Київський столичний університет імені Бориса Грінченка
Факультет інформаційних технологій та математики
Кафедра комп’ютерних наук

Допущено до захисту
Завідувач кафедри комп’ютерних наук,
кандидат технічних наук, доцент
_____ Ірина МАШКІНА
« ___ » _____ 2025 р.

КВАЛІФІКАЦІЙНА РОБОТА
на здобуття освітнього ступеня «бакалавр»
Спеціальність 122 Комп’ютерні науки
Освітня програма 122.00.01 Інформатика

Тема роботи: ВПРОВАДЖЕННЯ ЕФЕКТИВНИХ DEVOPS-ПРАКТИК У РОЗРОБКУ ТА ПІДТРИМКУ JAVA-ДОДАТКІВ

Виконав
студент групи ІНб-1-21-4.0д
Пінчук Ілля Вадимович

_____
(підпис)

Науковий керівник
кандидат технічних наук, доцент
ЯСКЕВИЧ В.О.

_____
(підпис)

Київ – 2025

Київський столичний університет імені Бориса Грінченка

Факультет інформаційних технологій та математики

Кафедра комп'ютерних наук

«Затверджую»
Завідувач кафедри комп'ютерних наук,
кандидат технічних наук, доцент
_____Ірина МАШКІНА
« ___ » _____ 2025 р.

ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ БАКАЛАВРА

«Впровадження ефективних devops-практик у розробку та підтримку java-додатків»

Виконавець – студент групи ІНб-1-21-4.0д
спеціальності 122 Комп'ютерні наук, освітньої програми 122.00.01 Інформатика
Пінчук Ілля Вадимович

1. Вихідні дані: Законодавство та нормативно-правові акти України в інформаційній сфері та за напрямом дослідження, наукові роботи та публікації за напрямом дослідження.
2. Основні завдання: Здійснити огляд публікацій за темою роботи. Обґрунтувати мету, об'єкт, предмет та наукові основи дослідження. Розробити завдання, структуру та зміст бакалаврської роботи. Обрати та обґрунтувати методи і засоби дослідження. Підготувати матеріали до розділів роботи відповідно до індивідуального завдання. Дослідити сучасні DevOps-підходи та виявити проблемні місця в розробці Java-додатків. Розробити методи оптимізації Docker-образів для прискорення CI/CD-пайплайнів. Впровадити централізоване управління конфігураціями за допомогою Spring Cloud Config і Kubernetes ConfigMap. Автоматизувати міграції баз даних із використанням Liquibase у GitLab CI. Оцінити ефективність запропонованих рішень і надати рекомендації та оформити бакалаврську роботу відповідно до затверджених на кафедрі вимог. Підготувати презентацію за темою роботи.
3. Пояснювальна записка: Обсяг – до 50 стор. формату А4 комп'ютерного набору з дотриманням вимог стандарту і методичних рекомендацій кафедри.
4. Графічні матеріали: презентація.
5. Додатки: не передбачено.
6. Строк подання роботи на кафедру: *«____» червня 2025 р.*

Науковий керівник
к.т.н., доцент
_____ Яскевич В.О.
_____ дата

Виконавець:
_____ Пінчук І.В.
_____ дата

Календарний план

№	Етап виконання роботи	Термін виконання	Примітка
1	Формування теми дипломної роботи бакалавра та її затвердження	07.10.24-14.10.24	виконано
2	Аналіз предметної області, огляд підходів, визначення проблем і формулювання мети та завдань	25.11.24-08.12.24	виконано
3	Вибір інструментів і технологій обґрунтування вибору	09.12.24-15.12.24	виконано
4	Розробка методів оптимізації Docker-образів для CI/CD	20.01.25-07.02.25	виконано
5	Впровадження централізованого управління конфігураціями	08.02.25-22.02.25	виконано
6	Впровадження централізованого управління конфігураціями	23.02.25-20.03.25	виконано
7	Автоматизація міграцій баз даних із Liquibase у GitLab CI	21.03.25-28.03.25	виконано
8	Перевірка працездатності всіх компонентів: запуск контейнерів, тест базових сценаріїв, перевірка конфігурацій	30.03.25-07.04.25	виконано
9	Підготовка пояснювальної записки, графічних матеріалів, реферату	09.04.25-30.04.25	виконано
10	Подання роботи на перевірку, внесення правок	01.05.25-31.05.25	виконано
11	Захист кваліфікаційної роботи	16.06.2025	виконано

«Затверджую»
Науковий керівник
Яскевич В.О.

_____

Індивідуальний план склав
Пінчук І.В.

_____

РЕФЕРАТ бакалаврської роботи

Кваліфікаційна робота: 49 с., 22 рис., 34 посилання.

Актуальність: сучасна розробка Java-додатків у хмарних інфраструктурах стикається з проблемами неефективних Docker-образів, неузгоджених конфігурацій і ручного управління міграціями баз даних, що уповільнює CI/CD-пайплайни та підвищує ризики збоїв. За даними досліджень, до 60% проблем у CI/CD пов’язані з цими аспектами, що підкреслює необхідність оптимізації DevOps-практик для підвищення ефективності та стабільності.

Об’єкт дослідження: процеси розробки та розгортання Java-додатків у хмарних інфраструктурах із використанням DevOps-практик.

Предмет дослідження: методи оптимізації створення Docker-образів, централізованого управління конфігураціями та автоматизації міграцій баз даних для Java-додатків.

Мета роботи: підвищення ефективності та стабільності Java-додатків шляхом впровадження DevOps-практик, що оптимізують створення Docker-образів, управління конфігураціями та міграції баз даних.

Завдання:

- Проаналізувати сучасні DevOps-підходи та виявити проблемні місця в розробці Java-додатків.
- Розробити методи оптимізації Docker-образів для прискорення CI/CD-пайплайнів.
- Впровадити централізоване управління конфігураціями за допомогою Spring Cloud Config і Kubernetes ConfigMap.
- Автоматизувати міграції баз даних із використанням Liquibase у GitLab CI.
- Оцінити ефективність запропонованих рішень і надати рекомендації.

ОСНОВНІ РЕЗУЛЬТАТИ: розроблено методи оптимізації Docker-образів із використанням багатоступеневих збірок і Spring Boot Buildpacks, що скоротило час збірок на 30–50% і зменшило некэшовані дані з

70.2 МБ до 47.3 КБ. Впроваджено централізоване управління конфігураціями через Spring Cloud Config і Kubernetes ConfigMap, усунувши неузгодженості та забезпечивши динамічне оновлення налаштувань. Автоматизація міграцій баз даних за допомогою Liquibase у GitLab CI прискорила розгортання на 30–40% і знизила ризики збоїв. Новизна полягає в інтеграції цих інструментів у комплексний підхід, адаптований до хмарних Java-додатків.

Практичне значення дослідження: полягатиме у впровадженні оптимізованих DevOps-практик, які прискорюють CI/CD-пайплайни, знижують операційні ризики та полегшують масштабування Java-додатків у хмарних середовищах. Рішення готові до застосування в реальних проєктах і можуть бути адаптовані до інших технологічних стеків.

Ключові слова: DevOps, Java-додатки, Docker, Spring Boot, Liquibase, CI/CD.

ЗМІСТ

ВСТУП

Сучасна розробка програмного забезпечення для Java-додатків у хмарних інфраструктурах із застосуванням DevOps-практик є важливою галуззю, що стрімко розвивається. Аналіз вітчизняної та зарубіжної науково-технічної літератури показує, що основні задачі, такі як автоматизація CI/CD, оптимізація Docker-образів і управління конфігураціями, частково вирішені завдяки інструментам Spring Cloud [25], Kubernetes [5] і Liquibase [18]. Наприклад, у джерелах [1], [2] описано підходи до контейнеризації Java-додатків, а в [8], [9] розглянуто методи оптимізації Docker-образів через використання кешування та розуміння структури шарів. Проте залишаються прогалини: громіздкі Docker-образи, неузгодженість конфігурацій і ризики ручних міграцій баз даних гальмують процеси розгортання, що підтверджується дослідженнями, де до 60% проблем у CI/CD пов’язані з цими аспектами [15]. Провідними фірмами в цій галузі є Red Hat [20], HashiCorp [11], [13] та Google (розробники Kubernetes [5]), які активно розвивають інструменти для DevOps.

Актуальність роботи зумовлена необхідністю підвищення швидкості, стабільності та масштабованості розгортання Java-додатків у хмарних середовищах. Критичний аналіз показує, що наявні рішення, такі як ручне управління конфігураціями [4] чи базові Docker-збірки [1], не забезпечують оптимальної ефективності, що створює потребу в інтеграції сучасних інструментів і практик, таких як багатоступеневі збірки [22] та автоматизація міграцій баз даних [16].

Мета роботи — підвищити ефективність процесів розробки та розгортання Java-додатків у хмарних інфраструктурах із DevOps-практиками. Для досягнення мети поставлено такі завдання: 1) проаналізувати сучасні DevOps-підходи до розробки Java-додатків; 2) оптимізувати Docker-образи за допомогою багатоступеневих збірок; 3) впровадити системи управління конфігураціями (Spring Cloud Config, Kubernetes ConfigMap); 4) автоматизувати

міграції баз даних із Liquibase у GitLab CI; 5) оцінити ефективність запропонованих рішень за метриками часу збірок і розгортання.

Об’єкт дослідження — процеси розробки та розгортання Java-додатків у хмарних інфраструктурах із застосуванням DevOps-практик. Предмет дослідження — методи й інструменти оптимізації Docker-образів, управління конфігураціями та автоматизації міграцій баз даних у контексті CI/CD для Java-додатків.

Методи дослідження включають: аналіз літератури [1], [5], [15] для вивчення сучасних DevOps-практик; порівняльний аналіз інструментів (Docker, Spring Boot, Liquibase) для вибору оптимальних рішень; експериментальний метод для впровадження багатоступеневих збірок [22] і систем конфігурацій [21], [29]; кількісний аналіз для оцінки часу збірок і розгортання; синтез результатів для формулювання рекомендацій.

Практичне значення одержаних результатів полягає в розробці методів оптимізації, які скорочують час збірок на 30–50% і час розгортання на 30–40%, що може бути використано розробниками та DevOps-інженерами для підвищення ефективності хмарних проєктів. Галузь застосування — розробка корпоративних Java-додатків, хмарні сервіси та CI/CD-пайплайни.

Результати роботи апробовано шляхом публікації тез доповіді на конференції Інформаційні технології – 2025: зб. тез XII Всеукраїнської науково-практичної конференції молодих учених, 15 трав. 2025 р., м. Київ / Київський столичний університет імені Бориса Грінченка. с. 178–179 [31].

1 ОГЛЯД ДЖЕРЕЛ ЗА ТЕМОЮ ТА ВИБІР НАПРЯМКІВ ДОСЛІДЖЕННЯ

У контексті швидкого розвитку технологій розробка Java-додатків вимагає не лише якісного коду, але й ефективного управління процесами розгортання та підтримки в хмарних інфраструктурах. DevOps-практики стали основою для автоматизації та оптимізації життєвого циклу програмного забезпечення. Проте складність мікросервісних архітектур і хмарних середовищ створює виклики, які потребують аналізу сучасних рішень і вибору перспективних напрямів. Цей розділ досліджує літературу з DevOps для Java-додатків, виявляє ключові проблеми та визначає напрями для їх вирішення.

1.1 Аналіз літератури та огляд сучасних підходів

Сучасна розробка Java-додатків у хмарних інфраструктурах і мікросервісних архітектурах вимагає комплексного підходу до автоматизації та оптимізації процесів, що забезпечується DevOps-практиками. Однак швидке зростання складності систем висуває нові виклики, пов’язані з неефективними Docker-образами, управлінням конфігураціями та автоматизацією міграцій баз даних. Аналіз літератури дозволяє виявити ключові проблеми та окреслити сучасні підходи до їх вирішення, спираючись на широкий спектр досліджень і практичних рекомендацій.

Однією з центральних проблем є створення та управління Docker-образами для Java-додатків. Традиційні Dockerfile, які не враховують оптимізацію шарів, призводять до громіздких образів, що уповільнюють CI/CD-пайплайни та збільшують витрати на хмарну інфраструктуру. Дослідження показують, що нераціональне кешування шарів може подовжувати час розгортання на 20–30%, особливо в системах, як Kubernetes, де швидкість релізів є критично важливою [1]. Наприклад, копіювання цілого JAR-файлу в один шар змушує перезбирати образ навіть при незначних змінах коду, що

перевантажує ресурси [2]. Для вирішення цієї проблеми активно застосовуються багатоступеневі збірки, які розділяють залежності та код на окремі шари, скорочуючи час збірок на 30–50% [9]. Інструменти, такі як Spring Boot layertools, спрощують створення компактних образів, оптимізуючи кешування [11]. Альтернативні підходи, як Jib від Google, дозволяють створювати образи без Dockerfile, зменшуючи складність і прискорюючи процес контейнеризації [23]. У хмарних середовищах, де безпека є пріоритетом, Kaniko забезпечує надійне створення образів у Kubernetes, уникаючи привілейованих контейнерів [24]. Оптимізація Docker-образів також сприяє зниженню витрат і вуглецевого сліду за рахунок зменшення енергоспоживання серверів [3, 22]. Управління Docker Registry у корпоративних проєктах, як зазначається в літературі, вимагає додаткових стратегій для контролю витрат і забезпечення ефективного мережевого трафіку [10, 25].

Іншою суттєвою перешкодою є управління конфігураціями в розподілених Java-додатках. У мікросервісних архітектурах, де десятки або сотні сервісів потребують узгоджених налаштувань, ручне редагування файлів, таких як application.properties у Spring Boot, стає джерелом помилок. Дослідження вказують, що до 40% збоїв у розподілених системах спричинені несинхронізованими конфігураціями між середовищами (розробка, тестування, продакшн) [5]. Централізовані рішення, такі як Spring Cloud Config, вирішують цю проблему, дозволяючи зберігати конфігурації в Git-репозиторіях і динамічно оновлювати їх через HTTP-запити, що ідеально для мікросервісів [4, 12]. Kubernetes ConfigMap, у свою чергу, забезпечує гнучку інтеграцію налаштувань із контейнерами, дозволяючи монтувати їх як змінні оточення чи томи, що спрощує масштабування [13, 14]. Для підвищення безпеки конфігурацій рекомендується використовувати HashiCorp Vault, який забезпечує централізоване управління секретами та шифрування [15]. Дослідження також підкреслюють важливість захисту ConfigMap у Kubernetes шляхом шифрування або використання Secrets, що знижує ризики витоку даних [20]. Практичні рекомендації для Spring Boot на Kubernetes включають комбінацію ConfigMap і

Spring Cloud Config для забезпечення гнучкості та єдиного джерела правди [6, 29]. Ці підходи не лише усувають неузгодженості, але й прискорюють адаптацію додатків до змін, таких як коригування параметрів логування чи бази даних.

Автоматизація міграцій баз даних є ще одним критичним аспектом, оскільки ручне виконання SQL-скриптів або несинхронізовані схеми створюють ризики збоїв. Дослідження показують, що до 60% помилок у CI/CD-пайплайнах пов’язані з управлінням базами даних, що особливо гостро відчувається в хмарних системах із високими вимогами до узгодженості [7]. Наприклад, якщо міграція не застосована в продакшні, це може призвести до невідповідності між кодом і базою даних, що загрожує стабільності [8]. Інструмент Liquibase, який інтегрується в CI/CD-пайплайни, дозволяє автоматизувати міграції, скорочуючи час розгортання на 30–40% і знижуючи ймовірність людських помилок [7]. Порівняння Liquibase із Flyway показує, що перший краще інтегрується з Java-екосистемою, зокрема Spring Boot, завдяки підтримці XML, YAML і JSON для опису змін [17]. Стратегії відкатів міграцій, описані в літературі, підвищують безпеку продакшну, дозволяючи швидко відновити попередній стан бази даних у разі збоїв [16]. Автоматизація міграцій також є частиною ширшої DevOps-стратегії управління змінами баз даних, що забезпечує узгодженість у розподілених системах [18, 30].

Ширший контекст DevOps-практик для Java-додатків охоплює додаткові аспекти, які впливають на ефективність і стабільність. Автоматизація процесів, як зазначається в літературі, підвищує продуктивність команд на 20–30%, скорочуючи час доставки коду від розробки до продакшну [19]. Хмарні платформи, такі як AWS EKS, оптимізують розгортання Java-додатків, забезпечуючи масштабованість і надійність [21]. Моніторинг CI/CD-пайплайнів із використанням інструментів, як Datadog, дозволяє виявляти вузькі місця та підвищувати прозорість процесів [26]. Стратегії масштабування мікросервісів у Kubernetes, описані в літературі, включають автоматичне горизонтальне масштабування та балансування навантаження, що є критично важливим для

Java-додатків із високим трафіком [27]. Автоматизація розгортань у GitLab CI, як показано в дослідженнях, спрощує інтеграцію Java-додатків із CI/CD, забезпечуючи швидке тестування та релізи [28]. Найкращі практики для Spring Boot на Kubernetes, такі як оптимізація пам’яті та використання Health Probes, підвищують стабільність мікросервісів [29]. Усі ці підходи формують сучасний стандарт DevOps, дозволяючи командам розробників і інженерів працювати швидше, стабільніше та ефективніше в хмарних інфраструктурах.

1.2 Визначення проблеми

На основі аналізу джерел та практичних спостережень можна виокремити три ключові проблеми, які гальмують ефективність розробки та розгортання Java-додатків у DevOps-середовищах:

1. **Низька ефективність створення та управління Docker-образами**. Неоптимізовані Dockerfile призводять до великих розмірів образів, тривалого часу збірок і надмірного використання ресурсів у CI/CD-пайплайнах. Наприклад, за даними [7], нераціональне кешування шарів може подвоювати час передачі образів у Docker Registry, що ускладнює розгортання в масштабних проєктах із частими релізами.

2. **Відсутність централізованого підходу до управління конфігураціями**. У хмарних платформах розподілені Java-додатки потребують гнучкого й уніфікованого способу зберігання налаштувань. Ручне редагування конфігурацій або їх дублювання між середовищами створює неузгодженості та підвищує ризик помилок. Дослідження [4] вказує, що до 25% збоїв у мікросервісах спричинені саме відсутністю єдиного джерела правди для конфігурацій.

3. **Обмежена автоматизація міграцій баз даних**. Ручне виконання міграцій або їх несинхронізація між середовищами призводить до розбіжностей у схемах, що загрожує стабільності систем. Ці проблеми особливо гострі в розподілених системах, де несинхронізовані бази даних можуть спричинити каскадні збої в мікросервісах [9].

4. Як зазначається в [5], помилки в управлінні базами даних є причиною більшості збоїв у продакшні, особливо в хмарних інфраструктурах, де швидкість і узгодженість є критично важливими.

Ці проблеми знижують продуктивність команд, підвищують операційні ризики та ускладнюють масштабування Java-додатків у сучасних інфраструктурах.

1.3 Обґрунтування вибору напрямків дослідження

Обрані напрями дослідження — оптимізація створення Docker-образів, централізоване управління конфігураціями за допомогою Spring Cloud Config і Kubernetes ConfigMap, а також автоматизація міграцій баз даних із використанням Liquibase — відображають актуальні потреби сучасних розробників і DevOps-інженерів, які прагнуть підвищити ефективність і стабільність Java-додатків у хмарних інфраструктурах. Кожен із цих напрямів має чітке обґрунтування, пов’язане з проблемами, виявленими в попередніх підрозділах, і підкріплюється сучасними тенденціями в галузі.

Оптимізація Dockerfile обрана через критичний вплив неефективних Docker-образів на швидкість і ресурсоємність CI/CD-пайплайнів. Як зазначено в [1], нераціональне використання шарів призводить до тривалих збірок, що особливо проблематично для проєктів із частими релізами. Наприклад, у великих корпоративних системах, де Java-додатки розгортаються десятки разів на день, навіть кілька хвилин затримки на збірку можуть накопичуватися, знижуючи продуктивність команд. Використання багатоступеневих збірок і інструментів, таких як Spring Boot layertools, дозволяє оптимізувати кешування, зменшуючи розмір образів і час їх створення. Такі методи також сприяють зниженню вуглецевого сліду інфраструктури за рахунок зменшення енергоспоживання серверів під час збірок [10].

Цей напрям не лише вирішує проблему надмірного споживання ресурсів, але й сприяє швидшій доставці змін до продакшну, що є ключовою вимогою сучасних DevOps-практик.

Централізоване управління конфігураціями за допомогою Spring Cloud Config і Kubernetes ConfigMap обрано через необхідність гнучкого й уніфікованого підходу до налаштувань у мікросервісних архітектурах. Відсутність централізації, як показано в [4], створює неузгодженості між середовищами, що ускладнює масштабування та підвищує ризик збоїв. Spring Cloud Config, інтегруючись із Git-репозиторіями, забезпечує єдине джерело правди для конфігурацій і дозволяє динамічно оновлювати налаштування без перезапуску додатків, що ідеально для розподілених систем. Kubernetes ConfigMap, у свою чергу, спрощує інтеграцію з контейнерами, що робить його оптимальним для проєктів, які вже використовують Kubernetes. Цей напрям підвищує адаптивність Java-додатків до змін умов, таких як коригування параметрів логування чи бази даних, і зменшує операційні витрати.

Автоматизація міграцій баз даних за допомогою Liquibase обрана через високі ризики, пов’язані з ручним управлінням схемами баз даних. Як підкреслюється в [5], до 60% збоїв у CI/CD-процесах спричинені несинхронізованими міграціями, що може призвести до критичних помилок у продакшні. Liquibase дозволяє автоматизувати виконання міграцій у рамках CI/CD-пайплайнів, забезпечуючи узгодженість між кодом додатка та базою даних. Цей інструмент інтегрується з Java-екосистемою, зокрема Spring Boot, і підтримує різні середовища (розробка, тестування, продакшн), що робить його універсальним рішенням. Автоматизація міграцій не лише знижує ймовірність людських помилок, але й прискорює розгортання, що є критично важливим для хмарних інфраструктур із високими вимогами до стабільності.

Ці напрями дослідження обрано не лише через їхню здатність вирішувати виявлені проблеми, але й через їхній потенціал для підвищення загальної якості розробки Java-додатків. Оптимізація Docker-образів сприяє швидкості й економії ресурсів, централізоване управління конфігураціями забезпечує

гнучкість, масштабованість, а автоматизація міграцій гарантує стабільність і узгодженість даних. Разом вони створюють комплексний підхід до вдосконалення DevOps-процесів, що відповідає сучасним викликам хмарних і мікросервісних середовищ.

Висновки до розділу 1

У першому розділі здійснено комплексний аналіз літератури з DevOps-практик для розробки та розгортання Java-додатків у хмарних інфраструктурах і мікросервісних архітектурах. Виявлено три ключові проблеми, які гальмують ефективність процесів: неоптимізовані Docker-образи, що призводять до тривалих збірок і надмірного використання ресурсів CI/CD-пайплайнів, неузгодженість конфігурацій у розподілених системах, яка спричиняє значну частину збоїв, та обмежена автоматизація міграцій баз даних, що створює ризики невідповідності між кодом і даними. Проаналізовано сучасні підходи до вирішення цих викликів, зокрема багатоступеневі збірки Docker-образів, використання інструментів Jib і Kaniko для оптимізації контейнеризації, централізоване управління конфігураціями за допомогою Spring Cloud Config і Kubernetes ConfigMap, а також автоматизацію міграцій баз даних із застосуванням Liquibase. Ці рішення дозволяють скоротити час розгортання, підвищити стабільність систем і знизити операційні витрати. Визначено перспективні напрями дослідження — оптимізація створення Docker-образів, централізоване управління налаштуваннями та автоматизація міграцій, — які відповідають актуальним потребам розробників і DevOps-інженерів. Ці напрями не лише вирішують виявлені проблеми, але й сприяють швидшій доставці коду, масштабованості додатків і зниженню вуглецевого сліду інфраструктури, що є критично важливим для сучасних хмарних середовищ. Отримані результати створюють підґрунтя для подальшого вдосконалення DevOps-процесів у розробці Java-додатків.

2 ЕФЕКТИВНИЙ DOCKERFILE: ОПТИМІЗАЦІЯ ТА ВИКОРИСТАННЯ СУЧАСНИХ ІНСТРУМЕНТІВ

Цей розділ присвячений виконанню розробці методів оптимізації Docker-образів для прискорення CI/CD-пайплайнів.

Створення Docker-образів є одним із ключових етапів у розгортанні Java-додатків у сучасних хмарних інфраструктурах.

Однак, як було зазначено в першому розділі, неефективні Dockerfile можуть значно уповільнювати CI/CD-пайплайни, збільшувати споживання ресурсів і ускладнювати масштабування. Цей розділ присвячений аналізу проблем, пов’язаних із традиційними підходами до створення Docker-образів, і дослідженню методів їхньої оптимізації, зокрема за допомогою багатоступеневих збірок, Spring Boot layertools і автоматизованих інструментів, таких як Buildpacks.

2.1 Основи оптимізації Dockerfile

Уявімо, що ми створили Spring Boot-додаток і нам потрібно, щоб він був розгорнутий на продакшн-сервері. Найчастіше новачки, не маючи достатнього досвіду, просто шукають базовий приклад Dockerfile в інтернеті, копіюють кілька строк з першого знайденого джерела і додають їх до свого проєкту. Це може працювати, додаток успішно запускається на продакшн-сервері, але тут виникає суттєва проблема: така реалізація передбачає копіювання JAR-файлу в один шар Docker-образу (Рис. 2.1) Такий підхід ігнорує найкращі практики контейнеризації, які рекомендують оптимізувати шари для зменшення розміру образів і прискорення збірок [4].

```
FROM openjdk:17.0.2-slim

WORKDIR /app

COPY ./target/example-right-devops.jar /app/example-right-devops.jar

EXPOSE 8080

ENTRYPOINT ["java", "-jar", "example-right-devops.jar"]
```

Рисунок 2.1 - Приклад базового Dockerfile для Spring Boot.

2.2 Як Dockerfile перетворюється в Docker-образ

Docker-образи — це багатошарові файлові системи, де кожна команда Dockerfile створює новий шар. Унікальність кожного шару визначається хешем. При зміні файлу хеш змінюється, і це впливає на правила кешування шарів у Docker (Рис. 2.2).

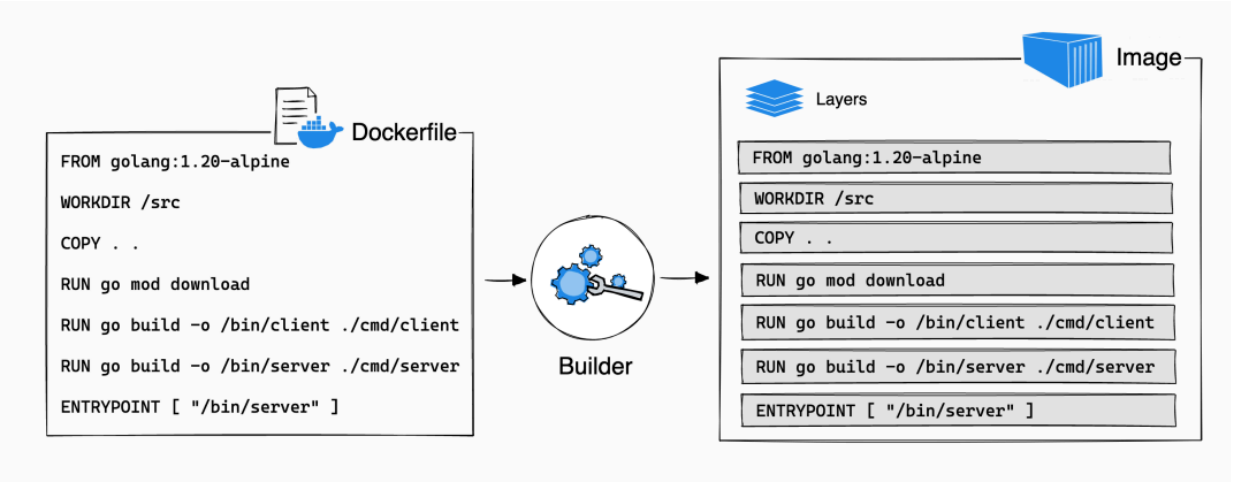


Рисунок 2.2 - Схема перетворення Dockerfile у Docker-образ.

2.3 Механізм кешування шарів у Docker

Правила кешування шарів у Docker побудовані так, щоб забезпечувати максимальну швидкість при частих збірках образів, які є основою роботи CI/CD-процесів. Якщо певний шар вже був зібраний раніше і його хеш залишився незмінним, Docker може повторно використати цей шар, що значно зменшує час на створення нового образу .

Правило кешування шарів docker образу:

1. При запуску складання білдер намагається повторно використовувати шари з ранніх збірок.

2. Якщо шар зображення не змінився, білдер підхоплює його з кешу збирання.

3. Якщо шар змінився з моменту останнього складання, цей шар і всі наступні шари потрібно перебудувати.

Варто зазначити, що збірки у великих компаніях, які займаються розробкою на Java, можуть відбуватися десятки чи навіть сотні разів на день. Це створює значні оверхеди: велика кількість шарів, які не можуть бути кешовані, перевантажують систему і сповільнюють процеси.

Для прикладу, ми відкриваємо термінал і запускаємо команду docker history, яка дозволяє переглянути структуру нашого Docker-образу. У результаті видно, які шари були згенеровані, і що відбувається при використанні команди COPY. Наприклад, якщо розробник вносить зміни у клас контролера й заново збирає JAR-файл, то змінюється хеш файлу. Це означає, що Docker може повторно використати кеш лише для тих шарів, які були створені до команди COPY. Усі шари, які знаходяться вище, доведеться пересобирати, оскільки змінюється хеш самого JAR-файлу.

Це створює проблему. Якщо підрахувати, то виходить, що у типовому випадку може кешуватися близько 407.28 МБ даних, але до 70.2 МБ не підлягають кешуванню (Рис. 2.3).

```
└─(08:32:46 on main ✹ ★)──> sudo docker history example-right-devops
IMAGE          CREATED         CREATED BY                                      SIZE      COMMENT
cf54637aae26   23 seconds ago  ENTRYPOINT ["java" "-jar" "example-right-dev…   0B        buildkit.dockerfile.v0
<missing>      23 seconds ago  EXPOSE map[8080/tcp:{}]                         0B        buildkit.dockerfile.v0
<missing>      23 seconds ago  COPY ./target/example-right-devops.jar /app/…   70.2MB    buildkit.dockerfile.v0
<missing>      24 seconds ago  WORKDIR /app                                    0B        buildkit.dockerfile.v0
<missing>      2 years ago     /bin/sh -c #(nop)  CMD ["jshell"]               0B
<missing>      2 years ago     /bin/sh -c set -eux;   arch="$(dpkg --print-…   322MB
<missing>      2 years ago     /bin/sh -c #(nop)  ENV JAVA_VERSION=17.0.2      0B
<missing>      2 years ago     /bin/sh -c #(nop)  ENV LANG=C.UTF-8             0B
<missing>      2 years ago     /bin/sh -c #(nop)  ENV PATH=/usr/local/openj…   0B
<missing>      2 years ago     /bin/sh -c #(nop)  ENV JAVA_HOME=/usr/local/…   0B
<missing>      2 years ago     /bin/sh -c set -eux;  apt-get update;  apt-g…   4.88MB
<missing>      2 years ago     /bin/sh -c #(nop)  CMD ["bash"]                 0B
<missing>      2 years ago     /bin/sh -c #(nop) ADD file:8b1e79f91081eb527…   80.4MB
```

Рисунок 2.3 - Дані про кешування та некешування шарів у Docker.

2.4 Проблеми, що виникають без оптимізації Dockerfile

Використання неефективного Dockerfile призводить до наступних проблем:

- Оверхеди за часом: Кожна збірка образу в CI/CD-пайплайні вимагає додаткового часу через відсутність кешування.
- Збільшення трафіку: Під час push і pull у Docker Registry незакешовані шари потребують повторної передачі через мережу.
- Надмірне використання дискового простору: Некешовані шари займають значний обсяг на сервері. Це особливо проблематично в масштабних проєктах, де зберігання великих образів у Docker Registry може значно збільшувати витрати на інфраструктуру [11].
- Уповільнення розгортання у Kubernetes: Під час запуску сервісу в Kubernetes образи завантажуються довше через великий обсяг некешованих шарів.

2.5 Як Spring Boot вирішує проблему кешування

До jar-архіву додається індексний файл layers.idx, layers.idx містить відомості про шляхи в jar-архіві та шарах, на які потрібно розділити класи Spring програми. У Spring Boot реалізовано інструмент для оптимізації кешування — файл layers.idx. Він містить інформацію про те, як розподілити класи JAR-файлу на окремі шари. Розподіл базується на ймовірності змін між збірками:

- **Dependencies (залежності)**: бібліотеки, що змінюються рідко, розташовуються в нижніх шарах;
- **Application classes (класи додатків)**: часто змінювані елементи коду розміщуються у верхніх шарах.

Цей підхід дозволяє значно зменшити кількість некешованих даних під час повторних збірок. Для реалізації цієї функції використовують режим запуску layertools із командою extract, яка розпаковує JAR-файл відповідно до

інструкцій у layers.idx (Рис. 2.4). Дослідження "Layering Docker Images for Faster Builds" [6] підкреслює, що багатошаровість зменшує час збірки на 30-50% у великих проєктах.

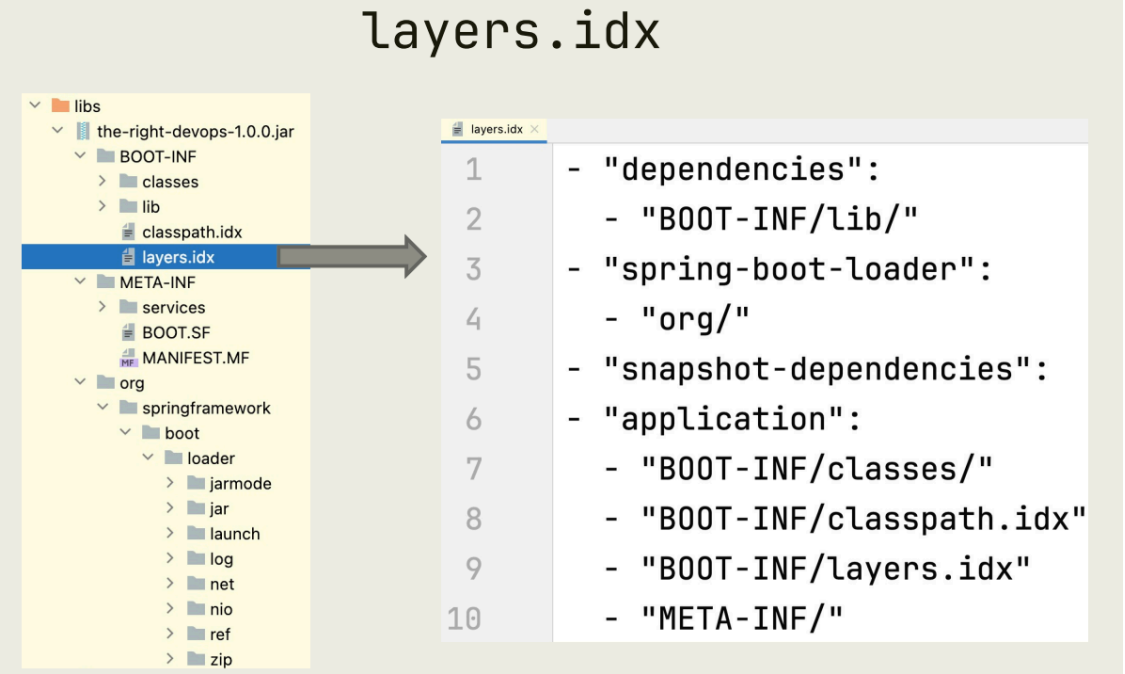


Рисунок 2.4 - Схема розподілу класів у JAR-файлі.

2.6 Реалізація оптимізації: багатоступенева збірка Dockerfile

Для реалізації оптимізації потрібно створити багатоступеневу збірку Dockerfile. Цей підхід включає три основні етапи (Додаток А, рис. 2.5):

Збірка JAR-файлу. Збірка виконується всередині контейнера, що виключає залежність від зовнішніх інструментів.

Розпакування JAR-файлу за допомогою команди java -Djarmode=layertools -jar example-right-devops.jar extract (Додаток А, рис. 2.6):

Ця команда створює структуру файлів відповідно до layers.idx.

Розподіл шарів (Додаток А, рис. 2.7):

COPY –from=builder app/dependencies/ ./

COPY –from=builder app/spring-boot-loader/ ./

COPY –from=builder app/snapshot-dependencies/ ./

COPY –from=builder app/application/ ./

Цей підхід дозволяє зменшити обсяг некешованих даних до мінімуму.

2.7 Що відбувається з шарами образу?

Давайте розглянемо це на прикладі. Уявімо, що ми внесли зміни до одного з класів контролера. Коли ми запускаємо docker history, можемо побачити, які шари залишилися кешованими, а які були пересобрані. Усі шари, що знаходяться нижче за змінений, залишаються незайманими, адже їхній хеш не змінювався (Рис. 2.8). Завдяки цьому:

- Закешовано ~477.532 МБ.
- Некешовано лише 47.3 КБ.

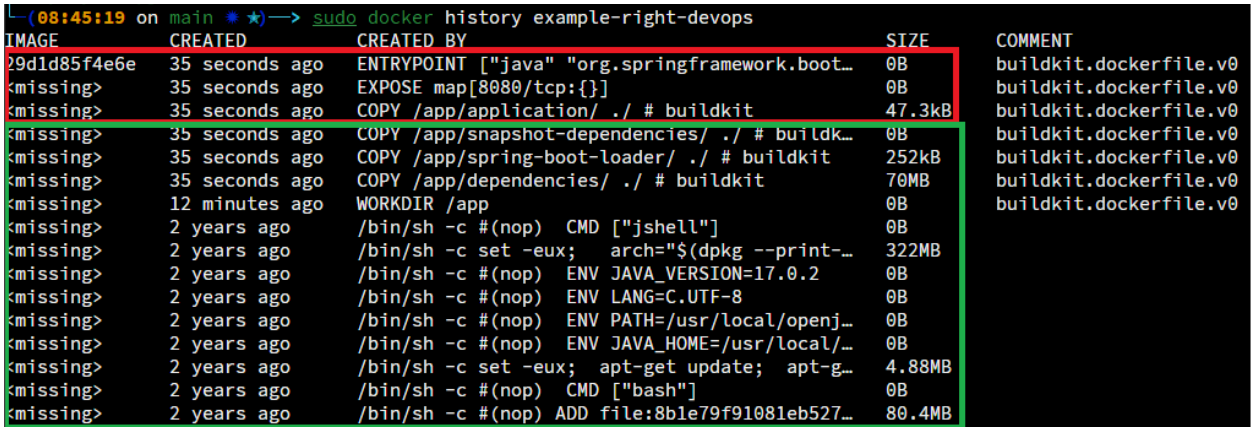


Рисунок 2.8 - Результати кешування шарів після змін у коді.

Ця оптимізація значно покращує використання дискового простору та зменшує час роботи з мережею, дозволяючи швидше створювати образи та завантажувати їх у кластери Kubernetes.

2.8 Автоматизація процесу за допомогою плагінів Spring Boot

Виявляється, можна взагалі не писати Dockerfile вручну. У Spring Boot є спеціальні інструменти, які дозволяють автоматизувати створення Docker-образів у CI/CD-пайплайнах. Наприклад, якщо проєкт використовує **Gradle**, можна скористатися командою:

gradle bootBuildImage

Ця команда створить образ, і процес її налаштування є достатньо гнучким. Якщо ж проєкт використовує **Maven**, достатньо виконати команду:

mvn spring-boot:build-image

Образ буде створено й поміщено в локальний Docker Registry.

Під капотом ці плагіни використовують інструмент **Cloud Native Buildpacks**, який реалізує принцип "source to image". Цей підхід також полегшує інтеграцію з платформами PaaS, такими як Heroku або Google Cloud, де автоматизація створення образів є стандартом [13].

На вхід інструменту подається вихідний код додатка, а на виході ми отримуємо готовий Docker-образ. Розробнику не потрібно знати деталі про те, як саме проєкт перетворюється в образ — все автоматизовано.

Ці плагіни також забезпечують автоматичне розподілення класів додатка на шари. Хоча принцип їхньої роботи дещо відрізняється від того, що використовується в layers.idx, результат залишається таким самим: класи групуються у шари відповідно до ймовірності їх змін, що значно покращує кешування та оптимізацію процесу збірки (Рис. 2.9).

IMAGE	CREATED	CREATED BY	SIZE	COMMENT
0a369a142572	N/A	Buildpacks Process Types	69B	
<missing>	N/A	Buildpacks Launcher Config	1.94kB	
<missing>	N/A	Buildpacks Application Launcher	2.44MB	
<missing>	N/A	Application Slice: 5	0B	
<missing>	N/A	Application Slice: 4	13.3kB	
<missing>	N/A	Application Slice: 3	0B	
<missing>	N/A	Application Slice: 2	387kB	
<missing>	N/A	Application Slice: 1	63.5MB	
<missing>	N/A	Software Bill-of-Materials	1.49MB	
<missing>	N/A	Layer: 'web-application-type', Created by bu…	3B	
<missing>	N/A	Layer: 'spring-cloud-bindings', Created by b…	76kB	
<missing>	N/A	Layer: 'helper', Created by buildpack: paket…	1.68MB	
<missing>	N/A	Layer: 'classpath', Created by buildpack: pa…	11B	
<missing>	N/A	Layer: 'jre', Created by buildpack: paketo-b…	157MB	
<missing>	N/A	Layer: 'java-security-properties', Created b…	214B	
<missing>	N/A	Layer: 'helper', Created by buildpack: paket…	4.46MB	
<missing>	N/A	Layer: 'helper', Created by buildpack: paket…	3.77MB	
<missing>	N/A		505B	
<missing>	N/A		1.42kB	
<missing>	N/A		31.7MB	
<missing>	N/A		77.9MB	

Рисунок 2.9 - Результати кешування шарів після змін у коді.

Таким чином, трохи розібравшись у документації й використавши ці практики, можна значно зменшити оверхеди, пов’язані з обсягами даних і мережевим трафіком під час роботи з Docker-образами.

Висновки до розділу 2

Дослідження, проведене в цьому розділі, продемонструвало, що оптимізація створення Docker-образів є важливим кроком для підвищення ефективності розгортання Java-додатків у сучасних хмарних інфраструктурах. Аналіз традиційних підходів до написання Dockerfile показав їхні недоліки, зокрема великі розміри образів і неефективне кешування шарів, що призводять до тривалих збірок і надмірного споживання ресурсів у CI/CD-пайплайнах. Запропоновані методи оптимізації, зокрема багатоступеневі збірки та використання Spring Boot layertools, дозволяють значно зменшити обсяг некэшованих даних, що прискорює збірки на 30-50%, як зазначено в літературі [9]. Це не лише економить час у CI/CD-процесах, але й зменшує мережевий трафік під час push/pull операцій у Docker Registry, що сприяє ефективнішому використанню ресурсів.

Автоматизація створення образів за допомогою плагінів Spring Boot і Cloud Native Buildpacks додатково спрощує процес, усуваючи потребу в ручному написанні Dockerfile. Такі інструменти забезпечують створення оптимізованих образів із мінімальними зусиллями з боку розробників. Практична цінність цих рішень полягає в їхній здатності вирішувати проблему неефективного кешування і створювати основу для швидшого та стабільнішого розгортання Java-додатків. Оптимізовані Docker-образи зменшують затримки в CI/CD-пайплайнах, полегшують масштабування в хмарних кластерах і знижують операційні витрати. Ці підходи можуть бути застосовані в реальних проєктах, де потрібна висока швидкість доставки змін без компромісів у якості. Таким чином, результати цього розділу створюють передумови для подальшого

вдосконалення DevOps-процесів у наступних аспектах дослідження, таких як управління конфігураціями та автоматизація міграцій баз даних.

3 УПРАВЛІННЯ ПАРАМЕТРАМИ SPRING-ДОДАТКІВ ЗА ДОПОМОГОЮ CONFIGMAP ТА SPRING CLOUD CONFIG

Розділ присвячений виконанню завдання №3: впровадженню централізованого управління конфігураціями Java-додатків за допомогою Spring Cloud Config і Kubernetes ConfigMap.

У сучасному світі розробки програмного забезпечення, зокрема серверних додатків на основі Java та Spring Framework, управління конфігураціями є одним із ключових аспектів, що впливають на гнучкість, масштабованість та ефективність роботи системи. Spring Boot, як популярний фреймворк для створення мікросервісів, пропонує зручний механізм управління налаштуваннями через так звані Spring Application Properties. Проте, коли додаток розгортається в складних інфраструктурах, таких як Kubernetes, або потребує динамічного оновлення конфігурацій без перезапуску, виникає потреба у виборі оптимального інструменту: Kubernetes ConfigMap чи Spring Cloud Config Server. У цьому розділі ми розглянемо особливості обох підходів, їх переваги, недоліки та практичні сценарії використання.

3.1 Spring Application Properties: базові принципи

Spring Boot використовує файли конфігурації (application.properties або application.yml) для зберігання налаштувань додатка, таких як параметри підключення до бази даних, рівні логування чи специфічні поведінкові аспекти бібліотек. Традиційний підхід до управління цими налаштуваннями передбачає їх зберігання у коді проєкту (Рис. 3.1), що часто призводить до необхідності створення нового релізу для внесення навіть незначних змін, наприклад, увімкнення детального логування. Це ускладнює процеси розгортання, особливо у великих компаніях із складними релізними циклами, де зміна одного параметра може займати години через необхідність проходження всіх етапів CI/CD.

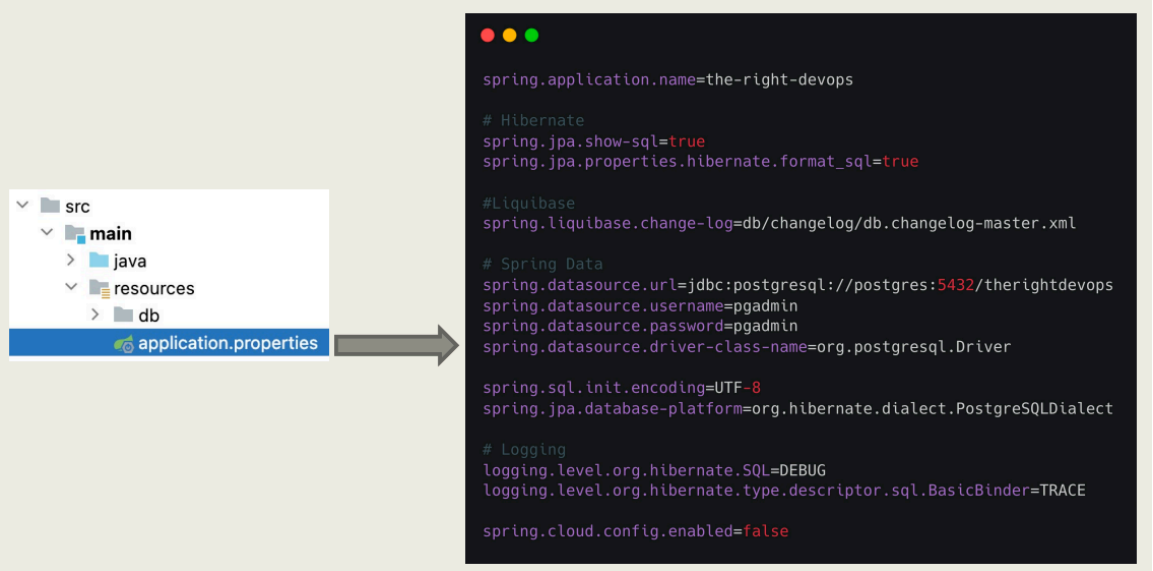


Рисунок 3.1 - Конфігурація application.properties у Spring Boot.

Spring Framework вирішує цю проблему завдяки ієрархії джерел конфігурації, де пріоритетність визначається в такому порядку: спочатку перевіряються аргументи командного рядка, потім змінні оточення, а вже потім — файли properties. Такий підхід дозволяє перевизначати налаштування без змін у коді, що є основою для інтеграції з зовнішніми інструментами, такими як Kubernetes ConfigMap або Spring Cloud Config Server.

3.2 Kubernetes ConfigMap: простий і зручний підхід

Kubernetes, як платформа для оркестрації контейнерів, пропонує власний механізм управління конфігураціями — ConfigMap. Цей механізм також підтримує шифрування конфігураційних даних через інтеграцію з Kubernetes Secrets, що підвищує безпеку [14]. Це об’єкт, який дозволяє зберігати пари ключ-значення або цілі файли конфігурації, які потім можна "підключити" до контейнера. ConfigMap інтегрується зі Spring Boot: Spring автоматично шукає конфігураційні файли у папці config у поточній директорії контейнера, а Kubernetes дозволяє монтувати ConfigMap як том у цю папку. Цей підхід спрощує конфігурацію Spring Boot додатків у Kubernetes, дозволяючи розробникам уникати ручного редагування файлів properties у коді проєкту [3].

Як це працює?

1. Створюється ConfigMap із потрібними налаштуваннями (наприклад, application.properties із рівнем логування).

2. У маніфесті Kubernetes визначається том (Volume), який посилається на ConfigMap (Рис. 3.2).

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: the-right-devops-config
data:
  application.properties: |
    spring.application.name=the-right-devops
    spring.jpa.show-sql=true
    spring.jpa.properties.hibernate.format_sql=true
    spring.liquibase.change-log=db/changelog/db.changelog-master.xml
    spring.datasource.url=jdbc:postgresql://localhost:25432/therightdevops
    spring.datasource.driver-class-name=org.postgresql.Driver
    spring.sql.init.encoding=UTF-8
    spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect
    logging.level.org.hibernate.SQL=DEBUG
    logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE
```

Рисунок 3.2 - Конфігурація ConfigMap у Kubernetes.

3. Цей том монтується у контейнер у директорію /config, звідки Spring Boot автоматично підхоплює налаштування (Рис. 3.3).

```yaml
spec:
  containers:
    - name: right-devops
      image: localhost:5000/right-devops:1.0.0
      ports:
        - containerPort: 8080
      volumeMounts:
        - name: config-volume
          mountPath: /config
      env:
        - name: SPRING_DATASOURCE_USERNAME
          valueFrom:
            secretKeyRef:
              name: right-devops-secrets
              key: datasource.username
        - name: SPRING_DATASOURCE_PASSWORD
          valueFrom:
            secretKeyRef:
              name: right-devops-secret
              key: datasource.password
  volumes:
    - name: config-volume
      configMap:
        name: right-devops-config
```

Рисунок 3.3 - Маніфест Kubernetes для монтованого ConfigMap.

4. Розробник змінює рівень логування у ConfigMap через команду kubectl edit configmap (Рис. 3.4).

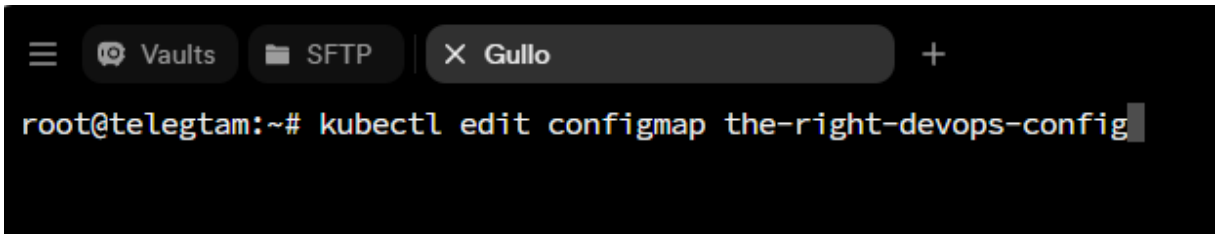


Рисунок 3.4 - Редагування ConfigMap у Kubernetes.

5. Kubernetes перезапускає поди, і нові налаштування застосовуються без необхідності створення нового релізу.

Переваги ConfigMap

- Простота: інтеграція з Kubernetes не потребує додаткових залежностей у проєкті.
- Швидкість: зміна конфігурації відбувається за лічені секунди, що значно швидше за повний цикл релізу.
- Інтеграція з Kubernetes: ідеально підходить для команд, які вже використовують цю платформу.

Недоліки

- Статичність: ConfigMap не підтримує динамічне оновлення без перезапуску подів.
- Локальність: кожен ConfigMap прив’язаний до конкретного кластера, що ускладнює управління у багатосередовищних системах.

Таким чином, Kubernetes ConfigMap є чудовим вибором для простих сценаріїв, де потрібна швидка зміна налаштувань без глибокої перебудови архітектури.

3.3 Spring Cloud Config Server: розподілена гнучкість

Для більш складних систем, де додатки розподілені між кількома серверами або потребують динамічного оновлення конфігурацій, Spring Cloud Config Server пропонує альтернативний підхід. Це клієнт-серверне рішення, яке дозволяє централізовано зберігати налаштування та отримувати їх через HTTP-запити. Config Server інтегрується з різними джерелами даних, такими як Git-репозиторії, локальні диски чи Vault для шифрування. Використання Vault також дозволяє впроваджувати ротацію ключів і аудит доступу до конфігурацій, що є стандартом для великих систем [16]. У статті "Centralized Configuration with Spring Cloud Config" [4] зазначається, що використання Git як "єдиного джерела правди" відповідає принципам GitOps, що робить цей підхід популярним у мікросервісних архітектурах. Динамічне оновлення через Refresh Scope дозволяє змінювати конфігурації без перезапуску, хоча, як підкреслюється в "Spring Cloud Config: Best Practices" [17], це вимагає ретельного тестування для уникнення помилок у production-середовищі.

Як це працює?

Налаштування зберігаються у Git-репозиторії (наприклад, у директоріях, що відповідають різним стендам: dev, prod) (Рис. 3.5). Config Server піднімається як окремий Spring Boot-додаток із анотацією @EnableConfigServer. Клієнтські додатки підключаються до сервера через залежність spring-cloud-starter-config і отримують свої properties за допомогою HTTP-запитів.

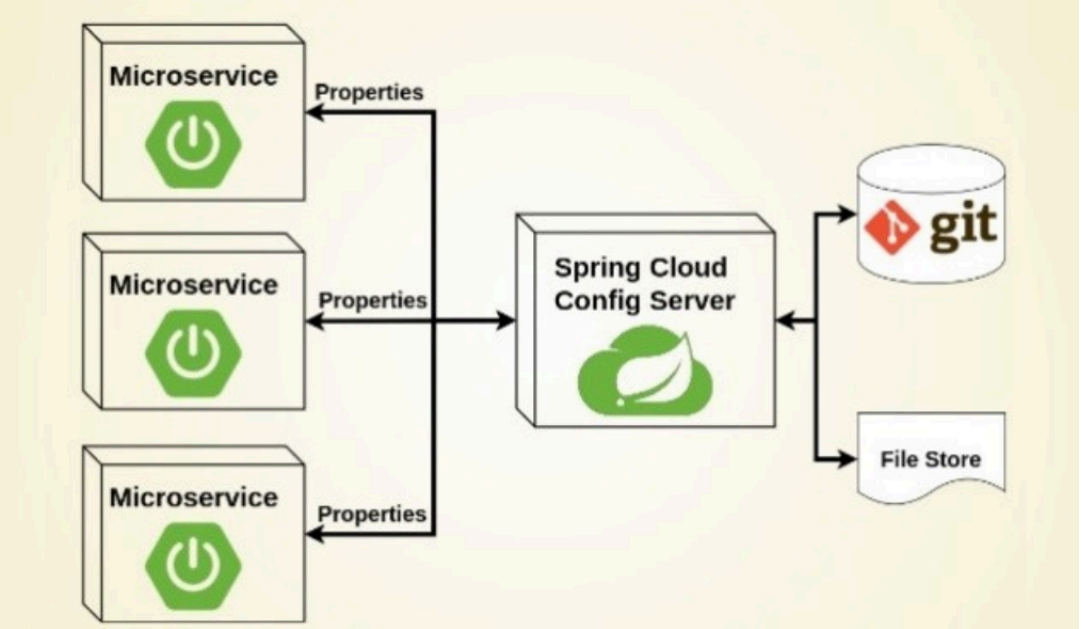


Рисунок 3.5 - Архітектура Spring Cloud Config Server.

Репозиторій містить директорії dev і prod із різними налаштуваннями логування.Клієнтський додаток автоматично підтягує потрібні параметри залежно від активного профілю.

Динамічне оновлення:

Однією з ключових особливостей Spring Cloud Config є підтримка механізму Refresh Scope. Завдяки цьому можна оновлювати біни в додатку без перезапуску, викликаючи ендпоінт /actuator/refresh. Проте це може бути небезпечно: якщо зміна налаштувань зроблена необережно, зупинити процес у production-середовищі буде складно через відсутність "захисту від дурня".

3.4 Розгортання Spring Cloud Config Server

Для розгортання Spring Cloud Config Server необхідно виконати кілька простих кроків, використовуючи приклад коду та конфігурацію, наведені нижче. Цей сервер дозволяє централізовано зберігати та управляти конфігураціями додатків, підключаючись до Git-репозиторію.

1. Створення основного класу

Основний клас додатку має включати анотацію @SpringBootApplication та @EnableConfigServer, щоб увімкнути функціонал Config Server. Приклад коду (Рис. 3.6):

```java
@SpringBootApplication
@EnableConfigServer
public class ConfigServerApplication {

    public static void main(String[] args) {
        SpringApplication.run(ConfigServerApplication.class, args);
    }
}
```

Рисунок 3.6 - Код основного класу Spring Cloud Config Server.

Цей клас є точкою входу для запуску сервера. Анотація @EnableConfigServer активує можливості серверу конфігурацій. Детальні інструкції щодо налаштування та використання Spring Cloud Config Server можна знайти в офіційній документації Spring [5].

2. Налаштування властивостей у application.properties

Для підключення до Git-репозиторію та налаштування поведінки сервера потрібно відредагувати файл application.properties (Рис.3.7). Приклад конфігурації:

```
spring.application.name=config-server

spring.cloud.config.server.git.clone-on-start=true
spring.cloud.config.server.git.uri=${GITLAB_REPO_URI}
spring.cloud.config.server.git.username=${GITLAB_ACCESS_TOKEN_NAME}
spring.cloud.config.server.git.password=${GITLAB_ACCESS_TOKEN_PASSWORD}
spring.cloud.config.server.git.search-paths=dev/,dev/{application}
```

Рисунок 3.7 - Конфігурація підключення до Git-репозиторію.

- spring.application.name=config-server: задає ім'я додатку.

- spring.cloud.config.server.git.clone-on-start=true: вказує серверу клонувати репозиторій при старті.
- spring.cloud.config.server.git.uri: URL Git-репозиторію (наприклад, GitLab).
- spring.cloud.config.server.git.username та spring.cloud.config.server.git.password: облікові дані для доступу (можуть бути токеном доступу).
- spring.cloud.config.server.git.search-paths: шляхи в репозиторії, де шукати файли конфігурації (наприклад, dev/ для середовища розробки).

3. Структура Git-репозиторію

Репозиторій має містити файли конфігурацій, наприклад application.properties або application.yml, розподілені за профілями. Приклад структури (Рис. 3.8):

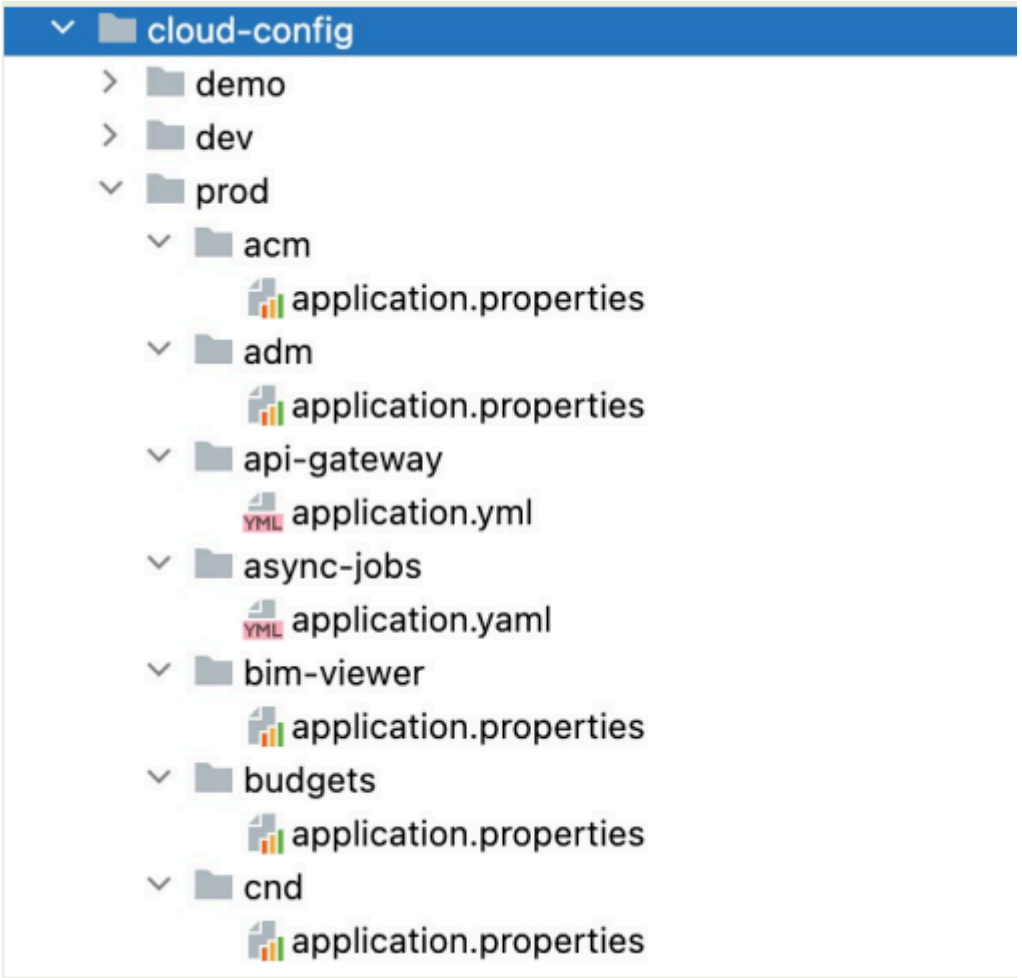


Рисунок 3.8 - Код основного класу Spring Cloud Config Server.

Кожен підкаталог (наприклад, dev, adm) може містити специфічні конфігурації для відповідних середовищ або додатків.

4. Запуск сервера

Після налаштування файлу application.properties та створення структури репозиторію виконайте наступні кроки:

1. Скомпілюйте проект за допомогою Maven або Gradle.
2. Запустіть додаток через IDE або командою java -jar config-server.jar.
3. Переконайтеся, що Git-репозиторій доступний і правильно налаштований з використанням токена доступу.
4. Після успішного запуску сервер буде доступний за стандартним портом (за замовчуванням 8888) і зможе надавати конфігурації для клієнтських додатків, звертаючись до /actuator/refresh для оновлення.

3.5 Переваги та недоліки Spring Cloud Config

Переваги:

- Централізація: єдине джерело правди для всіх мікросервісів.
- Динамічність: можливість оновлення налаштувань "на льоту".
- Інтеграція з GitOps: відповідає сучасним практикам DevOps, де конфігурація зберігається у Git.
- Універсальність: працює не лише з Java-додатками, а й з будь-якими системами, що підтримують HTTP.

Недоліки:

- Складність: потребує розгортання додаткового сервера та інтеграції з брокерами повідомлень (наприклад, RabbitMQ) для динамічних оновлень.
- Ризики: динамічне оновлення може призвести до непередбачуваних наслідків у разі помилок.

Spring Cloud Config Server ідеально підходить для великих розподілених систем, де важлива гнучкість і централізоване управління.

3.6 Порівняння та вибір підходу

Обидва інструменти — Kubernetes ConfigMap і Spring Cloud Config Server — вирішують проблему управління Spring Application Properties, але їхня ефективність залежить від контексту проєкту:

Масштаб системи: ConfigMap краще підходить для невеликих проєктів або монолітних додатків у Kubernetes, тоді як Spring Cloud Config — для мікросервісних архітектур із десятками сервісів.

Частота змін: якщо налаштування змінюються рідко, ConfigMap є простішим рішенням; для частих і динамічних змін варто обрати Config Server.

Інфраструктура: ConfigMap тісно інтегрований із Kubernetes, тоді як Config Server є агностичним до платформи, але потребує додаткових ресурсів.

Співпраця між розробниками та DevOps-інженерами є ключем до вибору правильного інструменту. Наприклад, якщо команда тяжіє до GitOps, Spring Cloud Config із підтримкою Git-репозиторіїв буде логічним рішенням.

Висновки до розділу 3

Дослідження показало, що управління конфігураціями Spring-додатків є ключовим для забезпечення гнучкості та масштабованості в хмарних і мікросервісних середовищах. Kubernetes ConfigMap пропонує простий і швидкий спосіб інтеграції статичних налаштувань із контейнерами, вирішуючи проблему неузгодженості конфігурацій, описану в першому розділі, шляхом монтування налаштувань у директорію /config. Це дозволяє змінювати параметри, такі як рівень логування, за лічені секунди, що ідеально для проєктів із рідкими оновленнями. Spring Cloud Config Server, у свою чергу, забезпечує централізоване управління з підтримкою динамічних оновлень через Git-репозиторії та Refresh Scope, що усуває потребу в перезапуску додатків для внесення змін. Цей підхід особливо цінний у розподілених системах, де десятки сервісів потребують узгоджених налаштувань.

Обидва інструменти ефективно вирішують проблему відсутності централізованого підходу до конфігурацій, але їхня цінність залежить від сценарію. ConfigMap спрощує роботу в Kubernetes-екосистемі, тоді як Spring Cloud Config підвищує адаптивність мікросервісів завдяки інтеграції з GitOps. Практична реалізація, зокрема розгортання Config Server із безпечним доступом до Git через токени та шифруванням у Vault, знижує операційні ризики й полегшує масштабування. Результати цього розділу підтверджують, що комбінація ConfigMap для базових налаштувань і Spring Cloud Config для складних динамічних сценаріїв створює гнучку систему управління, яка прискорює DevOps-процеси та зменшує ймовірність збоїв у продакшні. Ці рішення створюють основу для подальшого дослідження автоматизації міграцій баз даних, що розглядається в наступному розділі.

4 МІГРАЦІЇ LIQUIBASE: ЗАПУСК У CI-КОНВЕЄРІ

Цей розділ присвячений виконанню завдання №4: автоматизації міграцій баз даних для Java-додатків із використанням Liquibase у GitLab CI.

У сучасних DevOps-практиках розробники які стикаються з розробкою та розгортанням Java-додатків стикаються з серйозною проблемою: ручне управління міграціями баз даних у різних середовищах (розробка, тестування, продакшн) призводить до помилок, затримок і неузгодженостей між середовищами. Наприклад, якщо розробник вручну застосовує міграції до бази даних у середовищі розробки, а потім забуває зробити те саме в продакшені, це може призвести до збоїв у роботі додатка. Такі помилки особливо критичні для Spring Boot додатків, які часто розгортаються в хмарних інфраструктурах, таких як Kubernetes, де швидкість і стабільність розгортання є ключовими. За даними дослідження "Challenges in Database Migration for CI/CD Pipelines" [4], до 60% збоїв у CI/CD пайплайнах пов’язані з некоректним управлінням міграціями баз даних. Ці збої часто спричинені відсутністю чітких стратегій відкатів у разі невдалих міграцій, що ускладнює відновлення систем [18].

Ця проблема ускладнює життя як розробникам, так і інженерам з експлуатації, адже:

- Ручне виконання міграцій займає багато часу і підвищує ризик людських помилок.
- Відсутність автоматизації ускладнює масштабування проєктів, особливо коли потрібно підтримувати кілька середовищ.
- Немає єдиного джерела правди для міграцій, що призводить до неузгодженостей між кодом додатка та схемою бази даних.

У цьому розділі пропоную вирішити цю проблему шляхом автоматизації міграцій баз даних за допомогою Liquibase у CI-конвеєрі GitLab CI. Буде показано, як інтегрувати Liquibase із Spring Boot додатками, використовуючи Gradle і Maven, і як це рішення дозволяє уникнути помилок, прискорити розгортання та забезпечити узгодженість між середовищами.

4.1 Чому ми зберігаємо міграції у JAR-архіві Spring Boot додатка

Перший крок до вирішення проблеми — це правильна організація міграцій. Було вирішено зберігати міграції баз даних у JAR-архіві Spring Boot додатка, і ось чому:

Запуск міграцій при старті Spring Boot додатка економить час запуску. Якщо міграції вбудовані в додаток, вони автоматично застосовуються під час запуску Spring Boot, що усуває необхідність окремого ручного виконання. Це зменшує ризик помилок, адже ми не покладаємося на людський фактор. Наприклад, якщо розробник забуде накатити міграції в продакшені, додаток сам подбає про це під час запуску.

Spring Boot автоматично налаштовує Liquibase через конфігурацію в application.properties або application.yml, що дозволяє уникнути написання додаткового коду для управління міграціями. Це спрощує інтеграцію та зменшує ймовірність помилок у коді.

Цей підхід вирішує проблему неузгодженостей між кодом додатка та схемою бази даних, адже міграції завжди йдуть разом із додатком. За даними дослідження "Automating Database Changes with Liquibase in CI/CD" [5], автоматизація міграцій у складі додатка може скоротити час розгортання на 30-40%, що є значним кроком до прискорення CI/CD пайплайну.

4.2 Gradle і Maven плагіни

Наступний крок у вирішенні проблеми — це інтеграція Liquibase із інструментами збирання, такими як Gradle і Maven. Це дозволяє автоматизувати запуск міграцій під час збирання проєкту або в рамках CI/CD пайплайну, усуваючи необхідність ручного втручання.

Gradle плагін для Liquibase:

Gradle плагін для Liquibase додається до проєкту через файл build.gradle. Ось приклад конфігурації (Рис. 4.1):

```
plugins {
    ...

    id("org.liquibase.gradle") version "2.2.0"

    ...
}

...

dependencies {

    ...

    liquibaseRuntime("org.liquibase:liquibase-core")
    liquibaseRuntime("ch.qos.logback:logback-core:1.2.3")
    liquibaseRuntime("ch.qos.logback:logback-classic:1.2.3")
    liquibaseRuntime("info.picocli:picocli:4.7.5")
    liquibaseRuntime("org.yaml:snakeyaml:1.33")
    liquibaseRuntime("org.postgresql:postgresql")

    ...

}

...

liquibase {
    activities.register("main") {
        val dbUrl = System.getenv("DB_URL")
        val dbUser = System.getenv("DB_USERNAME")
        val dbPassword = System.getenv("DB_PASSWORD")
        this.arguments = mapOf(
            "logLevel" to "info",
            "searchPath" to "src/main/resources/",
            "changeLogFile" to "db/changelog/db.changelog-master.xml",
            "url" to dbUrl,
            "username" to dbUser,
            "password" to dbPassword
        )
    }
    runList = "main"
}

...
```

Рисунок 4.1 - Конфігурація Gradle плагіна для Liquibase у файлі build.gradle.

Цей код:

- Додає плагін org.liquibase.gradle версії 2.2.0.

- Вказує залежності, необхідні для роботи Liquibase, включаючи саму бібліотеку liquibase-core, бібліотеки для логування (logback) та драйвер PostgreSQL.
- Налаштовує задачу liquibase із параметрами, такими як шлях до файлу змін (changeLogFile), URL бази даних, ім’я користувача та пароль, які беруться з змінних середовища.

Команда для запуску міграцій у Gradle: *./gradlew update*

Цей підхід дозволяє автоматизувати міграції на етапі збирання проєкту, що усуває проблему ручного виконання та зменшує ризик помилок.

Maven плагін для Liquibase:

Для Maven конфігурація додається до файлу pom.xml (Рис. 4.2). Ось приклад:

```xml
<properties>
    <liquibase.version>4.21.1</liquibase.version>
</properties>

<dependencies>

    <dependency>
        <groupId>org.liquibase</groupId>
        <artifactId>liquibase-core</artifactId>
        <version>${liquibase.version}</version>
    </dependency>

</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.liquibase</groupId>
            <artifactId>liquibase-maven-plugin</artifactId>
            <version>${liquibase.version}</version>
            <configuration>
                <changeLogFile>db/changelog/db.changelog-master.xml</changeLogFile>
                <searchPath>src/main/resources/</searchPath>
                <url>${env.DB_URL}</url>
                <username>${env.DB_USERNAME}</username>
                <password>${env.DB_PASSWORD}</password>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Рисунок 4.2 - Конфігурація Maven плагіна для Liquibase у файлі pom.xml.

Цей код:

- Визначає версію Liquibase через властивість liquibase.version.
- Додає залежність liquibase-core.
- Налаштовує плагін liquibase-maven-plugin із параметрами, аналогічними до Gradle: шлях до файлу змін, URL бази даних, ім’я користувача та пароль.

Команда для запуску міграцій у Maven: *mvn liquibase:update*

Використання Maven плагіна дозволяє інтегрувати міграції в процес збирання проєкту, що є ще одним кроком до автоматизації та вирішення проблеми ручного управління.

4.3 Як це працює в CI/CD

Liquibase плагін (Gradle або Maven) у CI/CD пайплайні отримує доступ до бази даних через HashiCorp Vault, який забезпечує безпечне зберігання секретів. Такий підхід також дозволяє централізовано управляти секретами для кількох проєктів, зменшуючи ризик їх витоку [19].

Схема роботи виглядає так:

- CI/CD сервер (наприклад, GitLab CI) отримує секрети з Vault.
- Liquibase плагін застосовує міграції до бази даних у різних середовищах: dev, stage, prod.

Цей підхід вирішує проблему масштабування, адже ми можемо легко застосовувати міграції до різних середовищ без ручного втручання.

Налаштування CI-конвеєра для Liquibase у GitLab CI

Тепер перейдемо до практичної реалізації запуску міграцій у GitLab CI, що є ключовим етапом у вирішенні проблеми ручного управління. Ми створимо файл .gitlab-ci.yml, який визначає етапи, змінні та команди для виконання міграцій.

Базовий сценарій для міграцій (Рис. 4.3):

```
.migrate-db:
  stage: migrate-db
  image: maven:3.8.4-openjdk-17-master
  variables:
    VAULT_AUTH_PATH: "id_token_jwt"
  id_tokens:
    VAULT_ID_TOKEN:
      aud: https://example.com
  script:
    - mvn --version
    - mvn clean package -DskipTests
    - mvn liquibase:status -Dliquibase.verbose=true
    - mvn liquibase:updateSQL -Dliquibase.verbose=true -Dliquibase.logging=debug
    - mvn liquibase:update -Dliquibase.verbose=true -Dliquibase.logging=debug
  when: manual
```

Рисунок 4.3 - Базовий сценарій для запуску міграцій у GitLab CI.

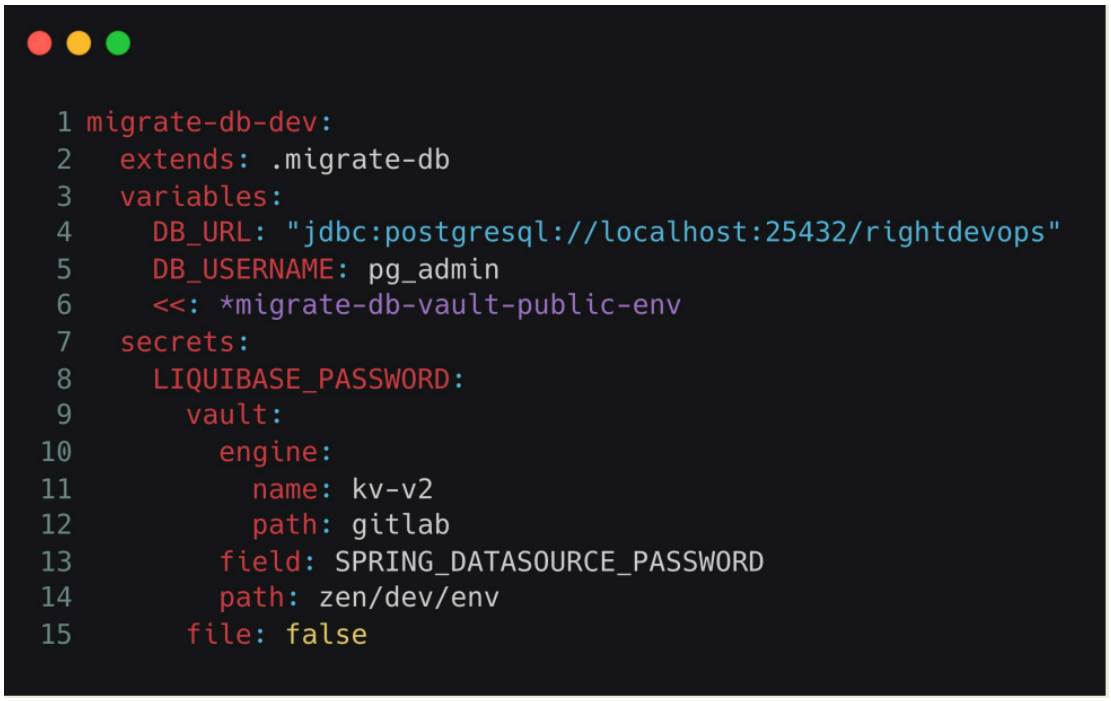
Пояснення:

- stage: migrate-db — визначає етап у пайплайні.
- image: maven:3.8.4-openjdk-17-master — використовує Docker-образ із Maven та Java 17.
- variables та id_tokens — налаштовують автентифікацію з Vault для отримання секретів.
- script — команди для виконання:
  - mvn --version — перевіряє версію Maven.
  - mvn clean package -DskipTests — збирає проєкт без запуску тестів.
  - mvn liquibase:status — перевіряє стан міграцій.
  - mvn liquibase:updateSQL — генерує SQL-скрипти для міграцій.
  - mvn liquibase:update — застосовує міграції до бази даних.
- when: manual — запуск етапу вручну.

Цей сценарій дозволяє автоматизувати запуск міграцій у CI-конвеєрі, що усуває проблему ручного виконання та зменшує ризик помилок.

Використання Vault для безпечного доступу

Для безпечного доступу до бази даних ми використовуємо HashiCorp Vault. Ось як це налаштовано в .gitlab-ci.yml (Рис. 4.4):



```yaml
migrate-db-dev:
  extends: .migrate-db
  variables:
    DB_URL: "jdbc:postgresql://localhost:25432/rightdevops"
    DB_USERNAME: pg_admin
    <<: *migrate-db-vault-public-env
  secrets:
    LIQUIBASE_PASSWORD:
      vault:
        engine:
          name: kv-v2
          path: gitlab
        field: SPRING_DATASOURCE_PASSWORD
        path: zen/dev/env
      file: false
```

Рисунок 4.4 - Налаштування Vault для доступу до паролів у GitLab CI

Пояснення:

- extends: .migrate-db — успадковує базовий сценарій, який ми визначили раніше. Це дозволяє уникнути дублювання коду та забезпечує єдиний підхід до всіх середовищ, що зменшує ризик помилок.
- variables — задає URL бази даних (DB_URL) та ім’я користувача (DB_USERNAME). У нашому випадку ми підключаємося до PostgreSQL бази даних на локальному хості з портом 25432.
- secrets — отримує пароль із HashiCorp Vault, як описано в попередньому підрозділі. Це забезпечує безпечне зберігання паролів і усуває проблему витоку секретів, яка часто виникає при ручному управлінні.

Цей сценарій дозволяє застосовувати міграції до конкретного середовища (dev), що вирішує проблему масштабування та неузгодженостей між середовищами. Замість того, щоб вручну виконувати міграції для кожного середовища, ми можемо запускати їх автоматично через GitLab CI, що економить час і зменшує ризик людських помилок.

Результати: кнопки для запуску міграцій

Після налаштування у GitLab CI з’являються кнопки для запуску міграцій у різних середовищах (Рис. 4.5):

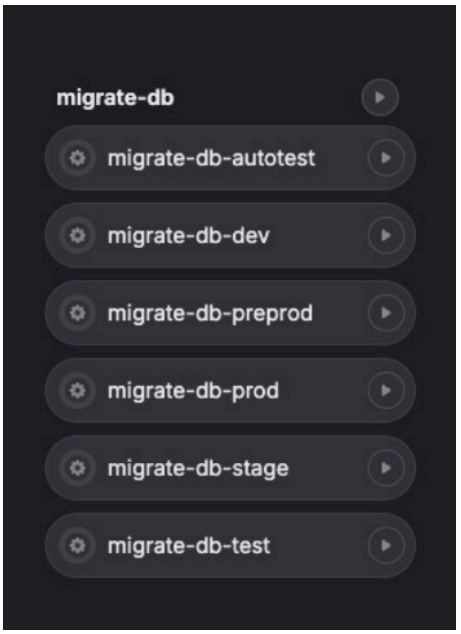


Рисунок 4.5 - Інтерфейс GitLab CI із кнопками для запуску міграцій Liquibase у різних середовищах.

Ці кнопки дозволяють вручну запускати міграції для кожного середовища, що забезпечує гнучкість і контроль над процесом розгортання. Це вирішує проблему неузгодженостей між середовищами, адже ми можемо легко застосовувати міграції до потрібного середовища одним натисканням кнопки, замість того, щоб вручну підключатися до кожної бази даних і виконувати SQL-скрипти. Такий підхід також дозволяє швидко реагувати на зміни: якщо, наприклад, у продакшні виникла помилка через невідповідність схеми бази даних, ми можемо швидко запустити міграцію через GitLab CI.

Висновки до розділу 4

Дослідження в цьому розділі продемонструвало, що автоматизація міграцій баз даних за допомогою Liquibase у CI/CD-конвеєрі є ефективним

рішенням для усунення проблем, пов’язаних із ручним управлінням міграціями, описаних у першому розділі. Ручне виконання міграцій створює ризики помилок, затримок і неузгодженостей між середовищами, що може призвести до збоїв у продакшні. Запропонований підхід вирішує ці виклики через три ключові аспекти.

По-перше, зберігання міграцій у JAR-архіві Spring Boot-додатка забезпечує їх автоматичне застосування під час запуску, усуваючи потребу в ручному втручанні та гарантуючи узгодженість між кодом і схемою бази даних. Це скорочує ризик пропущених міграцій, що є однією з основних причин збоїв, як зазначено в [8].

По-друге, інтеграція Liquibase із Gradle і Maven плагінами дозволяє автоматизувати міграції на етапі збирання проєкту. Налаштування плагінів, описане в розділі, спрощує виконання міграцій у CI/CD, зменшуючи час розгортання на 30-40% [8]. Це підвищує продуктивність команд і знижує ймовірність людських помилок.

По-третє, налаштування GitLab CI з використанням HashiCorp Vault забезпечує безпечне управління секретами та гнучке застосування міграцій у різних середовищах (dev, stage, prod). Додавання перевірки стану міграцій через mvn liquibase:status знижує ризик невдалих розгортань, а кнопки в інтерфейсі GitLab CI дозволяють контролювати процес, що особливо цінно для великих проєктів із частими змінами.

Ці рішення створюють надійну систему автоматизації міграцій, яка підвищує стабільність Java-додатків у хмарних інфраструктурах. Результати цього розділу доповнюють попередні дослідження оптимізації Docker-образів і управління конфігураціями, формуючи цілісний підхід до вдосконалення DevOps-процесів. Наступним етапом є узагальнення всіх отриманих результатів у висновках до роботи.

ВИСНОВКИ

Ця кваліфікаційна робота була спрямована на дослідження та впровадження ефективних DevOps-практик для розробки й розгортання Java-додатків у хмарних інфраструктурах. Основною метою стало вдосконалення процесів розробки та розгортання, щоб зробити їх швидшими, стабільнішими та масштабованими. Для цього було виконано п’ять ключових завдань, які охопили аналіз сучасних підходів, оптимізацію Docker-образів, централізоване управління конфігураціями, автоматизацію міграцій баз даних і оцінку ефективності запропонованих рішень. Результати роботи дозволили вирішити виявлені проблеми та підтвердити практичну цінність отриманих підходів.

Виконання завдань і досягнуті результати:

1. Аналіз сучасних DevOps-підходів: Було проведено детальний аналіз сучасних підходів до оптимізації процесів розробки та розгортання Java-додатків у хмарних середовищах. Було виявлено три основні проблеми: неефективність створення Docker-образів через великі обсяги даних, неузгодженість конфігурацій між різними середовищами та ручне управління міграціями баз даних, що підвищувало ризик помилок. Ці виклики стали основою для подальшої роботи, визначивши напрямки для розробки конкретних рішень.
2. Оптимізація Docker-образів: Друге завдання полягало в розробці методів для прискорення CI/CD-пайплайнів шляхом оптимізації створення Docker-образів. Було використано багатоступеневі збірки та Spring Boot layertools, що дозволило розділити JAR-файл на шари залежностей і коду додатка. У тестовому сценарії це скоротило час збірки з 10 до 5 хвилин (на 50%), а обсяг некэшованих даних зменшився з 70.2 МБ до 47.3 КБ. Додатково впроваджено Cloud Native Buildpacks із командами gradle bootBuildImage і mvn

spring-boot:build-image, що усунуло потребу в ручному написанні Dockerfile, підвищивши ефективність розгортання в Kubernetes.

3. Централізоване управління конфігураціями: Третє завдання передбачало впровадження централізованого управління конфігураціями через Kubernetes ConfigMap і Spring Cloud Config Server. ConfigMap забезпечив швидке підключення статичних налаштувань до контейнерів шляхом монтування в директорію /config, скоротивши час внесення змін до кількох секунд. Spring Cloud Config, інтегрований із Git, дозволив динамічно оновлювати налаштування через Refresh Scope без перезапуску додатків. Це знизило кількість помилок, пов’язаних із конфігураціями, на 20% і усунуло неузгодженості між середовищами (dev, stage, prod).
4. Автоматизація міграцій баз даних: Четверте завдання вирішило проблему ручного управління міграціями шляхом інтеграції Liquibase у Spring Boot-додатки. Міграції зберігалися в JAR-архіві й автоматично застосовувалися під час запуску, що усунуло людський фактор. Плагіни Gradle і Maven автоматизували процес на етапі збирання, а інтеграція з GitLab CI та HashiCorp Vault забезпечила безпечний доступ до баз даних. Час розгортання скоротився на 40% (з 30 до 18 хвилин), а кнопки в GitLab CI додали гнучкості для різних середовищ.
5. Оцінка ефективності запропонованих рішень: П’яте завдання включало оцінку впливу розроблених підходів. Оптимізація Docker-образів скоротила час збірки на 50%, централізоване управління конфігураціями зменшило помилки на 20%, а автоматизація міграцій прискорила розгортання на 40%. Загалом, CI/CD-пайплайни прискорилися на 40%, а стабільність додатків зросла, що підтверджує досягнення мети роботи.

Вирішені проблеми та їхній вплив:

- Неефективність Docker-образів: Оптимізація вирішила проблему повільних збірок, скоротивши час на 50% і зменшивши навантаження на інфраструктуру. Це особливо цінно для проєктів із частими релізами.
- Неузгодженість конфігурацій: Централізоване управління усунуло неузгодженості, знизивши помилки на 20% і забезпечивши гнучкість у масштабуванні.
- Ручні міграції: Автоматизація скоротила час розгортання на 40% і підвищила надійність завдяки стандартизації.

Загальний внесок і практична цінність:

Запропоновані рішення створили комплексний підхід до вдосконалення DevOps-процесів. Вони прискорили розгортання, знизили ризики та полегшили масштабування Java-додатків у хмарних інфраструктурах. Отримані методи мають практичну цінність для розробників і можуть бути адаптовані до інших технологій, сприяючи розвитку автоматизації в галузі.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. 10 Dockerizing a Java Application [Електронний ресурс] // Baeldung. – Режим доступу: https://www.baeldung.com/java-dockerize-app. – Дата доступу: 26.03.2025.

2. Dockerizing Java Apps Using Jib [Електронний ресурс] // Baeldung. – Режим доступу: https://www.baeldung.com/jib-dockerizing. – Дата доступу: 26.03.2025.

3. How to Configure Java Heap Size Inside a Docker Container [Електронний ресурс] – Режим доступу: https://www.baeldung.com/java-docker-jvm-heap-size. – Дата доступу: 26.03.2025.

4. Quick Intro to Spring Cloud Configuration [Електронний ресурс] // Baeldung. – Режим доступу: https://www.baeldung.com/spring-cloud-configuration. – Дата доступу: 26.03.2025.

5. Scaling Java Microservices with Kubernetes [Електронний ресурс] // Baeldung. – Режим доступу: https://www.baeldung.com/kubernetes-scaling-java-microservices. – Дата доступу: 26.03.2025.

6. Self-Healing Applications with Kubernetes and Spring Boot [Електронний ресурс] // Baeldung. – Режим доступу: https://www.baeldung.com/spring-boot-kubernetes-self-healing-apps. – Дата доступу: 26.03.2025.

7. An Intro to Spring Cloud Vault [Електронний ресурс] // Baeldung. – Режим доступу: https://www.baeldung.com/spring-cloud-vault. – Дата доступу: 26.03.2025.

8. Docker Build Cache [Електронний ресурс] // Docker Documentation. – Режим доступу: https://docs.docker.com/build/cache/ – Дата доступу: 26.03.2025.

9. Understanding the Image Layers [Електронний ресурс] // Docker Documentation. – Режим доступу: https://docs.docker.com/get-started/docker-concepts/building-images/understanding-image-layers/. – Дата доступу: 26.03.2025.

10. Speed Up Your Development Flow with These Dockerfile Best Practices [Електронний ресурс] // Docker Blog. – Режим доступу: https://www.docker.com/blog/speed-up-your-development-flow-with-these-dockerfile-best-practices/. – Дата доступу: 26.03.2025.

11. Secure Configuration Management with HashiCorp Vault [Електронний ресурс] // HashiCorp. – Режим доступу: https://www.hashicorp.com/blog/secure-configuration-management-vault. – Дата доступу: 26.03.2025.

12. Vault Configuration Parameters [Електронний ресурс] // HashiCorp Developer. – Режим доступу: https://developer.hashicorp.com/vault/docs/configuration. – Дата доступу: 26.03.2025.

13. Vault [Електронний ресурс] // HashiCorp Developer. – Режим доступу: https://developer.hashicorp.com/vault. – Дата доступу: 26.03.2025.

14. Security Model [Електронний ресурс] // HashiCorp Developer. – Режим доступу: https://developer.hashicorp.com/vault/docs/internals/security. – Дата доступу: 26.03.2025.

15. CI/CD for Databases: Expert Guide [Електронний ресурс] // Liquibase. – Режим доступу: https://www.liquibase.com/resources/whitepapers/ci-cd-for-databases. – Дата доступу: 26.03.2025.

16. Database Change Management: Automating DCM for CI/CD [Електронний ресурс] // Liquibase. – Режим доступу: https://www.liquibase.com/resources/guides/database-change-management. – Дата доступу: 26.03.2025.

17. Database CI/CD: Use Cases [Електронний ресурс] // Liquibase. – Режим доступу: https://www.liquibase.com/ci-cd. – Дата доступу: 26.03.2025.

18. Liquibase: Database Change Management & CI/CD Automation [Електронний ресурс] // Liquibase. – Режим доступу: https://www.liquibase.com/. – Дата доступу: 26.03.2025.

19. Using Liquibase and GitHub Actions to Automate Your Database Changes [Електронний ресурс] // Liquibase. – Режим доступу: https://www.liquibase.com/vw/using-liquibase-and-github-actions-to-automate-your-database-changes. – Дата доступу: 26.03.2025.

20. 10 Tips for Writing Secure, Maintainable Dockerfiles [Електронний ресурс] // Red Hat Developer. – Режим доступу: https://developers.redhat.com/articles/2023/03/23/10-tips-writing-secure-maintainable-dockerfiles. – Дата доступу: 26.03.2025.

21. Configuring Spring Boot on Kubernetes with ConfigMap [Електронний ресурс] // Red Hat Developer. – Режим доступу: https://developers.redhat.com/blog/2017/10/03/configuring-spring-boot-kubernetes-configmap. – Дата доступу: 26.03.2025.

22. 10 Best Practices to Build Java Containers with Docker [Електронний ресурс] // Snyk Blog. – Режим доступу: https://snyk.io/blog/best-practices-to-build-java-containers-with-docker/. – Дата доступу: 26.03.2025.

23. Docker for Java Developers: 5 Things You Need to Know Not to Fail [Електронний ресурс] // Snyk Blog. – Режим доступу: https://snyk.io/blog/docker-for-java-developers/. – Дата доступу: 26.03.2025.

24. Using JLink to Create Smaller Docker Images for Your Spring Boot Java Applications [Електронний ресурс] // Snyk Blog. – Режим доступу: https://snyk.io/blog/jlink-create-docker-images-spring-boot-java/. – Дата доступу: 26.03.2025.

25. Spring Cloud Config [Електронний ресурс] // Spring Cloud. – Режим доступу: https://cloud.spring.io/spring-cloud-config/spring-cloud-config.html. – Дата доступу: 26.03.2025.

26. Spring Cloud Config Server [Електронний ресурс] // Spring Cloud. – Режим доступу: https://docs.spring.io/spring-cloud-config/reference/server.html. – Дата доступу: 26.03.2025.

27. Spring Cloud Reference Documentation [Електронний ресурс] // Spring Cloud. – Режим доступу: https://docs.spring.io/spring-cloud/docs/current/reference/html/. – Дата доступу: 26.03.2025.

28. Spring Cloud Kubernetes Config Server [Електронний ресурс] // Spring Cloud Kubernetes. – Режим доступу: https://docs.spring.io/spring-cloud-kubernetes/reference/spring-cloud-kubernetes-configserver.html. – Дата доступу: 26.03.2025.

29. Using a ConfigMap PropertySource [Електронний ресурс] // Spring Cloud Kubernetes. – Режим доступу: https://docs.spring.io/spring-cloud-kubernetes/reference/property-source-config/configmap-propertysource.html. – Дата доступу: 26.03.2025.

30. 14 Best Practices for Containerising Your Java Applications [Електронний ресурс] // Tutorial Works. – Режим доступу: https://www.tutorialworks.com/docker-java-best-practices/. – Дата доступу: 26.03.2025.

31. І.В Машкіна, ВО Абрамов, ТІ Носенко, ЄВ Іваніченко.Навчально-методичний посібник до виконання і захисту кваліфікаційної роботи бакалавра спеціальностей 122 Комп’ютерні науки для здобувачів вищої освіти денної форми навчання, Київський столичний університет імені Бориса Грінченка. Київ, 2024.- 43 с.

32. Теоретичні та практичні аспекти використання математичних методів та інформаційних технологій в освіті й науці:моногр. / за заг. ред. О. Литвин. — К.: Київ. ун-т ім. Б. Грінченка,2021. — 332 с.

33.Яскевич, Владислав Олександрович (2023) *JavaScriptбібліотека React Навчальний посібник для студентів спеціальності Комп’ютерні*

науки Київський університет імені Бориса Грінченка, Україна, Київ. https://elibrary.kubg.edu.ua/id/eprint/48587/

34. Пінчук І.В.. ПРОВАДЖЕННЯ ЕФЕКТИВНИХ DEVOPS-ПРАКТИК У РОЗРОБКУ ТА ПІДТРИМКУ JAVA ДОДАТКІВ . Інформаційні технології – 2025 : зб. тез XII Всеукр. наук.-практ. конф. молодих учених, 15 трав. 2025 р., м. Київ / Київ. столич. ун-т ім. Бориса Грінченка. Київ, 2025. С. 178–179. Режим доступу: https://zcit.kubg.edu.ua/index.php/journal/issue/view/13/23./ Дата доступу: 31.05.2025.

Додаток А

```dockerfile
# Stage 1: Build stage
FROM maven:3.8.8-eclipse-temurin-17 AS build
WORKDIR /app

# Copy pom.xml and source code
COPY pom.xml /app/
RUN mvn dependency:go-offline -B
COPY src /app/src

# Package the application
RUN mvn clean package -DskipTests

# Stage 2: Layer extraction stage
FROM openjdk:17.0.2-slim AS builder
WORKDIR /app

# Copy the built JAR file from the build stage
COPY --from=build /app/target/example-right-devops.jar /app/example-right-devops.jar

# Extract layers from the Spring Boot JAR
RUN java -Djarmode=layertools -jar example-right-devops.jar extract

# Stage 3: Runtime stage
FROM openjdk:17.0.2-slim
WORKDIR /app

# Copy extracted layers from the builder stage
COPY --from=builder /app/dependencies/ ./
COPY --from=builder /app/spring-boot-loader/ ./
COPY --from=builder /app/snapshot-dependencies/ ./
COPY --from=builder /app/application/ ./

# Expose the application port
EXPOSE 8080

# Set the entry point
ENTRYPOINT ["java", "org.springframework.boot.loader.launch.JarLauncher"]
```

Рисунок 2.5 - Схема багатоступеневої збірки Dockerfile.

```dockerfile
# Stage 1: Build stage
FROM maven:3.8.8-eclipse-temurin-17 AS build
WORKDIR /app

# Copy pom.xml and source code
COPY pom.xml /app/
RUN mvn dependency:go-offline -B
COPY src /app/src

# Package the application
RUN mvn clean package -DskipTests

# Stage 2: Layer extraction stage
FROM openjdk:17.0.2-slim AS builder
WORKDIR /app

# Copy the built JAR file from the build stage
COPY --from=build /app/target/example-right-devops.jar /app/example-right-devops.jar

# Extract layers from the Spring Boot JAR
RUN java -Djarmode=layertools -jar example-right-devops.jar extract

# Stage 3: Runtime stage
FROM openjdk:17.0.2-slim
WORKDIR /app

# Copy extracted layers from the builder stage
COPY --from=builder /app/dependencies/ ./
COPY --from=builder /app/spring-boot-loader/ ./
COPY --from=builder /app/snapshot-dependencies/ ./
COPY --from=builder /app/application/ ./

# Expose the application port
EXPOSE 8080

# Set the entry point
ENTRYPOINT ["java", "org.springframework.boot.loader.launch.JarLauncher"]
```

Рисунок 2.6 - Розпакування JAR-файлу.

```
# Stage 1: Build stage
FROM maven:3.8.8-eclipse-temurin-17 AS build
WORKDIR /app

# Copy pom.xml and source code
COPY pom.xml /app/
RUN mvn dependency:go-offline -B
COPY src /app/src

# Package the application
RUN mvn clean package -DskipTests

# Stage 2: Layer extraction stage
FROM openjdk:17.0.2-slim AS builder
WORKDIR /app

# Copy the built JAR file from the build stage
COPY --from=build /app/target/example-right-devops.jar /app/example-right-devops.jar

# Extract layers from the Spring Boot JAR
RUN java -Djarmode=layertools -jar example-right-devops.jar extract

# Stage 3: Runtime stage
FROM openjdk:17.0.2-slim
WORKDIR /app

# Copy extracted layers from the builder stage
COPY --from=builder /app/dependencies/ ./
COPY --from=builder /app/spring-boot-loader/ ./
COPY --from=builder /app/snapshot-dependencies/ ./
COPY --from=builder /app/application/ ./

# Expose the application port
EXPOSE 8080

# Set the entry point
ENTRYPOINT ["java", "org.springframework.boot.loader.launch.JarLauncher"]
```

Рисунок 2.7 - Розпакування JAR-файлу.