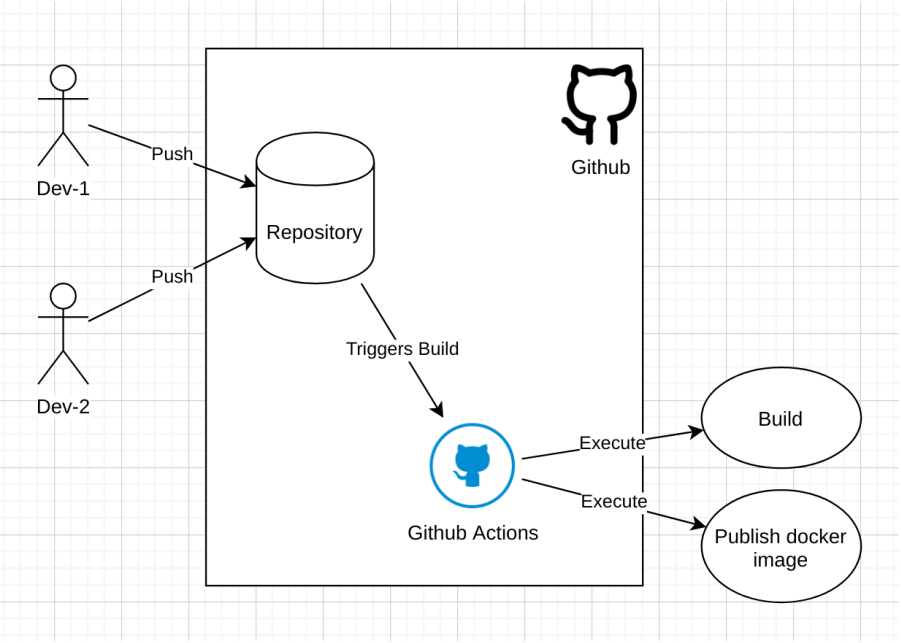


Рисунок 2.20 — Запуск та активація збірки через внутрішні механізми Github

2.5 Загальний Git workflow

Git Workflow — це рекомендація щодо використання Git для виконання роботи послідовно та продуктивно(рис. 2.21). Git workflow спонукає розробників ефективно та послідовно використовувати Git. Базова домовленість полягає у тому, що існують:

- master гілка
- dev гілка
- feature’s гілки

Розробники працюють в межах своєї feature гілки, синхронізуючись з dev гілкою. Після потрапляння в dev, feature гілки видаляються .Час від часу, коли проект на dev гілці стає стабільним, dev синхронізується з master гілкою і ми отримуємо наступну робочу версію продукту.

При цьому вважається, що гілки master, dev є готовими до розгортання. Всі останні зміни містяться в dev, тому гілка може бути нестабільною, на відміну від master.

Рисунок 2.21 — Git Workflow

2.6 Загальна CI/CD блок-схема

Початок CI процесу — запуск складання аплікацій, який виконує всі необхідні етапи для перевірки нашої програми на справність та, можливо, створення необхідних артефактів. У випадку, якщо гілка є стабільною та придатною до розгортання, ми створюємо докер образ та розгортаємо його на сервері.

Блок-схему зображено нижче на рис. 2.22:

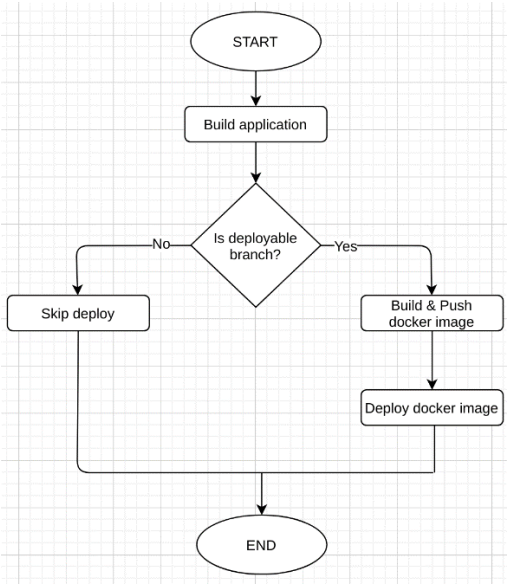


Рисунок 2.22 — Блок-схема CI / CD Pipeline

2.7 Continuous Integration

CI процес backend проекту на Spring Boot виконується за допомогою Jenkins CI(рис. 2.23), тож всі дії в межах виконання збірки описуються у файлі Jenkinsfile, який міститься в корені проекту, мають декларативний сценарій, написаний мовою програмування Groovy. Кожна стадія має певний результат після виконання кожного етапу.

- Білд програми — за необхідності скачує всі необхідні залежності, виконує тести, створює jar файл.

- Побудова docker образу — локально створює docker image за допомогою використання Dockerfile та його інструкцій.

- Відправка docker образу — виконує завантаження образу у віддалене сховище Dockerhub для його подальшого розгортання.

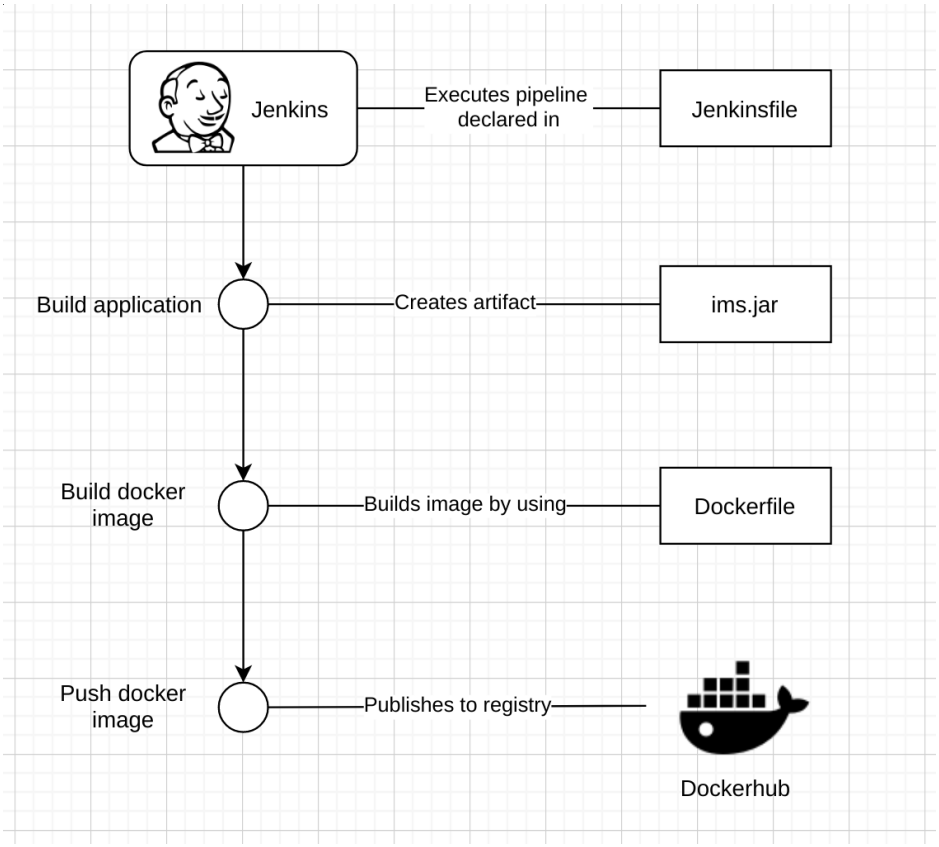


Рисунок 2.23 — Представлення Jenkins CI Pipeline

CI процес frontend проекту на Angular виконується за допомогою Github Actions(рис. 2.24), таким чином, етапи виконання збірки описуються у файлах ci.yml, cd.yml. За допомогою різного роду готових actions, виконуються більшість етапів CI, а також певна частина, яка впливає на кінцевий CD процес. Запуск 2 пайплайнів проходить паралельно, так як їхні сценарії не залежать один від одного.

Сценарій ci.yml:

- Встановлення пакетів для налаштування робочого середовища
- Запуск збірки аплікації

Сценарій cd.yml:

- Виконання створення нового Docker образу
- Доставка Docker образу у відповідний Docker registry(Dockerhub)
- Github автоматично виконує сценарії описані в .github/workflows папці, яка лежить в корені проекту.

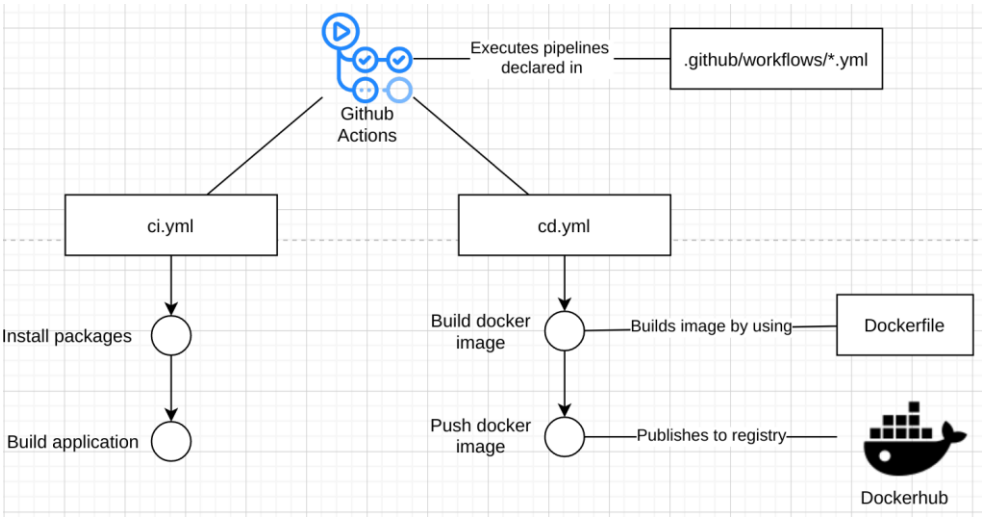


Рисунок 2.24 — Представлення Github Actions Pipeline

2.8 Continuous Delivery

Jenkins CI система, розгорнута в одному з GCP VM екземплярів(ci-jenkins), взаємодіючи з іншим VM екземпляром(ims-staging), займається

оперуванням процесу розгортання бекенд аплікацій(рис. 2.25), виконує наступні операції:

- Створення Docker образу
- Завантаження Docker образу у сховище для його зберігання
- Встановлення зв’язку з веб-сервером, де буде відбуватись розгортання
- Контроль процесу розгортання
- Стягнення останнього образу бекенд програми
- Оновлення існуючого контейнеру або створення нового, тим самим оновлення версії аплікації, яка перебуває на сервері

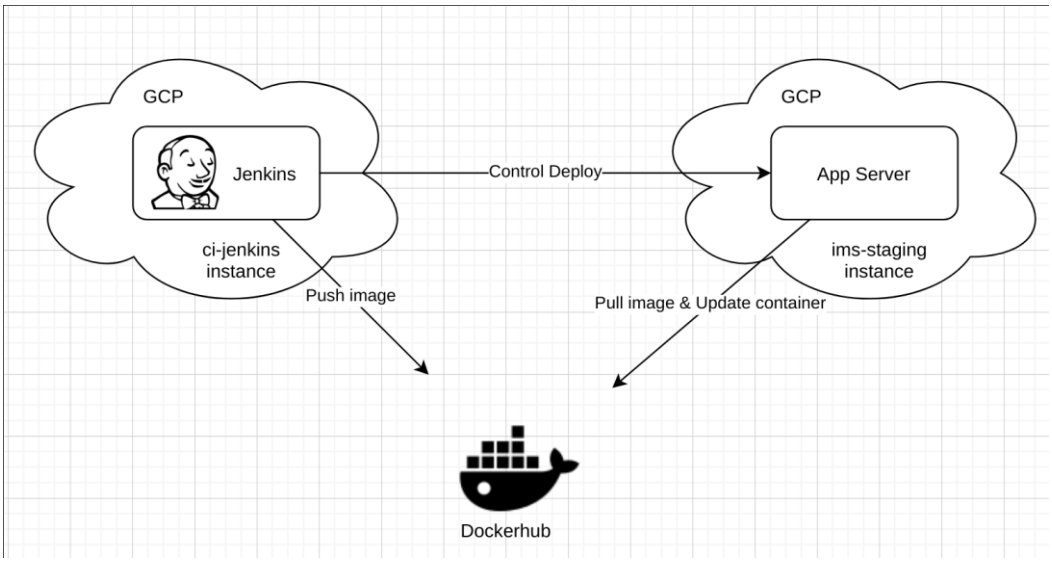


Рисунок 2.25 — Представлення backend Pipeline

Github Actions система, яка надається самим Github, не потребує виділеного сервера, а є доступною в ролі сервісу. Вона тільки створює новий образ фронтенд аплікації та доставляє його на Dockerhub. В свою чергу Dockerhub дозволяє встановити вебхук, що буде пов’язаним з Jenkins CI через Generic Webhook Trigger Plugin, як системою, яка може виконувати розгортання(рис. 2.26). Розгортання фронтенд аплікації нічим не відрізняється від бекенд, оскільки ми оперуємо контейнерами.

Дії кожної з підсистем можна описати наступним чином:

- Github Actions:
- Створення Docker образу

- Відправка Docker образу в Dockerhub

- Dockerhub

- Збереження Docker image

- Відправка webhook actions до Jenkins CI

- Jenkins CI

- Прийняття та аналіз webhook payload

- Стягнення останнього образу бекенд програми, створення чи оновлення контейнеру новою версією аплікації

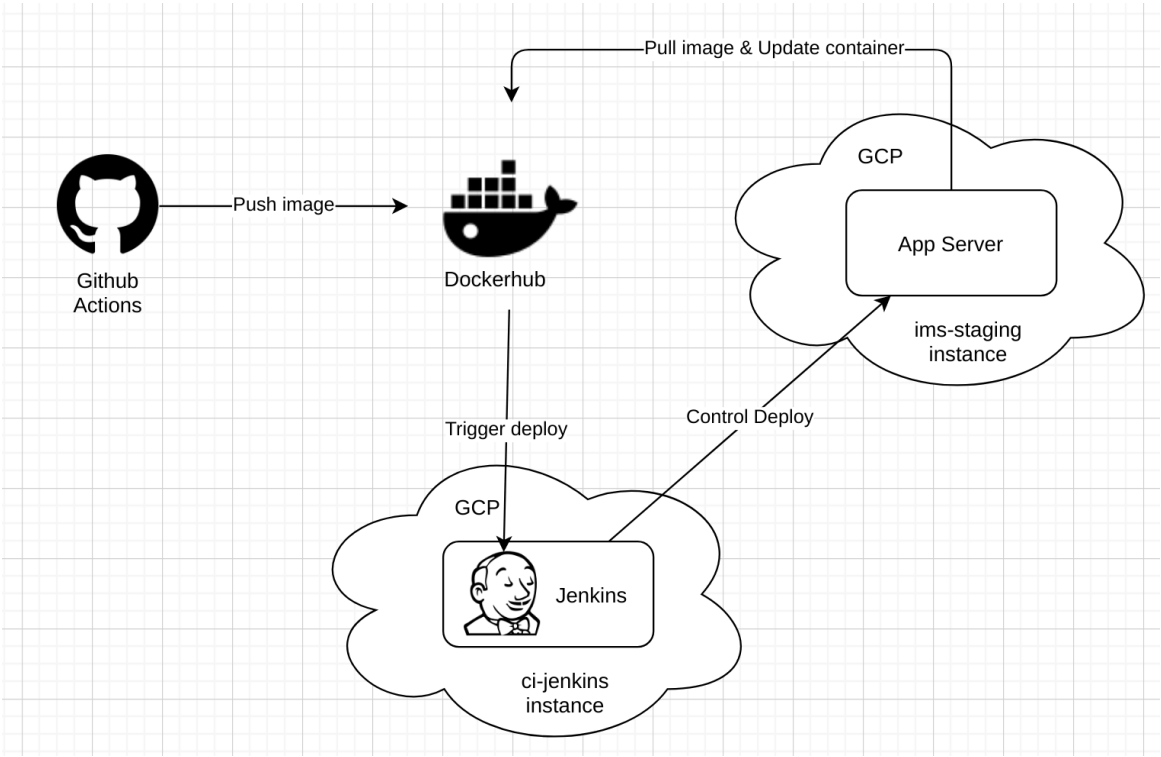


Рисунок 2.26 — Представлення frontend Pipeline

Jenkins CI може взаємодіяти з веб-сервером, де розміщено всі наші контейнери. Це можливо за допомогою Ssh-Agent plugin(рис. 2.27).

Ssh-Agent — це програма, яка відстежує ідентифікаційні ключі користувача та їх парольні фрази. Потім агент може використовувати ці ключі для входу на інші сервери, не вводячи знову користувача або пароль.

Для того щоб оновити версії наших програм нам потрібно: мати доступ до сервера, виконати відповідні команди, звертаючись до docker cli.

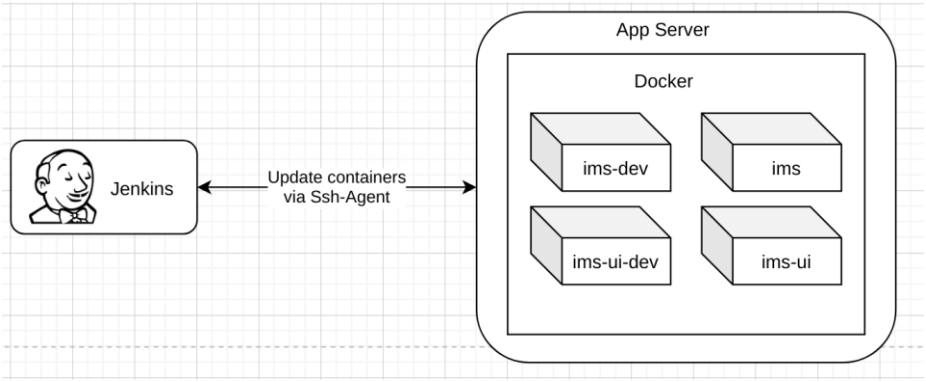


Рисунок 2.27 — Jenkins як загальна частина для розгортання

2.9 Проектування взаємодії backend з БД

Переважна більшість сучасних веб-додатків використовує бази даних для зберігання інформації. Так як клауд БД — це RDS, який надається AWS, в свою чергу, наші екземпляри віртуальних машин знаходяться в межах GCP, будуть взаємодіяти між собою(рис. 2.28).

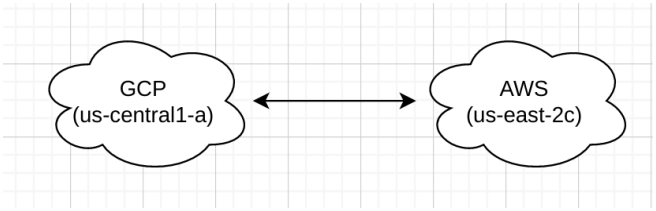


Рисунок 2.28 — Взаємодія AWS та GCP сервісів

Додаток може обмінюватися інформацією з БД, використовуючи з'єднання з нею. Якщо створювати при кожному зверненні до БД, ми можемо отримати програш в часі, бо виконання транзакції може зайняти всього кілька мілісекунд, в той час як на створення нового з'єднання може піти до кількох секунд. З іншого боку, можна створити одне з'єднання і звертатися до бази даних тільки через нього. Але це рішення може спричинити проблеми, в разі високого навантаження, наприклад: якщо одночасно сто користувачів спробує отримати доступ до бази даних використовуючи одне з'єднання, тоді

утвориться черга, що також згубно може позначитись на продуктивності додатка.

Database Connection Pool — це спосіб вирішення викладеної вище проблеми. У нашому розпорядженні є деякий набір(пул) з'єднань до бази даних. Коли новий користувач запитує доступ до БД, йому видається вже відкрите з'єднання з цього пулу. Якщо все відкриті з'єднання вже зайняті, створюється нове. Як тільки користувач звільняє одне з уже існуючих з'єднань, воно стає доступно для інших користувачів. Якщо з'єднання довго не використовується, воно закривається. RDS взаємодіє із backend аплікаціями(рис. 2.29) в Docker, створюючи пул з’єднань.

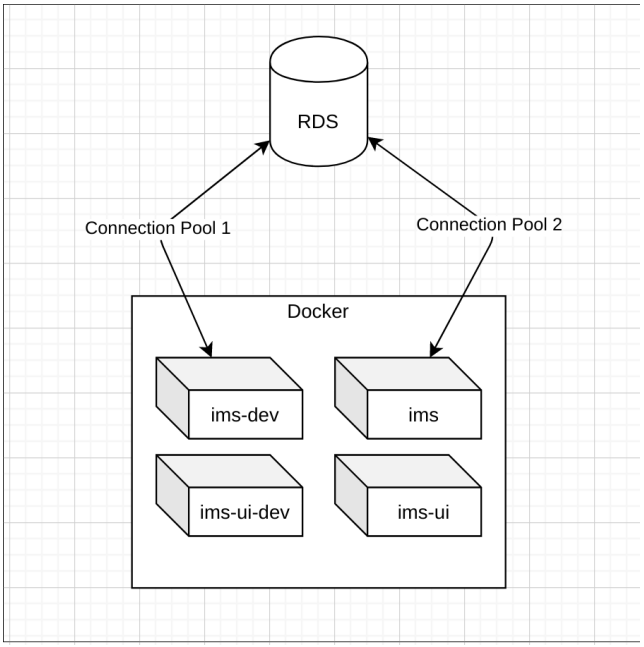


Рисунок 2.29 — Взаємодія RDS із backend аплікаціями

В даній ситуації, було 2 аплікації які взаємодіяли з БД. Перша перебуваючи всередині контейнера, є нестабільною, тобто dev-середовищем, що використовується для таких потреб як тестування та попереднього огляду. Друга — готова до використання для користувачів, здатна опрацьовувати велику кількість запитів. При цьому доступ до RDS відбувається для різних додатків під різними користувачами та до різних баз, що дозволяє запобігти проблемі цілісності даних.

3 РЕАЛІЗАЦІЯ

3.1 Перенесення проектів(бекенд та фронтенд) в Github

3.1.1 Створення організації

Організація — це сукупність облікових записів користувачів, які володіють сховищами. Організації мають одного або декількох власників, які мають адміністративні привілеї в межах організації.

Ми можемо створити нову організацію або перетворити існуючий особистий обліковий запис в організацію. В даному випадку ми створимо новий.

1. Відкривши Github, можна перейти до Settings(рис. 3.1)
2. Перейти до вкладки Organizations(рис. 3.2)
3. У розділі Organizations вибрати New organization(рис. 3.3)
4. Внести назву організації(рис. 3.4)
5. Створити організацію

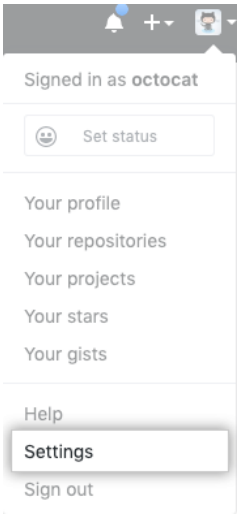


Рисунок 3.1 — Відкриття сторінки налаштувань

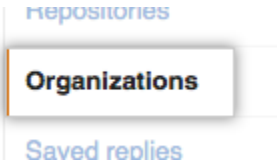


Рисунок 3.2 — Відкриття сторінки створення організації

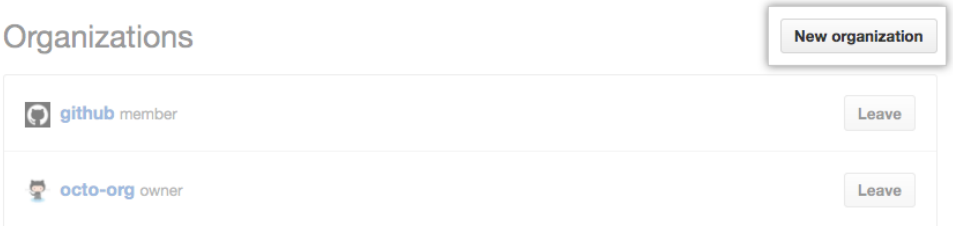


Рисунок 3.3 — Створення нової організації

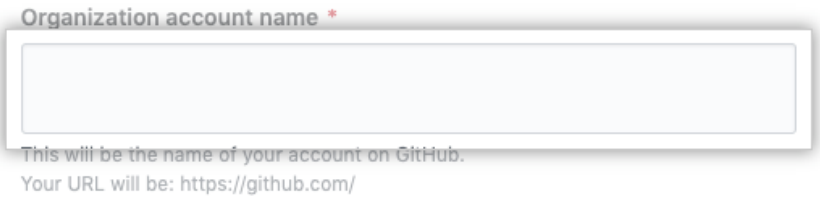


Рисунок 3.4 — Введення назви організації

3.1.2 Створення бекенд, фронтенд проектів в межах організації

Є можливість створити новий репозиторій в особистому обліковому записі або у будь-якій організації, де у вас є на це дозвіл.

1. У верхньому правому куті будь-якої сторінки потрібно скористатись drop-down menu та вибрати New repository(рис. 3.5)
2. Ввести назву бекенд проекту(рис. 3.6)
3. Ввести назву фронтенд проекту(рис. 3.7)
4. Перенести локальні версії проектів на Github(рис. 3.8, рис. 3.9)
    a. Перейти в директорію з проектом
    b. Добавити Remote URL
    c. Завантажити проект

В результаті, зовні Github Dashboard організації набув наступного вигляду(рис. 3.10)

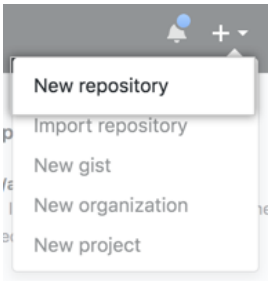


Рисунок 3.5 — Створення нового репозиторію

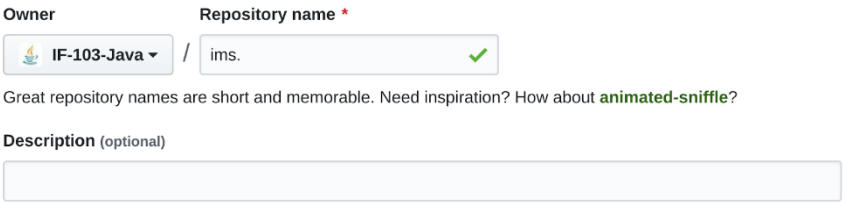


Рисунок 3.6 — Створення бекенд репозиторію

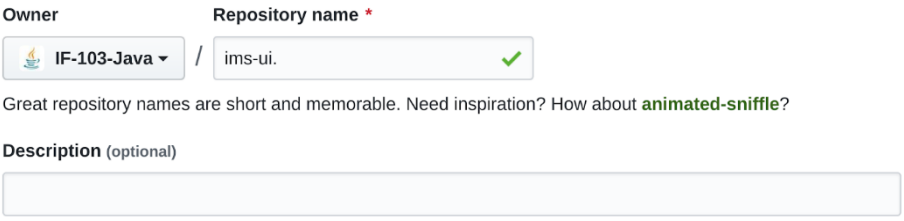


Рисунок 3.7 — Створення фронтенд репозиторію

```
$ cd ims
$ git remote add origin git@github.com:IF-103-Java/ims.git
$ git push -u origin master
```

Рисунок 3.8 — Завантаження коду бекенд репозиторію

```
$ cd ims-ui
$ git remote add origin git@github.com:IF-103-Java/ims-ui.git
$ git push -u origin master
```

Рисунок 3.9 — Завантаження коду фронтенд репозиторію

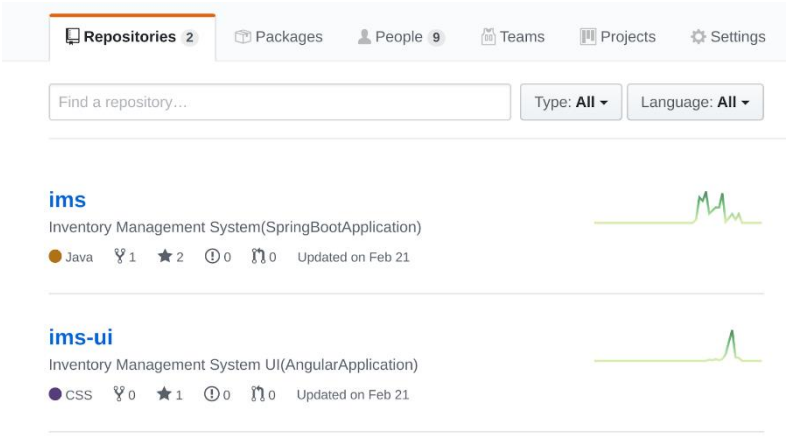


Рисунок 3.10 — Результат після створення проектів в межах організації

3.1.3 Налаштування правил для специфічних віток

Якщо ви є власником репозиторію або маєте права адміністратора, ви можете налаштувати захист гілок та застосувати певні правила перевірки коду перед його злиттям, такі як вимога більш ніж одного перегляду Pull Request-у або ж перевірки статусу, перш ніж дозволяти merge чи push.

Такі налаштування проводяться окремо для кожного з репозиторіїв. Для цього необхідно перейти в налаштування гілок та добавити деякі правила(рис. 3.11).

При додаванні правил кожна така гілка є захищена від force push, що дозволяє уникнути ситуації з небажаним та небезпечним для інших оновленням когось з учасників команди.

Гілки для яких це буде прийнято: master, dev(рис. 3.12). При цьому для master необхідна кількість людей, які підтвердили зміни — 3 людини, а для dev — 1 людина.

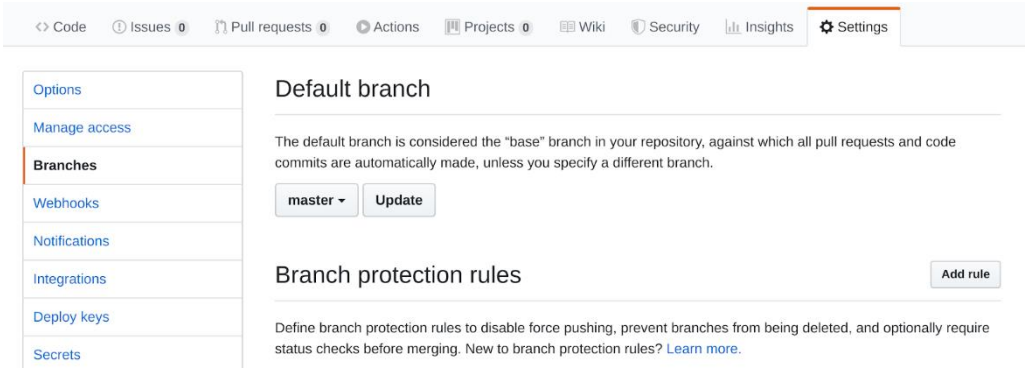


Рисунок 3.11 — Налаштування гілок Github

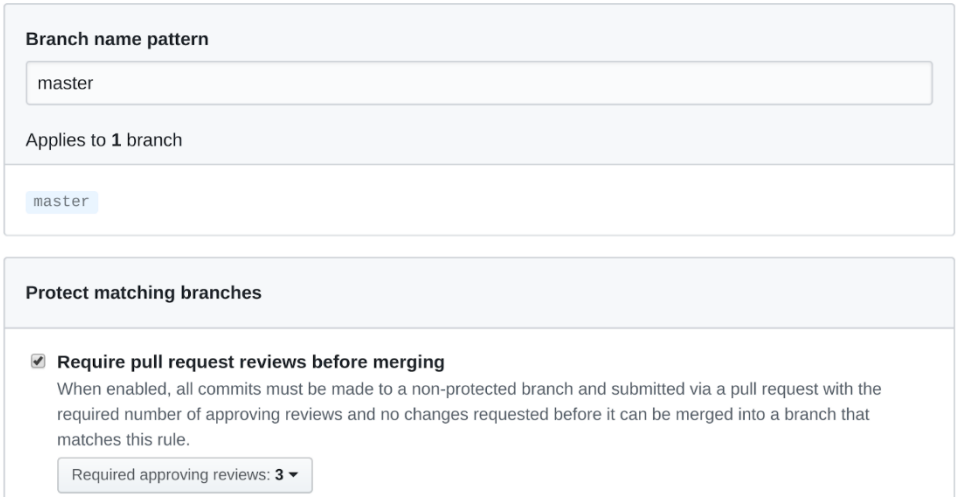


Рисунок 3.12 — Налаштування захисту гілок Github

3.1.4 Використання Code Review системи

У GitHub інструменти для перегляду коду вбудовані у кожен PR. Команда може створити відповідні правила, які покращують якість коду та вписуються у робочий процес. При цьому деякі зміни варто перевірити комусь з команди перед підтвердження певних змін. Нище буде приведено приклад стандартного workflow:

1. Створення нового PR, вимога змін одного з учасників команди(рис. 3.13)
2. Додаткові коментарі щодо коду(рис. 3.14)

3. Прийняття змін, їх злиття у dev гілку(рис. 3.15)

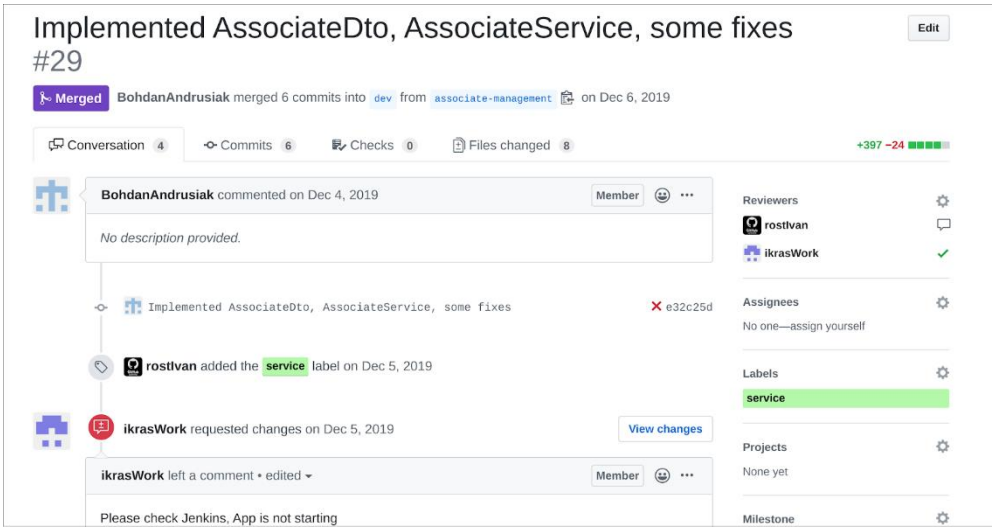


Рисунок 3.13 — Вигляд нового PR на Github

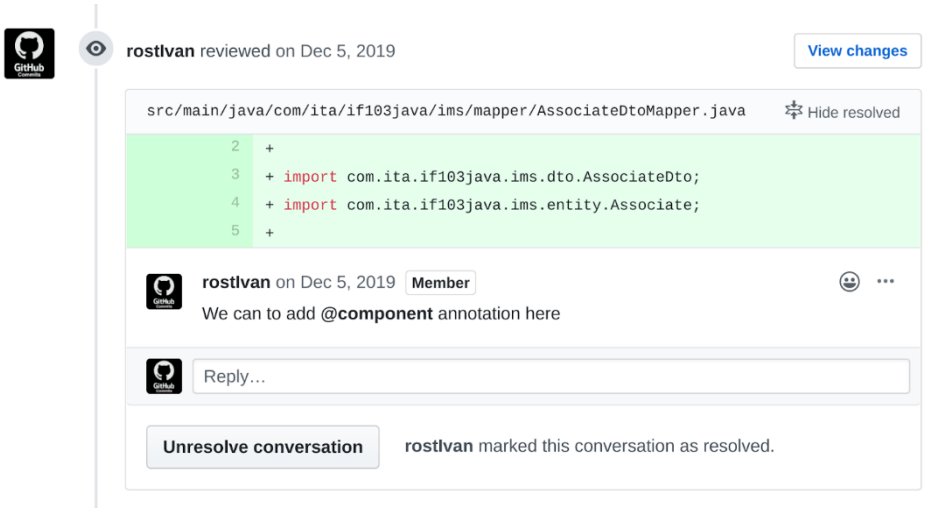


Рисунок 3.14 — Вигляд коментарів до PR на Github

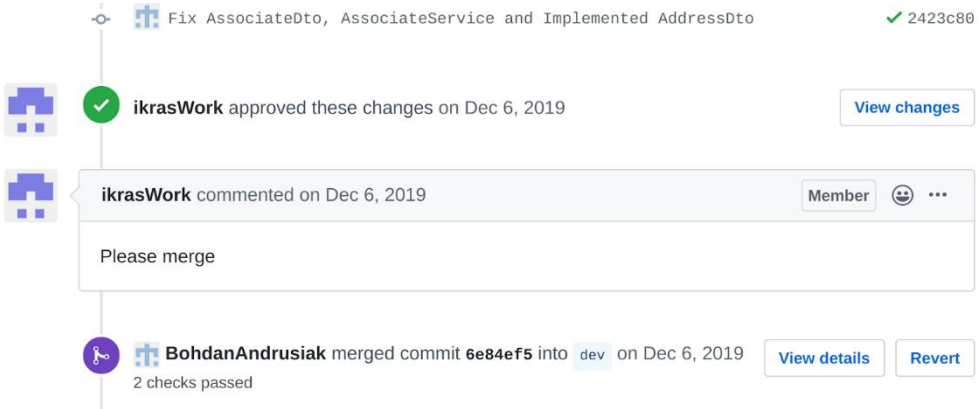


Рисунок 3.15 — Вигляд підтвердження змін до PR на Github

3.2 Створення 2-х VM(Centos 7) в Google Cloud Platform для подальших потреб

3.2.1 Створення загального образу на основі Centos 7 для подальшого створення нових VM екземплярів з нього

Для подальшого використання якогось базового образу для багатьох веб-серверів можна використати готовий механізм клауд провайдера — GCP, а саме — його можливість зберігати та створювати нові екземпляри віртуальних машин з використанням готових образів.

Для цього потрібно:

1. Відкрити Google Cloud Console, перейти на сторінку Compute Engine(рис. 3.16)
2. Вибрати секцію Images та натиснути на кнопку створення нового образу Create Image(рис. 3.17)

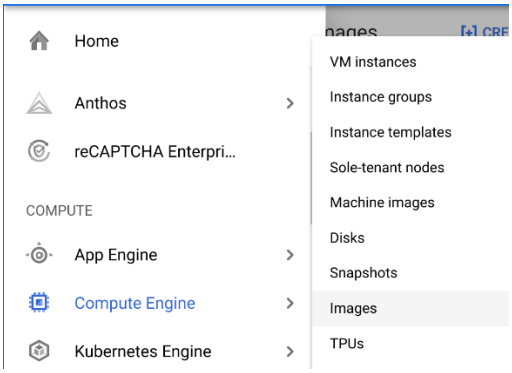


Рисунок 3.16 — Compute Engine в GCP

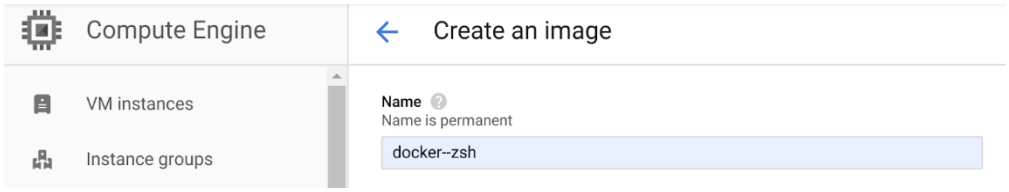


Рисунок 3.17 — Створення образу в GCP

3.2.2 Встановлення необхідних пакетів

Для роботи з Docker контейнерами нам потрібно встановити декілька пакетів, які необхідні для роботи з Docker Engine. Так як у нас основний образ створено на базі Centos 7, то для завантаження необхідних модулів ми будемо використовувати yum.

Yum — відкритий пакетний менеджер в Red Hat подібних системах. Був створений з метою полегшення процесу оновлення системи. Також, Yum робить пошук RPM-пакетів в своїх репозиторіях, їх встановлення, відстеження залежностей між ними, видалення таких пакетів які не використовуються, а також можливий даунгрейд (відкат версії пакету до попередньої).

Для встановлення Docker потрібно в межах консолі відповідного середовища базового екземпляру на базі Linux Centos:

1. Добавити відповідний репозиторій з необхідними пакетами(рис. 3.18). Встановити пакет yum-utils (який надає yum-config-manager) та налаштувати стабільну версію репозиторіїв для centos docker пакетів.
2. Встановити останню версію Docker Engine та containerd(рис. 3.19)
3. Запустити Docker(рис. 3.20)
4. Добавити docker групу(рис. 3.21)
5. Добавити користувача до групи docker для запуску контейнерів без sudo(рис. 3.22)

```
$ sudo yum install -y yum-utils

$ sudo yum-config-manager \
    --add-repo \
    https://download.docker.com/linux/centos/docker-ce.repo
```

Рисунок 3.18 — Добавлення репозиторіїв в Centos 7

```
$ sudo yum install docker-ce docker-ce-cli containerd.io
```

Рисунок 3.19 — Встановлення docker через yum

```
$ sudo systemctl start docker
```

Рисунок 3.20 —Запуск docker через systemctl

```
$ sudo usermod -aG docker $USER
```

Рисунок 3.21 — Добавлення групи з іменем docker

```
$ sudo groupadd docker
```

Рисунок 3.22 — Добавлення поточного користувача до docker групи

3.2.3 Створення інстансу ci-jenkins для хостингу Jenkins CI

Створення інстансу ci-jenkins виконується шляхом виконання наступних пунктів:

1. Відкрити Google Cloud Console, перейти на сторінку Compute Engine та вибрати вкладку VM instances(рис. 3.23)
2. Вибрати назву нового екземпляру — ci-jenkins(рис. 3.24)
3. Вибрати екземпляр типу N1 g1-small для загального використання(рис. 3.25)
4. Вибрати диск, який буде створено із загального образу, на якому, в свою чергу, вже є встановленими деякі необхідні пакети(рис. 3.26)

При виборі типу екземпляру було обрано саме N1 з 1 ядром ЦП та 1.7 ГБ оперативної пам’яті. Для запуску Jenkins та роботи в штатному режимі цього має вистачити при одночасному використанні для 2 проектів. При цьому використання менш потужного екземпляру не розглядається, оскільки, після його несправної роботи через нестачу ресурсів, його використання стало неможливим.

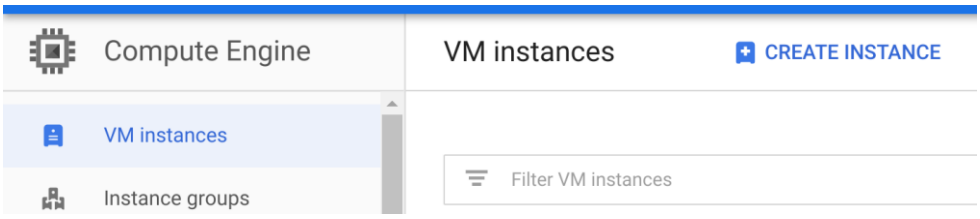


Рисунок 3.23 — Створення віртуальної машини на Google Cloud

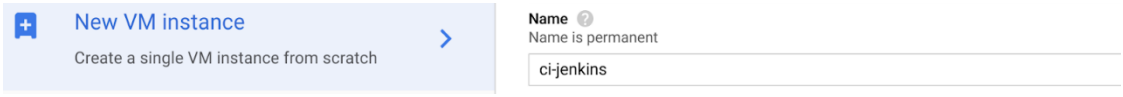


Рисунок 3.24 — Створення нового екземпляру ci-jenkins VM на GCP

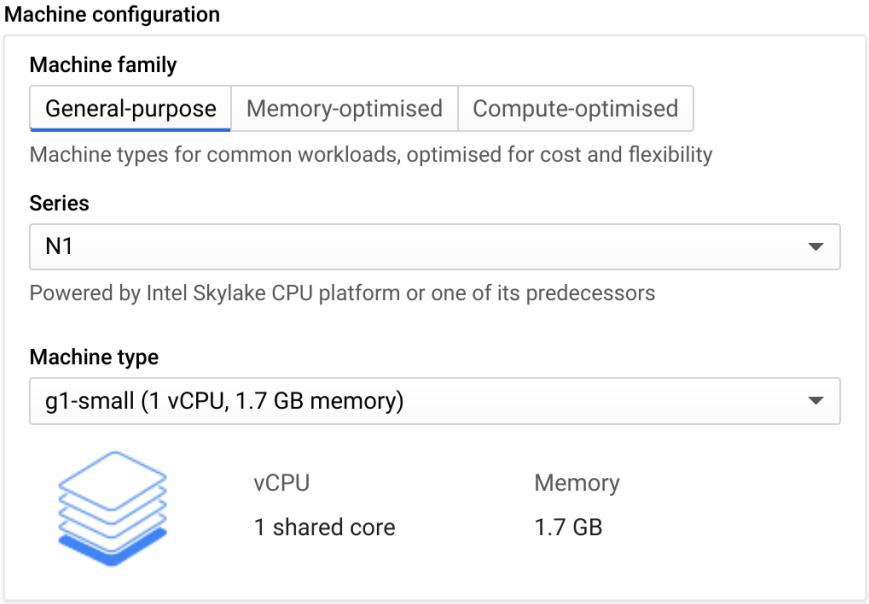


Рисунок 3.25 —Вибір типу g1-small віртуальної машини на GCP

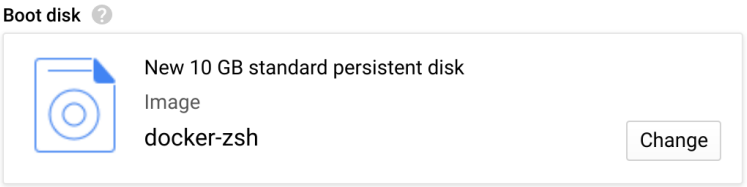


Рисунок 3.26 — Диск загального образу з встановленими пакетами

3.2.4 Створення інстансу ims-staging для розгортання аплікації

Етапи, які необхідні для створення:

1. Вибрати назву нового екземпляру — ims-staging. (рис. 3.27)
2. Вибрати екземпляр типу N1 n1-standard-1 для загального використання(рис. 3.28)
3. Вибрати наш базовий образ, який надасть необхідні пакети для старту наших аплікації(рис. 3.26)

При виборі типу екземпляру було обрано саме N1 з 1 ядром ЦП та 3.75 ГБ оперативної пам'яті. Для одночасного запуску 4 аплікацій, після тестування, виявилось можливим використовувати саме такий тип, який справляється з stage навантаженням без проблем.

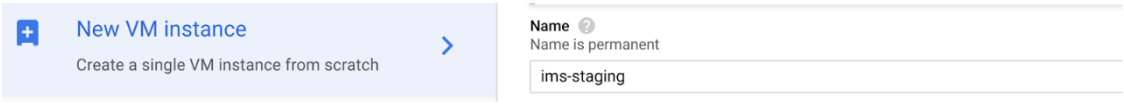


Рисунок 3.27 — Створення нового екземпляру ims-staging VM на GCP

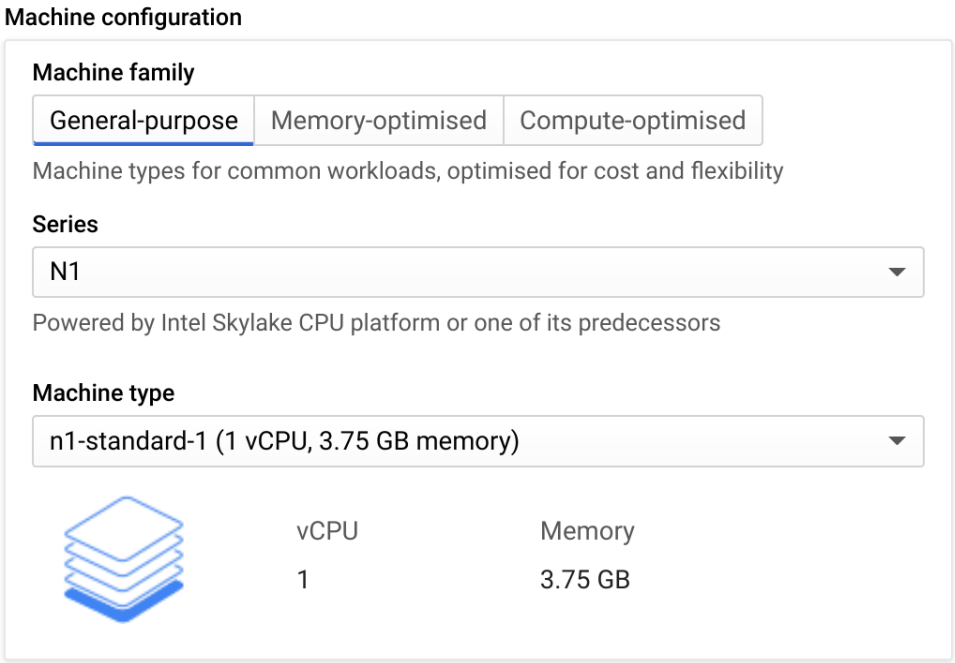


Рисунок 3.28 —Вибір n1-standard-1 типу віртуальної машини на GCP

3.3 Налаштування Jenkins CI для бекенд частини

3.3.1 Встановлення Tomcat на VM для подальшого розгортання Jenkins CI як Java Servlet Application

Для того щоб розгорнути Jenkins CI, варто спочатку правильно встановити Tomcat. Для справної роботи Tomcat Servlet Container необхідна Java, бажано останньої стабільної версії, або ж 8-мої, так як вона працює на більшості існуючих проектів.

Отже слід виконати наступні операції з консолі, попередньо підключившись до нашого ci-jenkins екземпляру:

1. Підключення до віддаленої консолі(рис. 3.29)
2. Встановити Java(рис. 3.30)
3. Завантажити та розпакувати Tomcat(рис. 3.31)
4. Перемістити у відповідну директорію, встановити змінну оточення $CATALINA_HOME(рис. 3.32)
5. Запустити Tomcat Server(рис. 3.33)
6. Переглянути чи Tomcat запущений(рис. 3.34)

```
$ ssh -i ~/.ssh/if103java-ci.pem username@ci-if103java.pp.ua
```

Рисунок 3.29 — З'єднання через SSH

```
$ sudo apt install openjdk-8-jdk
```

Рисунок 3.30 — Встановлення JDK

```
$ wget https://apache.ip-connect.vn.ua/tomcat/tomcat-9/v9.0.34/bin/apache-tomcat-9.0.34.tar.gz
$ tar xvfz apache-tomcat-9.0.0.M21.tar.gz
```

Рисунок 3.31 — Завантаження Tomcat

```
$ mkdir /opt/tomcat9 && mv apache-tomcat-9*/* /opt/tomcat9
$ echo "export CATALINA_HOME=/opt/tomcat9" >> ~/.zshrc
```

Рисунок 3.32 — Створення папки з Tomcat та змінної оточення

```
$ $CATALINA_HOME/bin/catalina.sh start
```

Рисунок 3.33 — Запуск Tomcat

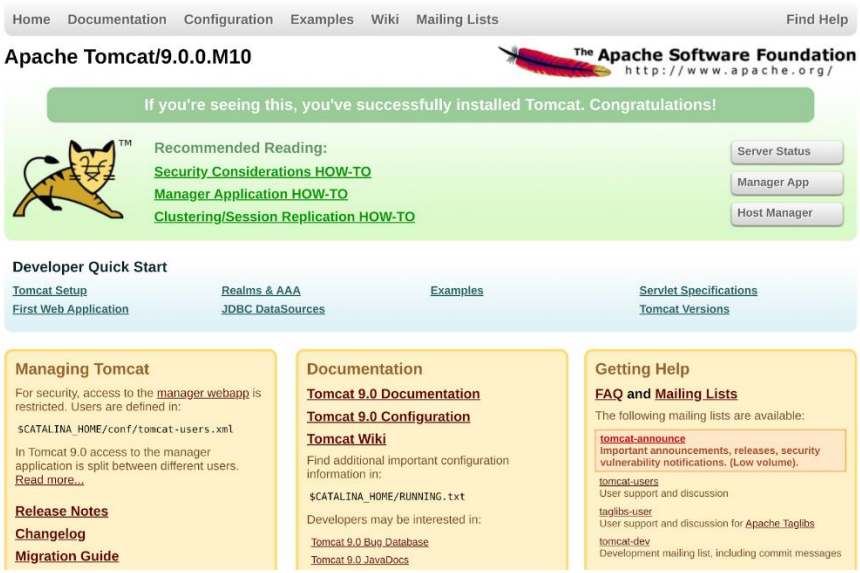


Рисунок 3.34 — Зовнішній вигляд Tomcat

3.3.2 Налаштування Tomcat для коректної роботи

Tomcat, зазвичай, запускається на порті 8080, який є для нього стандартним. Проте було би зручніше перенести його на порт 80, або ж налаштувати переадресацію з 80 на 8080.

Для цього може допомогти iptables. Порти можна перенаправити, якщо це потрібно. Ми можемо запускати сервіси Tomcat, які слухають порт 8080, але користувачі, які забули додати порт до URL-адреси, перейшли за портом 80, отримують перенаправлення на порт 8080.

Тож, для цього варто виконати переадресацію портів(рис. 3.35):

```
$ iptables -A INPUT -p tcp -dport 8080 -j ACCEPT
$ iptables -t nat -A PREROUTING -p tcp -dport 80 -j REDIRECT -to-ports 8080
```

Рисунок 3.35 — Переадресація портів з терміналу

Також існує інша проблема з таким підходом, коли ми стартуємо Tomcat процес, адже після перезавантаження сервера він перестане працювати. Щоб від автоматично повертався до робочого стану після перезавантаження можна зробити його службою за допомогою systemd — менеджера системи і служб для Linux. Для цього потрібно створити service файл(рис. 3.36). Приклад такого файлу буде зображено на рис. 3.37.

```
$ vim /etc/systemd/system/tomcat.service
```

Рисунок 3.36 — Відкриття консольного редактору vim

```
[Unit]
Description=Tomcat 9 servlet container
After=network.target

[Service]
Type=forking
Restart=always

User=username
Group=group

Environment="JAVA_HOME=/usr/lib/jvm/java-1.8.0-openjdk-amd64/jre"
Environment="CATALINA_HOME=/opt/tomcat9"

ExecStart=$CATALINA_HOME/bin/startup.sh
ExecStop=$CATALINA_HOME/bin/shutdown.sh

[Install]
WantedBy=multi-user.target
```

Рисунок 3.37 — Файл tomcat.service

Для створення найпростішого сервісного юніта треба описати три секції: Unit, Service, Install.

Опис змінних та їх значень:

- Description=Tomcat 9 servlet container — опис нашого юніта

- After=network.target — запускати юніт після сервісу network.target

- Type=forking — служба запускається 1 раз і процес розгалужується із завершенням батьківського процесу. Даний тип використовується для запуску класичних демонів.

- Restart=always — systemd автоматично перезапустить наш сервіс, якщо він раптом перестане працювати.

- User=username, Group=group — користувач та група, під яким стартує сервіс

- Environment="JAVA_HOME=...", Environment="CATALINA_HOME=..." — змінні оточення

- ExecStart=$CATALINA_HOME/bin/startup.sh — команда щодо запуску служби

- ExecStop=$CATALINA_HOME/bin/shutdown.sh — команда щодо зупинки служби

- WantedBy=multi-user.target — відповідає нашому звичному runlevel = 3, тобто це — багатокористувацький режим без графіки

Після створення файлу служби Linux, у нас є можливість нею скористатись, перезапустивши демон systemd(рис. 3.38).

```
$ sudo systemctl daemon-reload
$ sudo systemctl enable tomcat
$ sudo systemctl status tomcat
● tomcat.service - Tomcat 9 servlet container
   Loaded: loaded (/etc/systemd/system/tomcat.service; enabled; vendor preset: disabled)
   Active: active (running) since нд 2020-01-26 23:09:50 UTC; 2 months 18 days ago
 Main PID: 28673 (java)
```

Рисунок 3.38 — Статус Tomcat сервісу

Для доступу до інтерфейсу керування Tomcat веб-аплікаціями варто створити користувача, який буде мати ролі manager-gui та admin-gui.

Це необхідно вказати у файлі $CATALINA_HOME/conf/tomcat-users.xml(рис. 3.39).

```
<tomcat-users ...>
    <user username="admin" password="password" roles="manager-gui,admin-gui"/>
</tomcat-users>
```

Рисунок 3.39 — Файл tomcat-users.xml

За замовчуванням нові версії Tomcat не надають доступ до аплікації /host-manager та /manager за запитом, що надходять не з самого сервера. Щоб забрати

це обмеження на віддалені IP-адреси, потрібно змінити файл server.xml(рис. 3.40).

```
<Context antiResourceLocking="false" privileged="true" >
  <Valve className="org.apache.catalina.valves.RemoteAddrValve"
         allow="127\.\d+\.\d+\.\d+|::1|0:0:0:0:0:0:0:1"/>
</Context>
```

Рисунок 3.40 — Файл server.xml

Після цього стає доступною сторінка(рис. 3.41), де можна переглянути список запущених Java аплікацій, зупинити якісь з них, або ж їх перезавантажити.

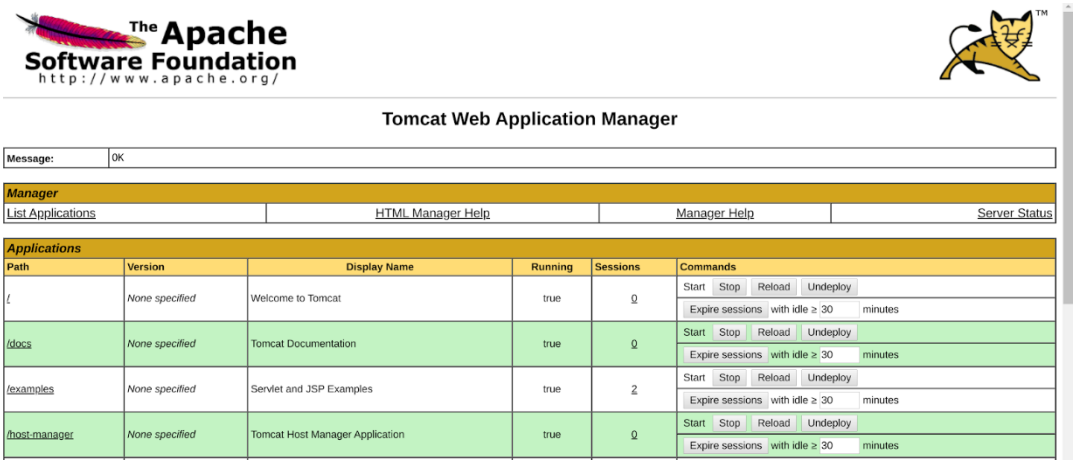


Рисунок 3.41 — Зовнішній вигляд Tomcat Web Application Manager

3.3.3 Розгортання Jenkins CI

У випадку розгортання будь-яких Java веб-аплікацій, шляхом використання готового war артефакту, зробити це можна 2-ма способами:

- перенесенням його до папки $CATALINA_HOME/webapps

- розгортанням з інтерфейсу вбудованої системи Tomcat Web Management.

Для цього існують наступні прості кроки:

1. Скачати останній стабільну версію war файлу, яку можна використовувати для розгортання(рис. 3.42)
2. Перейти до Tomcat Web Management, розгорнути завантажений артефакт(рис. 3.43)

```
$ wget -O /tmp/jenkins.war http://mirrors.jenkins.io/war-stable/latest/jenkins.war
```

Рисунок 3.42 — Завантаження останньої версії Jenkins WAR

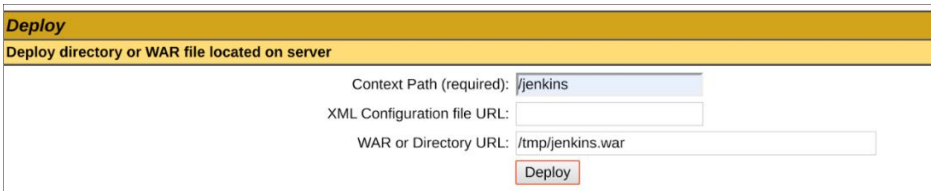


Рисунок 3.43 — Розгортання Jenkins в Tomcat

3.3.4 Зв'язка Jenkins з існуючим Github репозиторієм через webhook

При зв'язці Github репозиторію з нашим розгорнутим Jenkins, варто вказати його адресу та стандартний шлях для прийняття такого роду для запитів від Github — /github-webhook(рис. 3.44).

При виконанні якоїсь певної дії, наприклад: додавання змін у коді, запит на злиття коду, створення релізу — Github буде надсилати сповіщення для нашої системи Jenkins, яка буде розуміти який сценарій використовувати при цьому.

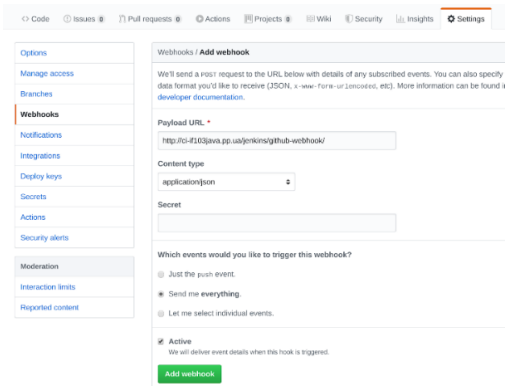


Рисунок 3.44 — Встановлення Jenkins Webhook для Github репозиторію

3.3.5 Створення проекту в CI системі, Multibranch Pipeline для нього

Jenkins CI підтримує можливість створення та збереження різної кількості проектів для різних мов та потреб. Тож, для створення нового проекту в межах системи Jenkins, можна перейти в головне меню та перейти до його створення(рис. 3.45).

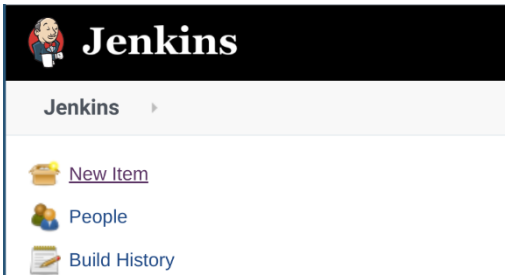


Рисунок 3.45 — Створення нового проекту в Jenkins

При виборі типу проекту, варто підходити до вибору, виходячи з ваших потреб. Проте, найбільш гнучким для інтеграції в більшість проектів є саме Multibranch Pipeline(рис. 3.46), який є абстракцією виконання певного роду дій, які описані в декларативному стилі, по-різному може відпрацювати з урахуванням гілок на яких він запускається.

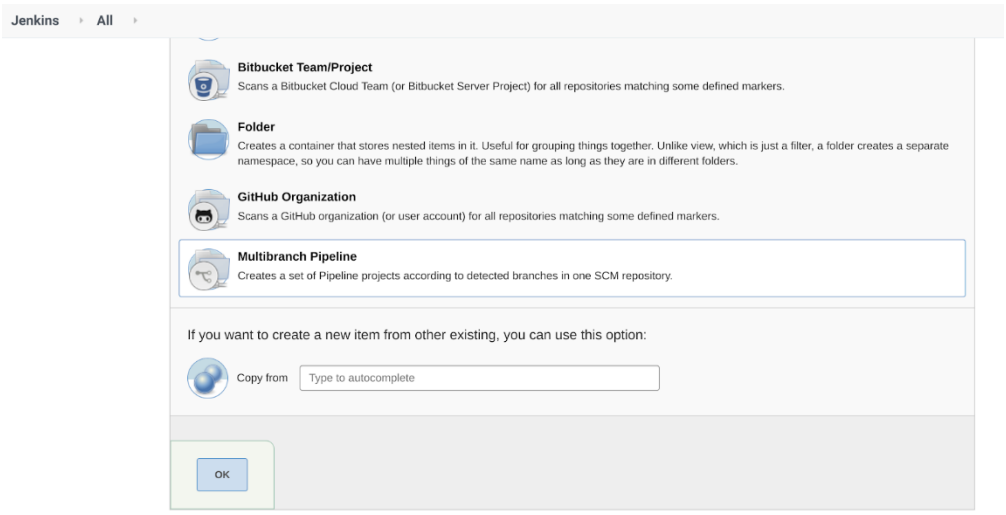


Рисунок 3.46 — Створення Multibranch Pipeline в Jenkins

У відповідних полях можна вписати URL до нашого репозиторію та стратегію обробки нових комітів(рис. 3.47). Також варто, в свою чергу, при використанні в межах організації, перевіряти на наявність Pull Request, проводити їх збірку.

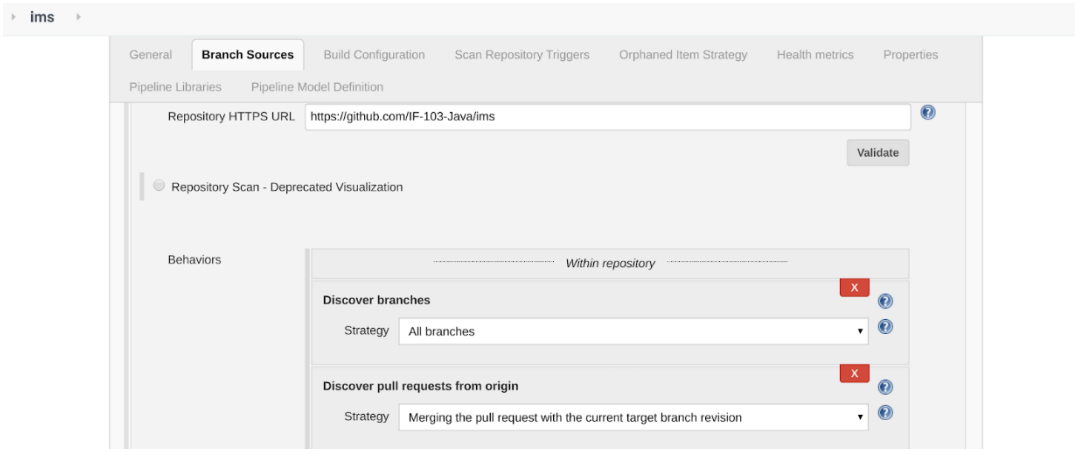


Рисунок 3.47 — Налаштування Multibranch Pipeline в Jenkins

Сам сценарій буде лежати прямо у файлі з назвою “Jenkinsfile”(рис. 3.48), який містить необхідні етапи та інструкції для виконання. Сканувати репозиторій з певною періодичністю немає сенсу, так як всі збірки будуть проходити шляхом прийняття вебхуку. Проте після збірок залишаються логи, файли різного роду, які в більшості випадків не приносять якоїсь суттєвої користі, особливо коли вони залишились з минулих старих збірок, тож їх можна автоматично чистити після певного часу.

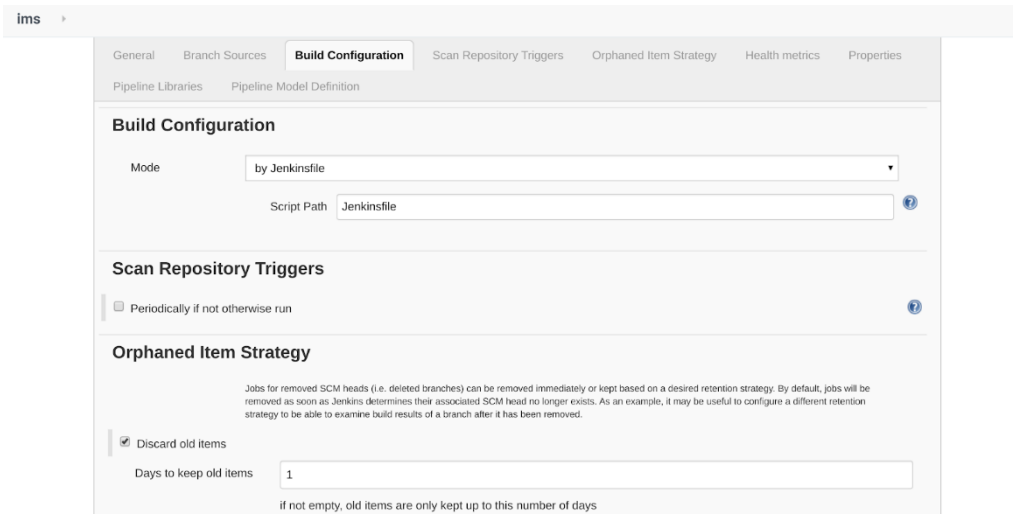


Рисунок 3.48 — Додаткова конфігурація Jenkins Pipeline

3.3.6 Збереження конфіденційної інформації

При використанні якихось даних для доступу до приватних ресурсів, необхідно використовувати конфіденційні дані, такі як логіни та паролі, секретні ключі та інші. У відкритому вигляді їх зберігати не варто. В результаті, Jenkins надає можливість зробити це безпечним шляхом.

Дженкінс може зберігати такі типи секретних даних:

- Secret text — звичайний текст

- Username and password — спеціальний тип для збереження логіну та паролю у вигляді username:password

- Secret file — звичайний файл з контентом всередині

- SSH Username with private key — це ssh ключ, який можна використовувати для входу на інший сервер

В межах даних проектів було добавлено наступні Credentials(рис. 3.49)

- ims-staging-user — користувач на веб-сервері

- ims-staging-host — адреса веб-сервера

- ims-staging-ssh-key — ключ до веб-сервера

- docker-hub-credentials — логін та пароль для використання акаунту Docker Hub

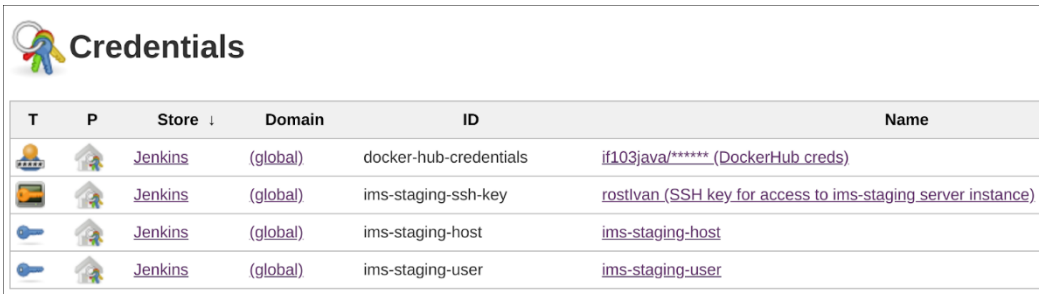


Рисунок 3.49 — Додані в Jenkins конфіденційні дані

3.3.7 Створення Jenkinsfile з сценарієм збирання проекту, створенням артефакту та його потенційним розгортанням

Jenkinsfile — це текстовий файл, який містить опис для Jenkins Pipeline. Зазвичай, declarative pipelines представляють у вигляді Jenkinsfile в корені сховища. Pipeline, в основному, містить етапи (stages) і кроки (steps). Загалом він має наступну структуру:

1) Збірка

- Клонування git сховища з кодом
- Налаштування змінних оточення
- Збірка docker
- Завантаження його в dockerhub

2) Деплой

- Підключення до віддаленого сервера
- Оновлення docker контейнеру на сервері

Нище буде розглянуто Jenkins pipeline для бекенд проекту(лістинг його коду наведено в додатку А). Кожен pipeline можна описати шляхом певного роду DSL мови, наприклад, це може бути groovy. Перша лінійка(рис. 3.50) “#!/usr/bin/env groovy” вказує шукати groovy для виконання сценарію. Так ми визначаємо, що це декларативний pipeline — “pipeline { ... }”. Агент призначений для виконання декларативних сценаріїв.

```
#!/usr/bin/env groovy

pipeline {
    agent any

    tools {
        jdk "JDK-13.0.1"
    }
```

Рисунок 3.50 — Початок Jenkins Pipeline

Для використання Maven та збірки Java Web аплікацій потрібно вказати JDK(рис. 3.51). Так як розробка велась на Java версії 13, то версії мають співпадати і у pipeline.

```
tools {
    jdk "JDK-13.0.1"
}
```

Рисунок 3.51 — Використання зовнішнього JDK в Jenkins

Протягом білда можна встановити відповідні змінні оточення в блоці “environment”(рис. 3.52).

```
environment {
    DOCKERHUB_REPO = "if103java/ims"
    IMAGE_TAG = "$BRANCH_NAME-build-$BUILD_NUMBER".replaceAll("/", "-")
    IMAGE_NAME = "$DOCKERHUB_REPO:$IMAGE_TAG"
    LATEST_IMAGE_NAME = "$DOCKERHUB_REPO:latest"
}
```

Рисунок 3.52 — Встановлення Jenkins змінних оточення

Якщо говорити про стадії, то першою стадією було простий показ версій(рис. 3.53), який може допомогти при відслідковуванні подальших несправностей та їх локального відтворення при потребі.

```
stages {
    stage('Show versions') {
        steps {
            sh '''
                    java -version
                    javac -version
                    docker --version
               '''
        }
    }
```

Рисунок 3.53 — Опис стадій Jenkins Pipeline

Запуск з консолі “./mvnw clean install”, запускає виконання тестів, створення jar-артефакту(рис. 3.54).

```
stage('Build application') {
    steps {
        sh './mvnw clean install'
    }
}
```

Рисунок 3.54 — Опис кроків для збірки аплікації

Стадія створення та доставки docker image(рис. 3.55) виконується тільки у випадку якщо це гілка dev чи master. При цьому, якщо це гілка “master”, то можна також оновити останній образ з тегом latest. Загалом після відправки образу, він більше нам не потрібен, тож його можна видалити.

```
stage('Build & Push docker image') {
    when { anyOf { branch 'master'; branch 'dev' } }

    steps {
        script {
            def springBootImage = docker.build(env.IMAGE_NAME)
            docker.withRegistry('', 'docker-hub-credentials') {
                springBootImage.push()
                if (env.BRANCH_NAME == 'master') {
                    springBootImage.push("latest")
                    sh "docker rmi ${env.LATEST_IMAGE_NAME}"
                }
                sh "docker rmi ${env.IMAGE_NAME}"
            }
        }
    }
}
```

Рисунок 3.55 — Збірка Docker образу в Jenkins

Стадії розгортання можуть відбуватись на dev та master(рис. 3.56), тож для цього було створено функцію під назвою “deployDockerContainer”(рис. 3.57) для уникнення дублікацій коду. Розгортання на master гілці вимагає підтвердження.

```
        stage('Deploy docker image(dev)') {
            when { branch 'dev' }
            steps {
                script {
                    deployDockerContainer(
                        "${env.IMAGE_NAME}",
                        "ims-spring-boot-dev",
                        8080,
                        ".ims-env-dev"
                    )
                }
            }
        }

        stage('Deploy docker image(master)') {
            when { branch 'master' }
            steps {
                timeout(time: 20, unit: 'MINUTES') { input message: 'Approve Deploy?', ok: 'Yes' }
                script {
                    deployDockerContainer(
                        "${env.LATEST_IMAGE_NAME}",
                        "ims-spring-boot",
                        80,
                        ".ims-env"
                    )
                }
            }
        }
    }
```

Рисунок 3.56 — Запуск розгортання Jenkins на різні гілки

Стадія розгортання відбувається за допомогою використання credentials, які використовуються при підключенні до сервера, де ми збираємося зберігати наші аплікації. Після підключення викликається інша функція, яка оновлює docker container, тим самим оновлює версію нашої програми на сервері.

```
def deployDockerContainer(String image, String container, Integer port, String evnFile) {
    withCredentials([string(credentialsId: 'ims-staging-user', variable: 'SSH_USER'),
                     string(credentialsId: 'ims-staging-host', variable: 'SSH_HOST')]) {
        sshagent(credentials: ['ims-staging-ssh-key']) {
            updateDockerContainer("$SSH_USER", "$SSH_HOST", image, container, port, evnFile)
        }
    }
}
```

Рисунок 3.57 — Доступ до віддаленого серверу через SSH

Під час виконання функції(рис. 3.58) відбуваються команди стягнення нової версії docker image, зупинка старого контейнеру, старт оновленого контейнеру, очистка старих образів.

```
def updateDockerContainer(String user, String host, String image, String container, Integer port, String evnFile) {
    commands = [
        "docker pull $image",
        "docker stop $container || true && docker rm $container || true",
        "docker run --env-file $evnFile \
                    --name $container \
                    -p $port:8080 \
                    --restart always \
                    -d $image",
        "docker image prune -af --filter 'until=12h'"
    ]
    commands.each { command ->
        sh "ssh -o StrictHostKeyChecking=no $user@$host '$command'"
    }
}
```

Рисунок 3.58 — Оновлення Docker образу через Jenkins

Результатом є успішне виконання всіх описаних стадій та кроків(рис. 3.59). При виконанні на гілці “master”сценарій пропускає виконання кроків, які призначені для “dev” гілки, а також для всіх PR.

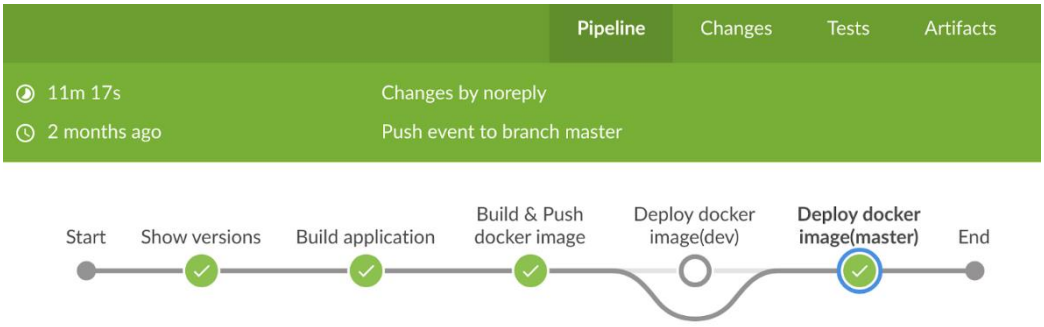


Рисунок 3.59 — Зовнішній вигляд успішного виконання Jenkins

3.4 Налаштування Github actions для фронтенд частини

3.4.1 Зв’язка з існуючим Github репозиторієм через .github/workflows

Для інтеграції Github Action з Github існує простий спосіб — створення .github/workflow папки в якій розмістити певного роду сценарії, описані в yml-файлах(рис. 3.60). Після цього можна оновити локальні зміни у віддаленому репозиторії. Результатом є файли які розпізнає Github(рис. 3.61), буде проводити збирання проекту.

```
$ cd ims-ui
$ mkdir -p .github/workflow
$ touch .github/workflow/ci.yml
$ touch .github/workflow/cd.yml
$ git commit -am "Add empty ci.yml, cd.yml"
$ git push origin master
```

Рисунок 3.60 — Добавлення файлів за допомогою git

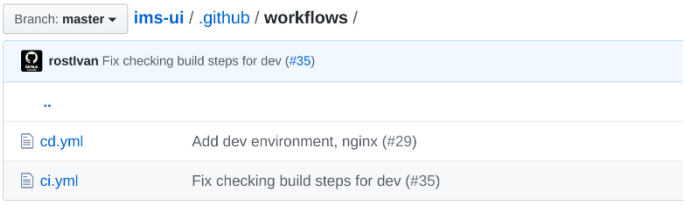


Рисунок 3.61 — Файли ci.yml, cd.yml на Github

3.4.2 Створення ci.yml, cd.yml з сценаріями збирання та створенням артефакту

Сама структура сценарію — звичайний yml-файл, який описує ряд операцій. Синтаксис Github Actions має власну термінологію:

- workflow — сценарії роботи

- job — процес, одиниця сценарію роботи, складається з одного або декількох завдань.

- step — завдання складається з одного або декількох кроків, які виконуються один за одним

- action — кожен крок складається з однієї або декількох дій. По суті — найменша одиниця виконання.

Передбачено 2 типи workflow(лістинг коду наведено в додатку В та додатку Г):

- ci.yml — файл для опису CI процесу

- cd.yml — файл для опису CD процесуCI процес(рис. 3.62):

- виконується на події: push, pull request

- проходить в оболонці nodejs контейнера, який запускається на ОС ubuntu

- включає запуск з консолі встановлення пакетів та збірку dev версії фронтенд аплікації

CD процес(рис. 3.63):

- виконується тільки на push гілки dev чи master

- проходить на в тому ж самому середовищі, що і білд

- визначає на якій з гілок буде йти збірка та виставляє відповідні змінні оточення

- використовує готовий плагін для оновлення образу в Dockerhub

```yaml
name: Verify code building

on: [push, pull_request]

jobs:
  build:
    runs-on: ubuntu-latest
    container:
      image: node:12.14.1-alpine

    steps:
      - uses: actions/checkout@v1
      - name: Install & Build
        run: |
          npm install
          npm run build:dev
```

Рисунок 3.62 — Файл ci.yml для Github Actions

```yaml
name: Publish to Registry
on:
  push:
    branches:
      - master
      - dev
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Set env to dev
        if: endsWith(github.ref, '/dev')
        run: |
          echo "::set-env name=ENV::dev"
      - name: Set env to prod
        if: endsWith(github.ref, '/master')
        run: |
          echo "::set-env name=ENV::prod"
      - name: Checkout
        uses: actions/checkout@master
      - name: Build & Push docker image
        uses: elgohr/Publish-Docker-Github-Action@master
        with:
          registry: docker.io
          name: if103java/ims-ui
          username: ${{ secrets.DOCKER_USERNAME }}
          password: ${{ secrets.DOCKER_PASSWORD }}
          snapshot: true
          buildargs: ENV
```

Рисунок 3.63 — Файл cd.yml для Github Actions

3.4.3 Збереження secrets в Github

Для того щоб використовувати Dockerhub у межах Github Actions потрібно використовувати логін та пароль, який зберігається у зашифрованому вигляді в самому Github(рис. 3.64). При потребі можна звернутись до таких даних за допомогою способу, який продемонстровано на рис. 3.65.

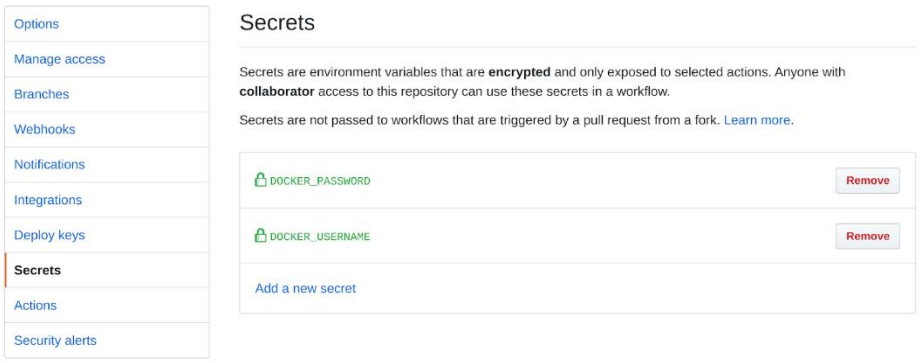


Рисунок 3.64 — Збереження секретних змінних в Github Actions

```
username: ${{ secrets.DOCKER_USERNAME }}
password: ${{ secrets.DOCKER_PASSWORD }}
```

Рисунок 3.65 — Доступ до секретних змінних Github Actions Pipeline

3.5 Інтеграція використання Docker контейнерів в існуючу інфраструктуру

3.5.1 Додавання Dockerfile до двох проектів

Docker може автоматично створювати образи читаючи інструкції з Dockerfile. Файл Dockerfile — текстовий документ, містить всі команди для складання образу. За допомогою команди “docker build” можна зібрати образ з послідовності інструкцій вказаному у Dockerfile. Docker виконує інструкції з

Dockerfile по порядку. Dockerfile було добавлено до 2-ох проектів — фронтенд та бекенд (лістинг коду наведено в додатку Д та в додатку Е).

Першою інструкцією повинна бути `FROM`, що задає базовий образ на основі якого буде відбуватися складання. У даному випадку це openjdk версії 13, яка включає JDK, необхідну для роботи з Java Spring Boot бекенд аплікацією(рис. 3.66).

Інструкція VOLUME створює том з заданим ім'ям і робить його спільним для хоста та контейнера. Тут використовується /tmp, яка насправді необхідна вбудованому Tomcat у jar-файл.

Інструкція ARG задає змінні які користувач передає у команді “docker build” за допомогою --build-arg.

Інструкція COPY копіює нові файли або каталог з файлової системи хоста у файлову систему контейнера. В даному випадку копіюється ims.jar, який можна буде запустити.

Інструкція ENTRYPOINT дозволяє налаштувати контейнер так що б він працював як виконуваний файл, тобто запуск програми відбувається саме тут.

```
FROM openjdk:13.0.1

VOLUME /tmp
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} ims.jar
ENTRYPOINT ["java", "--enable-preview", "-jar", "/ims.jar"]
```

Рисунок 3.66 — Dockerfile бекенд ims аплікації

Dockerfile фронтенд аплікації(рис. 3.67) трохи відрізняється наявністю базового образу на базі Alpine Linux, який займає меншу кількість дискового простору. Для його побудови копіюється весь сирцевий код, встановлюються необхідні пакети, робиться збірка, яка в свою чергу створює необхідні файли в /app/dist директорії, запускается nginx веб-сервер для обслуговування нашої Angular програми.

```
FROM node:12.14.1-alpine AS builder
WORKDIR /app
ARG ENV=prod
COPY package.json package-lock.json ./
RUN npm install
COPY . .
RUN npm run build:$ENV

FROM nginx:1.17.8-alpine
WORKDIR /usr/share/nginx/html/
COPY --from=builder /app/dist/* .
COPY --from=builder /app/nginx.conf /etc/nginx/conf.d/default.conf
```

Рисунок 3.67 — Dockerfile фронтенд ims-ui аплікації

3.5.2 Створення Docker Hub репозиторіїв для зберігання образів

У зв’язку з тим, що образи будуть зберігатись на Dockerhub, то необхідно створити обліковий запис, відповідно, в його межах, нові репозиторії, для фронтенд та бекенд аплікацій.

1. Вхід в акаунт(рис. 3.68)
2. Поетапне створення 2 репозиторів(рис. 3.69)
3. Локальне створення образу для ims(Spring Boot) та його відправка до сховища(рис. 3.70)
4. Локальне створення образу для ims-ui(Angular) та його відправка до сховища(рис. 3.71)
5. Результатом є їхня наявність в Dockerhub та можливість завантажити їх собі(рис. 3.72)

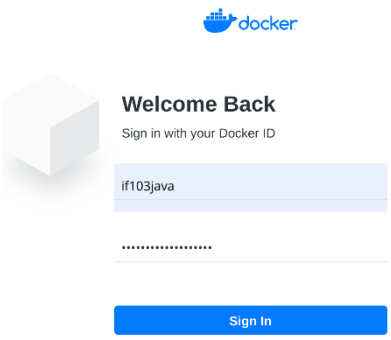


Рисунок 3.68 — Вхід в особистий обл. запис Dockerhub

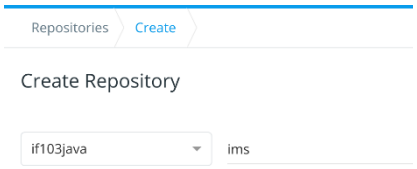


Рисунок 3.69 — Створення репозиторію в Dockerhub

```
$ cd ims
$ docker build -t if103java/ims:latest .
$ docker push if103java/ims:latest
```

Рисунок 3.70 — Створення, пуш Docker образу для бекенд аплікації

```
$ cd ims-ui
$ docker build --build-arg ENV=dev -t if103java/ims-ui:dev .
$ docker push if103java/ims-ui:dev
```

Рисунок 3.71 — Створення, пуш Docker образу для фронтенд аплікації

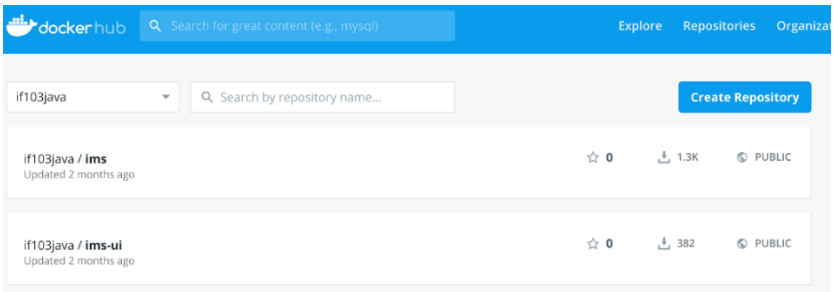


Рисунок 3.72 — Список образів аплікацій на Dockerhub

3.5.3 Автоматизація розгортання шляхом використання Jenkins

При оновленні образу фронтенд аплікації можна автоматично робити розгортання оновлення цього образу на віддаленому сервері. Для цього можна скористатись Dockerhub Webhook, який буде спрацьовувати на “push” нового docker image. Цей вебхук буде потрапляти на Jenkins та сприяти розгортанню аплікації. Generic Webhook Trigger — це плагін Jenkins, який може:

1. Отримувати будь-який HTTP запит за адресою “$JENKINS_URL/generic-webhook-trigger/invoke”

2. Витягнути значення з Webhook Payload
3. Запустити збірку з цими значеннями як змінними оточення, що буде виконувати якийсь певний заданий сценарій(лістинг наведено в додатку Б)

Для налаштування повного циклу треба:

1. Перейти в фронтенд репозиторії на Dockerhub в секцію Webhooks(рис. 3.73)
2. Встановити Jenkins generic-webhook-trigger(рис. 3.74)
3. Добавити змінну оточення “TAG”, яка буде читатись з вебхуку за шляхом “push_data.tag”(рис. 3.75)
4. Встановити фільтрацію на тег, який може бути 2-х типів — dev, latest, оскільки dev — останній образ з гілки dev, а latest — з гілки master(рис. 3.76)
5. Створити Jenkinsfile(рис. 3.77). У обох стадіях використано функцію “deployDockerContainer”, яку було взято з попереднього Jenkins Pipeline

Результатом є робочий Generic Webhook Pipeline(рис. 3.78), який буде спрацьовувати тільки в ті моменти, коли нам це насправді потрібно.

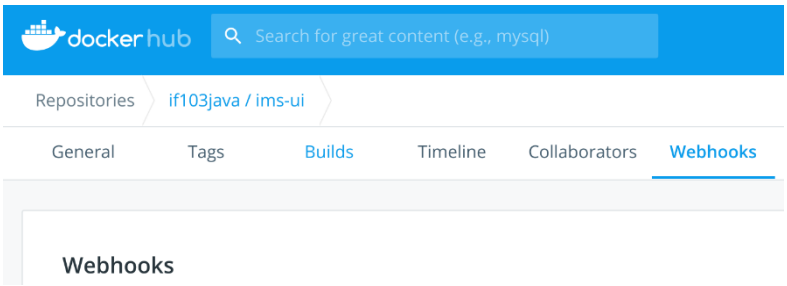


Рисунок 3.73 — Добавлення webhook до Dockerhub репозиторію

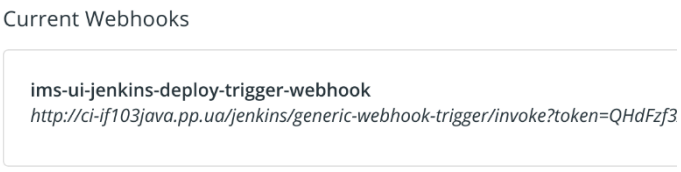


Рисунок 3.74 — Приклад URL для Jenkins generic-webhook

Рисунок 3.75 — Парсинг змінних generic-webhook

Рисунок 3.76 — Фільтрація generic-webhook

```
stages {
  stage('Deploy dev angular image') {
    when { environment name: 'TAG', value: 'dev' }
    steps {
      script {
        deployDockerContainer("${env.DEV_IMAGE_NAME}", "ims-ui-angular-dev", 4201)
      }
    }
  }

  stage('Deploy latest angular image') {
    when { environment name: 'TAG', value: 'latest' }
    steps {
      script {
        deployDockerContainer("${env.LATEST_IMAGE_NAME}", "ims-ui-angular", 4200)
      }
    }
  }
}
```

Рисунок 3.77 — Стадії розгортання для фронтенд алпікації

Рисунок 3.78 — Результат виконання Jenkins Pipeline

3.6 Створення клауд БД(RDS) для потреб бекенд аплікації

3.6.1 Створення RDS на базі MySQL

Для створення екземпляра БД на базі MySQL в межах RDS варто:

- Увійти до AWS Management Console, відкрити Amazon RDS console(рис. 3.79)

- Вибрати створення MySQL БД(рис. 3.80)

- Вказати версію БД, тип екземпляру БД, унікальний ідентифікатор, логін, пароль, частоту автоматичного створення бекапів(рис. 3.81)

Після створення БД є доступною для вхідних з'єднань. Для підключення потрібно знати тільки url:port, який дозволить встановити з'єднання.

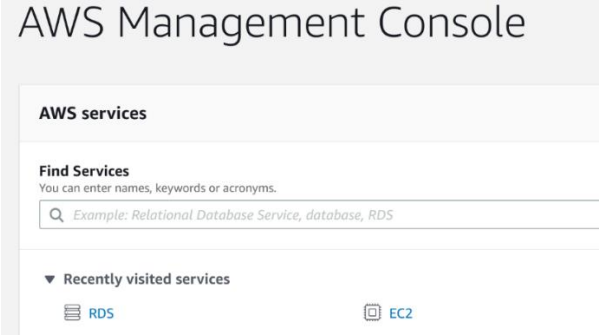


Рисунок 3.79 — Консоль керування AWS

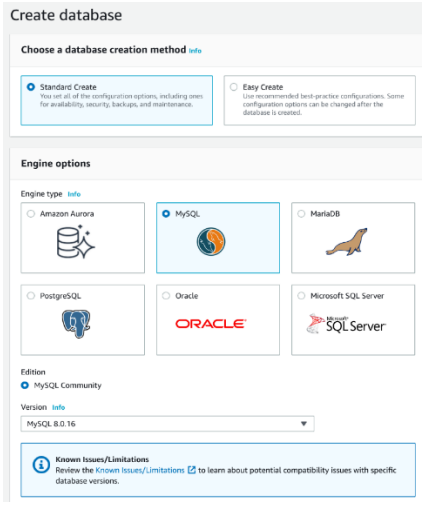


Рисунок 3.80 — Створення RDS

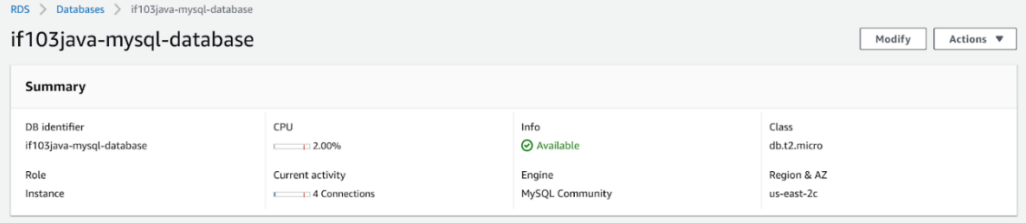


Рисунок 3.81 — Екземпляр RDS

3.6.2 Інтеграція з Spring Boot

При використанні віддаленої БД, одним з необхідних умов є саме підключення до неї. Воно може відбуватись різними способами, проте, в любому випадку, варто десь зберігати такі змінні як: адреса бази даних, її користувач, пароль.

При використанні Spring Boot існує можливість легко зчитувати змінні з файлів та використовувати їх в Java коді. Файли *.properties використовуються для збереження певних змінних у вигляді ключ-значення у одному файлі, для запуску програми в іншому середовищі. Наприклад при локальному запуску та на сервері, адже там ці змінні можуть бути іншими.

У Spring Boot властивості зберігаються у файлі application.properties під classpath. Файл application.properties(рис. 3.82) знаходиться в папці src/main/resources.

```
→  ims git:(master) cat src/main/resources/application.properties
#MySQL DB Config
db.driver=com.mysql.cj.jdbc.Driver
db.url=jdbc:mysql://host:port/db_name
db.username=db_username
db.password=db_password
db.minPoolSize=3
db.maxPoolSize=18
db.poolIncrement=3
```

Рисунок 3.82 — Вміст файлу application.properties

Анотація @Value використовується для зчитування значення властивості в коді Java(рис. 3.83). Синтаксис для зчитування значення властивості

показаний нижче. При цьому, якщо ці змінні можуть бути перевантажені за допомогою використання змінних середовища. Наприклад, “db.username” з application.properties можна перезаписати змінною оточення з назвою “DB_USERNAME”.

```
@Value("${db.driver}")
private String driverClassName;

@Value("${db.url}")
private String url;

@Value("${db.username}")
private String username;

@Value("${db.password}")
private String password;

@Value("${db.minPoolSize}")
private int minPoolSize;

@Value("${db.maxPoolSize}")
private int maxPoolSize;

@Value("${db.poolIncrement}")
private int poolIncrement;
```

Рисунок 3.83 — Змінні з application.properties в Java коді

Для того, щоб приєднатись до БД можна скористатись змінними, які ми прочитали з properties-файлу. При цьому, для створення з’єднання, використовувався готовий конектор — “c3p0”. Сам c3p0 — це бібліотека Java, що надає зручний спосіб управління з'єднаннями з базою даних, який надає ComboPooledDataSource. Його можна використовувати при створенні JdbcTemplate(рис. 3.84). За допомогою JdbcTemplate можна робити різного роду запити до БД.

```
@Bean
public DataSource dataSource() throws PropertyVetoException {
    ComboPooledDataSource dataSource = new ComboPooledDataSource();

    dataSource.setDriverClass(driverClassName);
    dataSource.setJdbcUrl(url);
    dataSource.setUser(username);
    dataSource.setPassword(password);
    dataSource.setMinPoolSize(minPoolSize);
    dataSource.setAcquireIncrement(poolIncrement);
    dataSource.setMaxPoolSize(maxPoolSize);

    return dataSource;
}

@Bean
public JdbcTemplate jdbcTemplate(DataSource dataSource) {
    return new JdbcTemplate(dataSource);
}
```

Рисунок 3.84 — Створення Connection Pool

4 ОБГРУНТУВАННЯ ЕКОНОМІЧНОЇ ДОЦІЛЬНОСТІ ПРОЕКТУ

DevOps зазвичай вводиться в організації з певних, добре зрозумілих і, як правило, дуже виправданих причин. Такі компанії як Google, Amazon, Netflix, IBM, Microsoft будують свою архітектуру на базі мікросервісів, тож без використання DevOps методик вони не зможуть розвивати свої продукти. DevOps прискорює загальний темп циклу розробки та випуску програмного забезпечення, покращує якість доставки коду та допомагає вирішувати проблеми швидко та якісно. Проте ці всі нематеріальні якості було би правильно оцінити у вигляді цифр. Це означає, що будь-яка добре розроблена програма повинна містити здоровий погляд на рентабельність інвестицій.

ROI (return on investment) — коефіцієнт окупності. Він демонструє прибутковість або збитковість тієї чи іншої інвестиції, вимірюється у відсотках[28]. Для розрахунку потрібно знати(формула 4.1): дохід від вкладень і їх розмір.

$$ROI = \frac{NI}{COI} \times 100\%, \qquad (4.1)$$

де *NI* — Чистий дохід(Net Income);

COI — Вартість інвестицій(Cost of Investment).

При цьому потрібно розуміти з чого складається чистий дохід та у що варто інвестувати кошти(формула 4.2).

Чистий дохід складається з:

- Відсутності втрат коштів на downtime(проміжок часу, коли сервіс не працює)

- Виграшу в продуктивності команди

Вартість інвестицій складається з:

- Витрат на пошук відповідних та рекрутингу спеціалістів в області DevOps

- Витрат на підтримку інфраструктури та засобів, які необхідні для функціонування системи

$$ROI = \frac{(DL + PG) - (RC + TC)}{RC + TC} \times 100\%, \qquad (4.2)$$

де *DL* — гроші, втрачені через простоювання системи(Downtime Lost);
PG — кошти від підвищення продуктивності праці(Productivity Gain);
RC — витрати на пошук спеціалістів(Recruitment Costs);
TC — витрати на підтримку сервісів(Tools Costs).

За даними дослідження, проведеного IDC та AppDynamics[29], втрата за годину простою(Downtime) від 8,580 до 686 250 доларів, залежно від розміру компанії. І в середньому компанії, що стикаються з простоями, втрачають 163 674 доларів на рік. Варто також зазначити, що середня зарплата за роботу адміністратора складає 16.42 доларів за год.[30] Тож при використанні даних практик можна націлити його роботу на більш важливі речі та, як результат, покращити його продуктивність. Також це досягається через те, що QA з годинною оплатою в 19.15 доларів[31], не потребують чекати поки новий функціонал потрапляє на сервер, адже це робиться автоматично.

При цьому показники з формули (4.2) не можуть бути правильно виміряними, оскільки по своїй природі є абстрактними та такими, які майже неможливо передбачити. Чому не вимірюється рентабельність інвестицій DevOps? Рентабельність інвестицій DevOps не вимірюється, оскільки вона насправді не може бути виміряна. DevOps — це не стільки інвестиція, скільки операційний процес, а кількість невідомих змінних в організації занадто велика, щоб зробити такий розрахунок можливим. DevOps — це неможливий для

вимірювання процес щодо ROI, що безумовно, має значення, але його неможливо незалежно та точно виміряти[32].

На сьогодні, за даними IDC[33], 20% проектів з розробки програмного забезпечення отримують перевагу від підходу DevOps. До 2021 року ця цифра зросте до від 35 до 40%. Не випадково так багато IT-команд застосовують методологію DevOps для проекту, адже до використання цього сприяє:

- економія часу на вирішення проблем
- пришвидшення запуску нових функцій
- зменшення ризиків завдяки автоматизації процесів
- значне підвищення задоволення клієнтів від якості кінцевого продукту

Ряд факторів які дозволяють DevOps бути вигідними для сторін бізнесу та постачальників послуг:

- Швидкий час виходу на ринок
- Менша кількість співробітників, проте підвищена продуктивність
- No Downtime
- Скорочення інфраструктурних витрат
- Поліпшення якості та продуктивності програм

Це все, швидше, про перспективу. Очевидно, що всі ці фактори(рис. 4.1) важливі, оскільки кожен з часом перетворюється на якусь цінність.

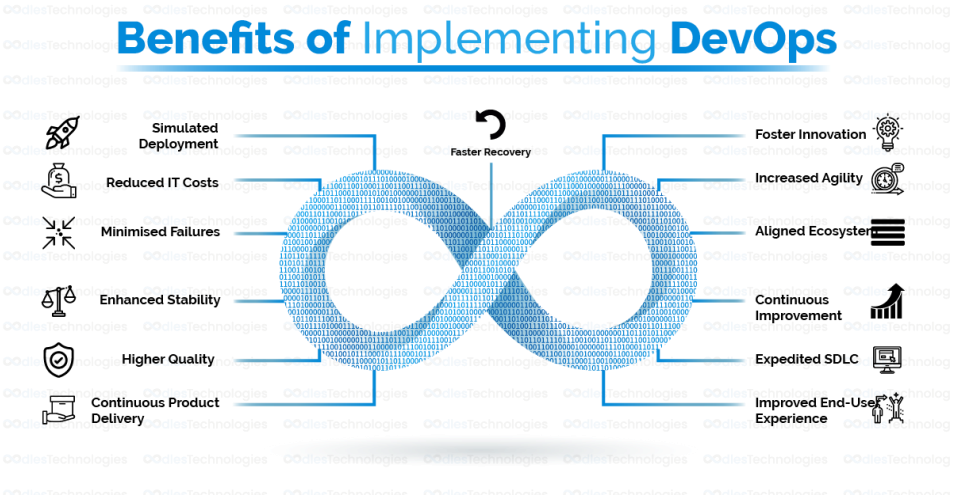
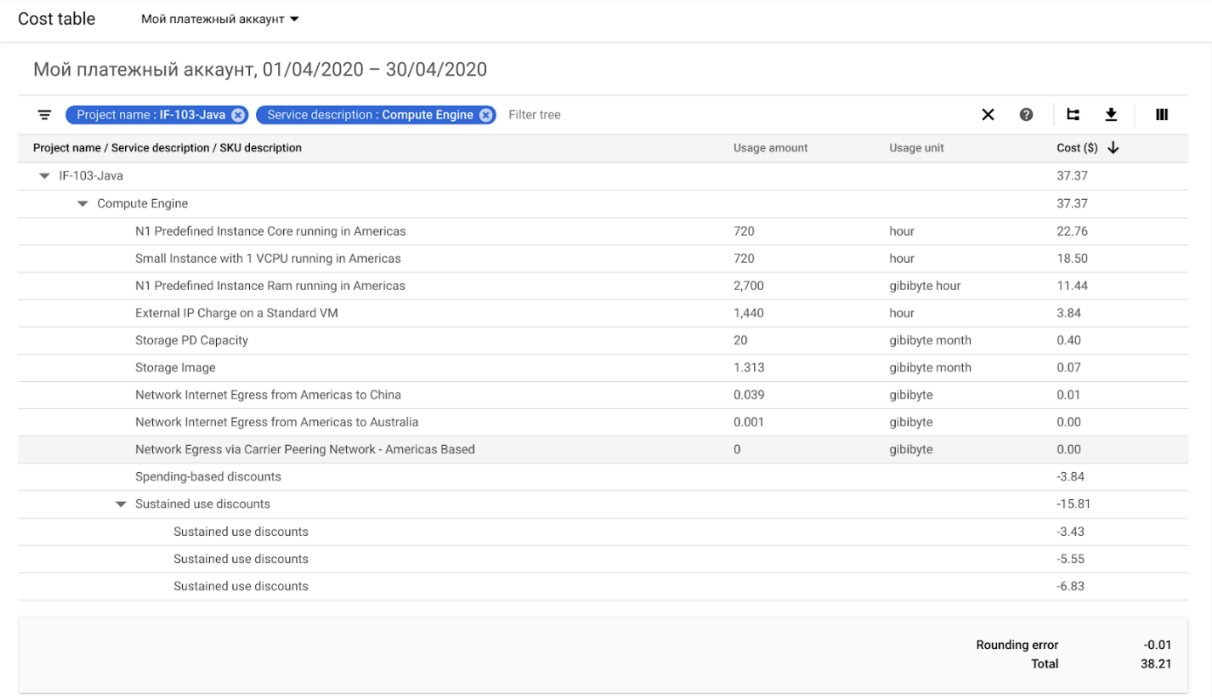


Рисунок 4.1 — Переваги використання DevOps[34]

Зокрема, варто зазначити, що спроба імплементації та підтримки простого рішення для порівняння не є дорогою. Загалом, система розроблена у рамках даної роботи, обійшлася всього у 39 доларів за місяць(рис. 4.2). Вона включає в себе хостинг CI / CD та 4-ох аплікацій на GCP.



Cost table Мой платежный аккаунт

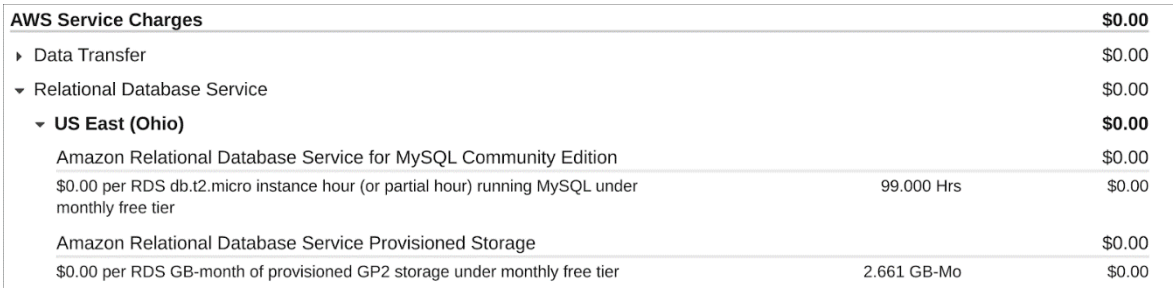
Мой платежный аккаунт, 01/04/2020 – 30/04/2020

Project name : IF-103-Java | Service description : Compute Engine | Filter tree

Project name / Service description / SKU description	Usage amount	Usage unit	Cost ($)
IF-103-Java			37.37
Compute Engine			37.37
N1 Predefined Instance Core running in Americas	720	hour	22.76
Small Instance with 1 VCPU running in Americas	720	hour	18.50
N1 Predefined Instance Ram running in Americas	2,700	gibibyte hour	11.44
External IP Charge on a Standard VM	1,440	hour	3.84
Storage PD Capacity	20	gibibyte month	0.40
Storage Image	1.313	gibibyte month	0.07
Network Internet Egress from Americas to China	0.039	gibibyte	0.01
Network Internet Egress from Americas to Australia	0.001	gibibyte	0.00
Network Egress via Carrier Peering Network - Americas Based	0	gibibyte	0.00
Spending-based discounts			-3.84
Sustained use discounts			-15.81
Sustained use discounts			-3.43
Sustained use discounts			-5.55
Sustained use discounts			-6.83
Rounding error			-0.01
Total			38.21

Рисунок 4.2 — Витрати на GCP для реалізації та підтримки інфраструктури

Відповідно до цього, багато клауд-провайдерів роблять знижки за часте використання, або ж надають якісь послуги безкоштовно на певний період часу. Як, наприклад, це робить AWS для деяких своїх сервісів. RDS можна спробувати безкоштовно на протязі 1 року(рис. 4.3).



AWS Service Charges		$0.00
Data Transfer		$0.00
Relational Database Service		$0.00
US East (Ohio)		$0.00
Amazon Relational Database Service for MySQL Community Edition		$0.00
$0.00 per RDS db.t2.micro instance hour (or partial hour) running MySQL under monthly free tier	99.000 Hrs	$0.00
Amazon Relational Database Service Provisioned Storage		$0.00
$0.00 per RDS GB-month of provisioned GP2 storage under monthly free tier	2.661 GB-Mo	$0.00

Рисунок 4.3 — Витрати на AWS для реалізації та підтримки інфраструктури

DevOps — це дуже практичний і цінний актив для сьогоднішніх організацій через широкий спектр реальних переваг завдяки постійній інтеграції та постійній доставці без погіршення якості продуктів та послуг. Використовуючи Agile, співпраця між командами Development та Ops приводить до швидкого циклу розвитку, покладаючись на терміни і контроль витрат, покращують рентабельність інвестицій.

В конкретній ситуації, під час розробки даного рішення, воно виправдало себе економічно та допомогло у багатьох ситуаціях під час імлементації аплікації, при перевірці коду, а особливо, при його розгортанні, таким чином не витрачаючи час на це, а концентруючи свою увагу на розробці нового функціоналу. Нові можливості додатку можна було одразу бачити, тож це задовольнило вимоги клієнтів.

Підсумовуючи, можна зрозуміти чому це важливо. Причиною є мобільність та гнучкість ваших додатків та команди в цілому. Відгуки клієнтів та тестування — це дуже важливо, коли справа стосується створення хорошого продукту, який переживе початковий етап. Через це, є сенс адаптовуватись відповідно до вимог ринку, щоб продукт був хорошим та здатним виконувати свою роботу. Однак зробити це непросто через тривалі цикли ітерації та масштабування великої команди. Найбільша перевага DevOps, з точки зору бізнесу, очевидна — мова йде про швидкість доставки. Коли ви реалізуєте культуру DevOps, вона дозволяє ефективно і вчасно вносити зміни з можливістю хорошого кінцевого результату. Кінцевий результат — кращий продукт в цілому, який задовольняє бізнес потребам.

ВИСНОВКИ

В межах даного дипломного проекту було розроблено автоматизовану систему, яка дозволяє покращити процеси створення, тестування та впровадження програмного забезпечення, що відбувається за допомогою адаптації різного роду готових Pipeline, які в свою чергу, дозволяють описати повторюваний сценарій перевірки та доставки готового продукту до користувачів. На основі поставлених проектних задач було виконано: перенесення проектів до віддалених репозиторіїв, створення 2 екземплярів VM в системі GCP, створення та інтеграцію клауд БД в існуючий проект, використовуючи RDS в межах AWS, контейнерезацію аплікацій через Docker контейнери, імплементацію сценаріїв перевірки коду та його доставки для фронтед та бекенд проектів, за допомогою використання Github Actions та Jenkins.

Зазвичай рішення введення DevOps в проект є вимушеним з відповідних причин, проте в умовах розробленого проекту, він був залучений для автоматизації розгортання та покращення якості коду, разом з тим — продуктивності команди, а не для отримання якоїсь грошової вигоди на довготривалій основі. Результати були більш ніж задовільними, так як це позитивно повпливало на швидкість командної розробки. Для порівняння 1 staging реліз в місяць перетворився у 4, що є кращим показником ніж було раніше, при цьому впевненість в працездатності та правильності роботи аплікації залишилась на хорошому рівні. Повторне застосування для аналогічних проектів, якась певна адаптація даного рішення чи використання схожих принципів можливе та легко інтергується в інсуючі аплікації різного роду та не вимагає значних зусиль на їх реалізацію.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

REFERENCES

1. Дженніфер Девіс Філософія DevOps. Мистецтво управління IT : друковане видання : 2018. 92 с.
2. DevOps - wiki. URL: https://uk.wikipedia.org/wiki/DevOps (дата звернення: 05.11.2019)
3. Одиниці DevOps. URL: https://ztabs.co/wpcontent/uploads/2019/09/What-is-Devops.png (дата звернення: 05.11.2019)
4. Складові DevOps. URL: https://media.joomeo.com/large/5a7c8328464d5.jpg (дата звернення: 05.11.2019)
5. CI - wiki. URL: https://uk.wikipedia.org/wiki/Неперервна_інтеграція (дата звернення: 07.11.2019)
6. CD - wiki. URL: https://uk.wikipedia.org/wiki/Безперервна_доставка (дата звернення: 07.11.2019)
7. Еберхард Вольф Continuous delivery. Практика неперервних апдейтів : друковане видання : 2016. 78 с.
8. Стадії процесів CI/CD. URL: https://fiverr-res.cloudinary.com/images/q_auto,f_auto/gigs2/151126533/original/c9e9877f800871a67434921ef4ba8b2eacfb5e01/setup-pipelines-for-git-gitlab-ci-cd.png (дата звернення: 08.11.2019)
9. Системи керування версіями - studopedia. URL: https://studopedia.com.ua/1_403314_sistemi-keruvannya-versiyami.html (дата звернення: 05.12.2019)
10. Система відгалужень Git. URL: https://lh3.googleusercontent.com/proxy/R2zJ1tnU_REQJwRoJM4LNkR23vDJEhsA7TNP0MASw8ZyIUVbt2kD8Ro4ZcTXpYSjZqWxL4YWX0aExhirwemLYkp5onml8-Mx1i2odr6lNYr02HSq (дата звернення: 05.12.2019)

11. Основні компоненти Docker. URL: https://www.georgestudenko.com/wp-content/uploads/2019/12/docker-engine-components-flow-freshblue-492x277.png (дата звернення: 10.12.2019)
12. Принцип роботи веб-сервісів для спільної розробки програмного забезпечення. URL: https://d1jnx9ba8s6j9r.cloudfront.net/blog/wp-content/uploads/2017/12/gitHub.png (дата звернення: 16.12.2019)
13. Rafal Leszko Continuous Delivery with Docker and Jenkins: Delivering software at scale : друковане видання : 2018. 140 с.
14. Joseph Muli Beginning DevOps with Docker : друковане видання : 2019. 215 с.
15. Gene Kim The DevOps Handbook : друковане видання : 2019. 302 с.
16. Jenkins-Docker Continuous Integration & Continuous Deployment Pipeline - medium. URL: https://medium.com/@prakashkumar0301/docker-jenkins-ci-cd-pipeline-dd54854125f3 (дата звернення: 12.03.2020)
17. How to CI and CD a Node.JS Application Using GitHub Actions - medium. URL: https://blog.bitsrc.io/https-medium-com-adhasmana-how-to-do-ci-and-cd-of-node-js-application-using-github-actions-860007bebae6 (дата звернення: 16.04.2020)
18. Популярні AWS сервіси. URL: https://qph.fs.quoracdn.net/main-qimg-c6728894b228aad3ac89ac95d0e82573 (дата звернення: 25.01.2020)
19. Популярні GCP сервіси. URL: https://www.gonnasolutions.com/wp-content/uploads/2019/01/google-cloud-platform.jpg (дата звернення: 25.01.2020)
20. Архітектура Spring Boot аплікацій. URL: https://miro.medium.com/max/1200/0*ifhmxzv1I_AAwLyH.png (дата звернення: 30.01.2020)
21. Складові Spring Boot аплікацій. URL: https://www.javakurs.ch/blog/springboot/springboot2/simplearchitecture.png (дата звернення: 30.01.2020)

22. Архітектура Angular аплікацій. URL: https://lh6.googleusercontent.com/proxy/HKscpFMp71_WyCRplLfmBO0ghVBtiEWo5CahLPquEMPOhGMMNR5u_uq9k3TlgpwgpIe_PZ1CY2KMhMCuV_B3t9m5iPGSLe2EGKu7yEJb4y7xEh4=w1200-h630-p-k-no-nu (дата звернення: 25.01.2020)
23. Структура Maven проекту. URL: https://upload.wikimedia.org/wikipedia/commons/thumb/c/cf/Maven_CoC.svg/200px-Maven_CoC.svg.png (дата звернення: 25.01.2020)
24. Структура Angular проекту. URL: https://s.dou.ua/storage-files/angular-cli-700.png (дата звернення: 25.01.2020)
25. Scrum - wiki. URL: https://ru.wikipedia.org/wiki/SCRUM (дата звернення: 25.01.2020)
26. Архітектура взаємодії Client – API – DB. URL: https://i.pinimg.com/600x315/3a/9b/03/3a9b03c69d4355d1a849b16452e4948b.jpg (дата звернення: 25.01.2020)
27. Візуалізація роботи CI / CD системи. URL: https://cdn-images-1.medium.com/max/1200/0*HEK2WLuM2qTqf95N.png (дата звернення: 25.01.2020)
28. ROI Formula - corporatefinanceinstitute. URL: https://corporatefinanceinstitute.com/resources/knowledge/finance/return-on-investment-roi-formula/ (дата звернення: 20.02.2020)
29. The True Cost of Downtime - appdynamics. URL: https://www.appdynamics.com/blog/product/the-true-cost-of-downtime-infographic/ (дата звернення: 26.02.2020)
30. Average Office Administrator Hourly Pay - payscale. URL: https://www.payscale.com/research/US/Job=Office_Administrator/Hourly_Rate (дата звернення: 15.03.2020)

31. Hourly Rate for Skill: Quality Assurance (QA) - payscale. URL: https://www.payscale.com/research/US/Skill=Quality_Assurance_(QA)/Hourly_Rate (дата звернення: 15.03.2020)
32. Бер К., Спаффорд Д. Проект «Фенікс». Роман про те, як DevOps змінює бізнес на краще : друковане видання : 2015. 29 с.
33. Бер К., Спаффорд Д. Проект «Фенікс». Роман про те, як DevOps змінює бізнес на краще : друковане видання : 2015. 85 с.
34. Переваги використання DevOps. URL: https://assets-global.website-files.com/5d84c93db32e46eb891ea57c/5eaacc9a4c1f30b483dee062_bVJuP0h3pZakOQ2YJm_hn8lF0TEIkzjhYVMXvUQtvBNBDKEUzrN_rkl3uEQUIq1B-qRqxPciB35gzoab3U3Xs-L6AqWVDPjT1_oOxo7ZbuWdRZrtd5d7hWFUVIDXVoCsEPeXF09N.jpeg (дата звернення: 05.04.2020)
35. CIT 2:2018. Стандарт кафедри інформаційних технологій. Дипломний проект. Вимоги до змісту та оформлення [Чинний від 2018-09-03]. Вид. офіц. Івано-Франківськ, 2018. 43 с.

ДОДАТОК А

Лістинг коду Jenkins Pipeline для CI / CD бекенд проекту

```
#!/usr/bin/env groovy
pipeline {
    agent any
    tools {
        jdk "JDK-13.0.1"
    }
    environment {
        DOCKERHUB_REPO = "if103java/ims"
        IMAGE_TAG = "$BRANCH_NAME-build-
$BUILD_NUMBER".replaceAll("/", "-")
        IMAGE_NAME = "$DOCKERHUB_REPO:$IMAGE_TAG"
        LATEST_IMAGE_NAME = "$DOCKERHUB_REPO:latest"
    }

    stages {
        stage('Show versions') {
            steps {
                sh '''
                    java -version
                    javac -version
                    docker --version
                '''
            }
        }
        stage('Build application') {
```

```
    steps {
        sh './mvnw clean install'
    }
}

stage('Build & Push docker image') {
    when { anyOf { branch 'master'; branch 'dev' } }

    steps {
        script {
            def springBootImage = docker.build(env.IMAGE_NAME)
            docker.withRegistry('', 'docker-hub-credentials') {
                springBootImage.push()
                if (env.BRANCH_NAME == 'master') {
                    springBootImage.push("latest")
                    sh "docker rmi ${env.LATEST_IMAGE_NAME}"
                }
                sh "docker rmi ${env.IMAGE_NAME}"
            }
        }
    }
}

stage('Deploy docker image(dev)') {
    when { branch 'dev' }
    steps {
        script {
            deployDockerContainer(
```

```
                "${env.IMAGE_NAME}",
                "ims-spring-boot-dev",
                8080,
                ".ims-env-dev"
            )
        }
    }
}

stage('Deploy docker image(master)') {
    when { branch 'master' }
    steps {
        timeout(time: 20, unit: 'MINUTES') { input message: 'Approve Deploy?', ok: 'Yes' }
        script {
            deployDockerContainer(
                "${env.LATEST_IMAGE_NAME}",
                "ims-spring-boot",
                80,
                ".ims-env"
            )
        }
    }
}
}
```

```
def deployDockerContainer(String image, String container, Integer port, String
evnFile) {
    withCredentials([string(credentialsId: 'ims-staging-user', variable: 'SSH_USER'),
                     string(credentialsId: 'ims-staging-host', variable: 'SSH_HOST')]) {
        sshagent(credentials: ['ims-staging-ssh-key']) {
            updateDockerContainer("$SSH_USER", "$SSH_HOST", image, container,
port, evnFile)
        }
    }
}


def updateDockerContainer(String user, String host, String image, String container,
Integer port, String evnFile) {
    commands = [
        "docker pull $image",
        "docker stop $container || true && docker rm $container || true",
        "docker run --env-file $evnFile \
                    --name $container \
                    -p $port:8080 \
                    --restart always \
                    -d $image",
        "docker image prune -af --filter 'until=12h'"
    ]

    commands.each { command ->
        sh "ssh -o StrictHostKeyChecking=no $user@$host '$command'"
    }
}
```

ДОДАТОК Б

Лістинг коду Jenkins Pipeline для CD фронтенд проекту

```
#!/usr/bin/env groovy
pipeline {
  agent any
  environment {
    DOCKERHUB_REPO = "if103java/ims-ui"
    DEV_IMAGE_NAME = "$DOCKERHUB_REPO:dev"
    LATEST_IMAGE_NAME = "$DOCKERHUB_REPO:latest"
  }

  stages {
    stage('Deploy dev angular image') {
      when { environment name: 'TAG', value: 'dev' }
      steps {
        script {
          deployDockerContainer("${env.DEV_IMAGE_NAME}", "ims-ui-angular-
dev", 4201)
        }
      }
    }

    stage('Deploy latest angular image') {
      when { environment name: 'TAG', value: 'latest' }
      steps {
        script {
```

```
        deployDockerContainer("${env.LATEST_IMAGE_NAME}", "ims-ui-
angular", 4200)
       }
     }
   }
  }
}

def deployDockerContainer(GString image, String container, Integer port) {
  withCredentials([string(credentialsId: 'ims-staging-user', variable: 'SSH_USER'),
                   string(credentialsId: 'ims-staging-host', variable: 'SSH_HOST')]) {
    sshagent(credentials: ['ims-staging-ssh-key']) {
      updateDockerContainer("$SSH_USER", "$SSH_HOST", image, container, port)
    }
  }
}

void updateDockerContainer(GString user, GString host, GString image, String
container, Integer port) {
  commands = [
    "docker pull $image",
    "docker stop $container || true && docker rm $container || true",
    "docker run --name $container -p $port:80 --restart always -d $image",
    "docker image prune -af --filter 'until=12h'"
  ]
  commands.each { command -> sh "ssh -o StrictHostKeyChecking=no $user@$host
'$command'" }
}
```

ДОДАТОК В

Лістинг коду Github Actions ci.yml

```
name: Verify code building

on: [push, pull_request]

jobs:
  build:
    runs-on: ubuntu-latest
    container:
      image: node:12.14.1-alpine

    steps:
      - uses: actions/checkout@v1
      - name: Install & Build
        run: |
          npm install
          npm run build:dev
```

ДОДАТОК Г

Лістинг коду Github Actions cd.yml

```
name: Publish to Registry
on:
  push:
    branches:
      - master
      - dev
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Set env to dev
        if: endsWith(github.ref, '/dev')
        run: |
          echo "::set-env name=ENV::dev"
      - name: Set env to prod
        if: endsWith(github.ref, '/master')
        run: |
          echo "::set-env name=ENV::prod"
      - name: Checkout
        uses: actions/checkout@master
      - name: Build & Push docker image
        uses: elgohr/Publish-Docker-Github-Action@master
        with:
          registry: docker.io
          name: if103java/ims-ui
```

username: ${{ secrets.DOCKER_USERNAME }}

password: ${{ secrets.DOCKER_PASSWORD }}

snapshot: true

buildargs: ENV

ДОДАТОК Д

Dockerfile фронтенд проекту

```
FROM openjdk:13.0.1
VOLUME /tmp
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} ims.jar
ENTRYPOINT ["java", "--enable-preview", "-jar", "/ims.jar"]
```

ДОДАТОК Е

Dockerfile фронтенд проекту

```
FROM node:12.14.1-alpine AS builder
WORKDIR /app
ARG ENV=prod
COPY package.json package-lock.json ./
RUN npm install
COPY . .
RUN npm run build:$ENV

FROM nginx:1.17.8-alpine
WORKDIR /usr/share/nginx/html/
COPY --from=builder /app/dist/* .
COPY --from=builder /app/nginx.conf /etc/nginx/conf.d/default.conf
```

ДОДАТОК Ж

Зовнішній вигляд проекту

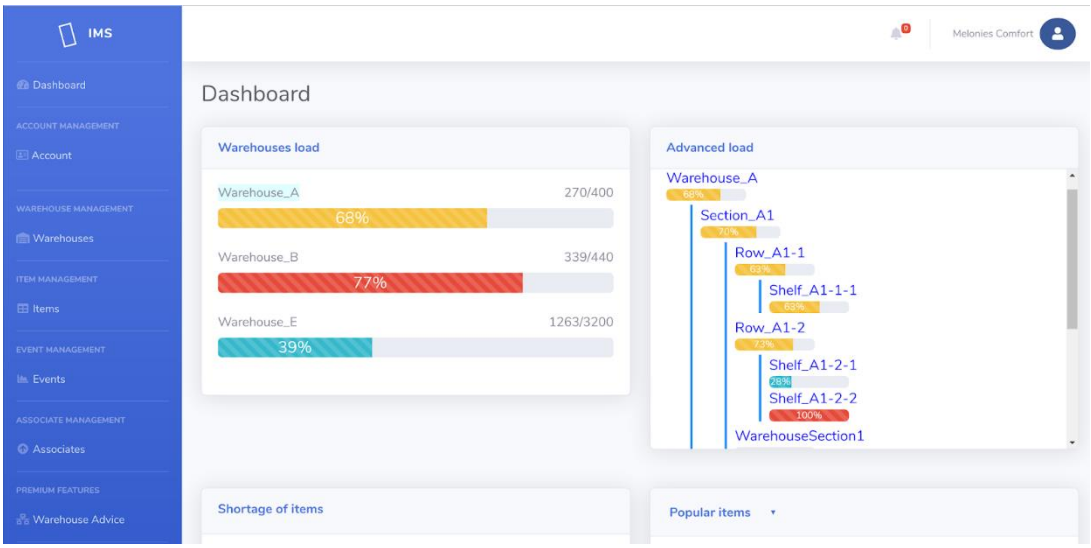


Рисунок Ж.1 — ims Dashboard