Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Факультет інформатики
Кафедра мультимедійних систем

ЗАСТОСУВАННЯ DOCKER КОНТЕЙНЕРІВ ДЛЯ ВИКОНАННЯ СИСТЕМНИХ ПРОГРАМ

Текстова частина до курсової роботи

за спеціальністю 121 «Інженерія програмного забезпечення»

Керівник курсової роботи
ст.в. к-т. т. н.___Черкасов Д.І.___
(прізвище та ініціали)
_____
(підпис)
“_—_” _____ 2022 р.

Виконав студент Папроцький І.А.
(прізвище та ініціали)
_____
(підпис)
“_—_” _____ 2022 р.

Київ-2022

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мережних технологій факультету інформатики

ЗАТВЕРДЖУЮ

Завідувач кафедри мережних технологій,

доктор. фіз-мат. наук – Малашонок Г.І.

_____ *(підпис)*

“____” _____ 2021 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на курсову роботу

студенту Папроцькому Ігорю Андрійовичу

Факультету інформатики 1 р.н. магістерської програми «Інженерія програмного забезпечення»

ТЕМА: Застосування Docker контейнерів для виконання системних програм

Зміст ТЧ до курсової роботи:

Календарний план

Вступ

Огляд теоретичного матеріалу та здійснення дослідження

Висновки

Список використаної літератури та посилань

Додатки (за необхідністю)

Дата видачі “____” _____ 2021 р. Керівник _____

(підпис)

Завдання отримав _____

(підпис)

Тема: Застосування Docker контейнерів для виконання системних програм

Календарний план виконання роботи:

№	Назва етапу	Термін виконання	Примітка
1.	Отримання теми курсової роботи	11.10.2021	
2.	Пошук тематичної літератури	21.11.2021	
3.	Ознайомлення з літературою	25.12.2021	
4.	Накопичення теоретичної бази для текстової частини роботи	15.02.2022	
5.	Написання тестової частини роботи	20.03.2022	
6.	Реалізація практичних прикладів для підкріплення теоретичної частини роботи	20.03.2022	
7.	Перегляд змісту роботи керівником	15.05.2022	
8.	Внесення змін до курсової роботи відповідно до зауважень наукового керівника	20.05.2022	
9.	Створення презентації	25.05.2022	
10.	Здача роботи для перевірки на плагіат	09.06.2022	
11.	Отримання оцінки наукового керівника	13.06.2022	

ЗМІСТ

АНОТАЦІЯ

У роботі розглянуті деякі аспекти використання платформи Docker, її архітектури, концепцій віртуалізації і контейнеризації. Основна увага зосереджена на огляді платформи з точки зору зручності платформи для використання при розробці програмного забезпечення. Практична частина роботи зосереджена на розробці практичних прикладів способів використання контейнеризації для розгортання локальної версії бази даних та виконання міграцій схеми БД за допомогою допоміжних інструментів. В результаті теоретичних та практичних частин курсової роботи проаналізовано переваги використання Docker у робочому процесі інженерів з розробки програмного забезпечення, а також у інших сценаріях його використання.

У першому розділі розглянута предметна область, а також проаналізовано розповсюджені сценарії використання платформи. У цьому ж розділі відбувається постановка завдання курсової роботи.

У другому розділі розглянуті теоретичні аспекти концепцій віртуалізації та контейнеризації, розглянута архітектура та деякий інструментарій платформи Docker.

У третьому розділі описано конкретний сценарій використання платформи у розробці програмного забезпечення, виходячи з якого створено пункти технічного завдання для практичних прикладів. Детально розглядається практична частина курсової роботи, реалізація поставлених завдань, частина коду прикладів винесена в додатки. В кінці розділу проаналізовано практичний приклад.

З усього вищезгаданого зроблені висновки в кінці роботи.

ВИКОРИСТАНІ СКОРОЧЕННЯ

- **CI/CD** – Continuous Integration/Continuous Deployment (у пер. з англ. – безперервна інтеграція та безперервна доставка) – комбінація практик налаштування автоматизації процесу побудови, тестування та розгортання додатків.
- **БД** – База даних.
- **IDE** – Integrated Development Environment – програмне забезпечення для розробки та написання коду з широким інструментарієм. У даній курсовій роботі у практичній частині використовується IDE IntelliJ IDEA.

ВСТУП

Актуальність та практичне значення обраної теми

В еру широкого виростання хмарних технологій у процесі розгортання та підтримки застосувань прийнято приділяти велику увагу саме тому, де саме знаходитиметься той чи інший сервер застосунку. Крім цього, велику увагу приділяють способу запуску застосування, надання доступу до ресурсів та портів, налаштування мережі та взаємодії з іншими застосунками або системами, а також багатьом іншим аспектам роботи з програмним забезпеченням. Як результат та як рішення поставленим завданням, більшість сучасних реалізацій веб-додатків не обходяться без необхідності віртуалізації робочого середовища.

Одним із найбільш розповсюджених способів віртуалізації являється використання Docker контейнерів. Саме завдяки платформі Docker перед командою розробників з'являються можливості гнучкого управління екземплярами застосування, її ресурсами, мережею та взаємодією з іншими програмними компонентами.

Більше того, платформа Docker дозволяє запускати контейнер на більшості сучасних операційних систем, адже всередині контейнеру абсолютно незалежно можна розгорнути образ іншої операційної системи. В свою чергу, це дозволяє фактично не залежати від батьківського середовища, де розгортається застосунок. Як результат, фахівці широко користуються можливостями Docker для виконання як невеличких точкових завдань на цільовій операційній системі, так і для розгортання комплексного середовища з багатьма складно пов'язаними компонентами.

Таким чином, знання деталей використання Docker часто являється вимогою на співбесідах при прийомі на роботу як на позицію розробника прикладного програмного забезпечення, так і на позиції DevOps інженерів, системних адміністраторів, і навіть тестувальників. Навіть більше, ці знання дійсно допомагають фахівцям якісно і швидко виконувати свої завдання.

Структура роботи

Робота складається зі вступу, трьох розділів, висновку, списку використаних джерел та набору додатків.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ. ПОСТАНОВКА ЗАВДАННЯ КУРСОВОЇ РОБОТИ

1.1 Базовий огляд предметної області

Варто розпочати з того, що Docker, згідно офіційної документації, це відкрита платформа для розробки, доставки, та запуску програмного забезпечення. Головною перевагою продукту являється можливість управляти своєю інфраструктурою таким же чином, як розробник управляє своїм застосунком. Так, завдяки його перевагам щодо упаковки додатків, їх доставки, тестування та розгортання, можна значно зменшити час, потрібний на впровадження і налаштування всіх процесів, що знаходяться між моментом написання коду застосунку, та моментом запуску застосунку у потрібному середовищі у потрібних умовах. [1]

Як відомо, робота з Docker базується навколо упакування та запуску застосунку в деякому ізольованому середовищі, яке називається контейнером. Причому, на одній робочій машині можна запускати багато контейнерів з різним вмістом одночасно. У контейнери запаковують все, що потрібно для роботи цільового застосунку. Таким чином, запускаючи екземпляри конкретно налаштованого контейнеру на будь-яких інших робочих машинах, користувач може бути певен, що вони працюватимуть однаково. Для підтримання повторного використання контейнерів для багатьох різних запусків однакових контейнерів, або, наприклад, поширення контейнерів між своїми колегами, розробник може упакувати його конфігурацію у спеціальний образ. Сама ж конфігурація поміщається у спеціальний файл під назвою Dockerfile де містяться усі інструкції для налаштування контейнеру під час його запуску. У разі необхідності створення одразу кількох контейнерів, які можуть взаємодіяти між собою, розробнику надається можливість використання інструменту Docker Compose, а конфігурацію для усіх контейнерів можна зручно описати у спеціальному файлі формату YAML та назвою «docker-compose.yml».

Більше того, беручи до увагу функціонал упаковування конфігурації контейнерів у образи, варто згадати й про зручну та широко використовувану платформу Docker Hub. За допомогою неї користувачі можуть ділитися своїми збірками конфігурацій зі спільнотою. Таким же чином, на платформі розміщені велика кількість офіційних образів, з деяким стандартним функціоналом, наприклад з операційною системою Linux необхідної версії, та уже встановленими бібліотеками або інструментами в них. Користувач може вільно використати їх у своєму файлі конфігурації, додатково налаштовувати та доповнювати усім необхідним для його потреб. При створенні контейнеру образу буде завантажений із платформи та налаштований з усіма необхідними інструкціями.

Варто зазначити, що принципи віртуалізації, контейнеризації, архітектура Docker та способи використання згаданих інструментів будуть детально розглянуті у наступних розділах.

Оглядаючи предметну область варто також згадати шалену популярність використання даної платформи. Поглянемо на статистику представлену спільнотою відомого сайту Stack Overflow у їх щорічному опитуванні користувачів. У 2021 році із 76 тисяч опитуваних близько 48% активно використовували впродовж року, або хотіли б використовувати у наступному році Docker як інструмент у своїх проектах. [2] Таким чином, це друга найбільш використовувана технологія серед інших інструментів (Other tools) користувачами сайту, що відображає загальний тренд серед фахівців по світу.

1.2 Аналіз розповсюджених сценаріїв використання Docker

Використовуючи вищеописані принципи, контейнери Docker стають у нагоді для дуже багатьох цілей на різних етапах розробки та впровадження застосунку.

Так, наприклад, під час написання веб застосунку розробник може створювати допоміжні контейнери для запуску та автоматичного налаштування

бази даних для інтеграційних тестів (яку можна безліч разів запускати з абсолютно ідентичною конфігурацією, заповнювати необхідними даними та очищати при виконанні тестів). Навіть більше, для застосунків на мові програмування Java, наприклад, існує ціла бібліотека під назвою Testcontainers, яка дозволяє автоматично запускати окремі Docker контейнери з необхідною архітектурою для кожного тестового класу або тестового методу, гнучко управляти ними, і також автоматично вимикати їх після завершення тестування.

Той же розробник може одночасно використовувати Docker для локального запуску у контейнері емулятора середовища хмарного провайдера AWS під назвою LocalStack, і таким чином запускати всю мережу необхідних застосунків, лямбда функцій, файлових бакетів, черг повідомлень, та інших сервісів, якими він користується у реальному середовищі при роботі його системи, у даному випадку, для локального тестування та відлагодження бізнес процесів перед їх розгортанням на реальному середовищі у хмарного провайдера.

У іншій частині процесу розробки, згаданий уявний розробник може використовувати Docker контейнери для автоматичного запуску усіх необхідних процедур CI/CD для побудови, тестування та розгортання його застосунку. Прикладами таких готових інструментів, які інтегруються з Docker контейнерами, являються продукти CircleCI, Gitlab CI/CD, Jenkins та інші.

Продовжуючи приклад, розробник та його команда може розгортати однакові за умовами оточення для розробки (зазвичай називають - development), тестування (test), підготовки та перевірки до остаточного розгортання (staging), та, власне остаточної роботи застосунку (production). Ці оточення запускаються часто саме в контейнерах, відрізняючись лише складом коду на різний момент часу розробки, адже всі вони використовуються для різних цілей, а функціонал продукту поступово просувається від оточення розробки до кінцевого оточення. До того ж, часто ці контейнери керуються системами керування контейнерами, прикладами яких є Kubernetes або OpenShift.

До описаного ланцюга прикладів можна додати й наступні сценарії та випадки використання платформи Docker:

- застосування працюватиме на будь-якому пристрої, адже контейнер є ізольованим від середовища, в якому виконується. Таким чином, користувачу не потрібно встановлювати пакети програмного забезпечення, необхідні для роботи застосунку, на своєму локальному середовищі. Усе буде зроблено всередині контейнеру, із завжди однаковою операційною системою, ресурсами, та іншими необхідними компонентами;
- за допомогою контейнерів можна значно спрощувати процес налаштування локального середовища розробки для новоприбулих працівників певного робочого проекту. Маючи заздалегідь прописану конфігурацію усіх необхідних контейнерів можна не просто полегшувати процес розробки на будь-якому пристрої, а і позбуватись необхідності ручного налаштування усіх необхідних компонентів системи для усіх причетних до її розробки та підтримки;
- оскільки з кожним днем все більше проектів починають використовувати мікросервіси як основну архітектуру своїх проектів, Docker стає у нагоді завдяки можливості легко розгортати та керувати контейнерами із мікросервісами на різних оточеннях. У порівнянні з монолітним застосунком, така система піддає набагато меншому ризику процес розробки у разі необхідності горизонтально масштабувати застосунок, адже часто потрібно лише підв’язати більше контейнерів із застосуванням до вже існуючих; [3]
- контейнери також часто використовуються для вирішення проблеми розгортання тренованих моделей машинному навчанні. Оскільки головною складністю являється правильний запуск одразу багатьох таких моделей у кінцевому оточенні, яке працюватиме з кінцевим користувачем,

контейнери Docker дозволяють запускати й керувати такими застосунками надзвичайно зручно та швидко.

З усього вищенаведеного головним висновком являється наявність великої кількості способів застосування контейнерів у розробці програмного забезпечення.

1.3 Постановка завдання курсової роботи

Відповідно до згаданої популярності платформи Docker, її широкої функціональності, великої кількості можливих сценаріїв використання було виділено такі завдання для цієї курсової роботи:

- описати концепції віртуалізації та контейнеризації, різницю між ними;
- описати архітектуру Docker, деталі її роботи;
- дослідити переваги використання Docker під час розробки, підтримки та впровадження програмного забезпечення, та вплив його використання на витрати робочого часу у розробників;
- на практичному прикладі дослідити переваги використання платформи для виконання системних програм незалежно від зовнішнього середовища;
- зробити висновки щодо результатів опису та дослідження.

РОЗДІЛ 2. ТЕОРЕТИЧНІ ВІДОМОСТІ

2.1 Концепція віртуалізації

Як відомо, у середині 90-х років фахівці почали розуміти, що не всі фізичні пристрої, на яких працювало програмне забезпечення, використовували весь свій потенціал. [4] Наприклад, деякі програмні рішення могли бути використані лише на конкретних типах пристроїв або конкретних операційних системах, що заставляло організації витрачати додаткові кошти та час на закупівлю, встановлення та налаштування конкретних типів пристроїв для своїх серверів. Саме в таких ситуаціях у нагоді стала віртуалізація, адже з нею організації могли поділити свої фізичні пристрої на частини, на яких могли запускатись одразу кілька операційних систем та одиниць необхідного їм різного програмного забезпечення. Таким чином досягався виграш у кількості коштів та сил робітників направлених на закупівлю необхідного обладнання та підтримання інфраструктури дата-центрів у цілому.

Головна ідея віртуалізації заключається в тому, щоб поділити фізичні ресурси пристрою, створюючи віртуальну або програмно створену версію обчислювального ресурсу за допомогою спеціального програмного забезпечення. [5] За допомогою віртуалізації кілька операційних систем або застосувань можуть бути запущені одночасно на одному фізичному пристрої, підвищуючи цим гнучкість та ефективність використання апаратного забезпечення.

Відповідно, при використанні меншої кількості пристроїв, дата центром стає простіше та практичніше управляти, чим організації, що мають такі дата центри активно і користуються. Додатково, підвищується рівень захищеності в плані кібербезпеки при роботі з віртуальними машинами, і однією з головних причин на це є краща ізоляція ресурсів системи та краща керованість моніторингом доступу до такої системи. Сучасні компанії (особливо у сфері IT) часто використовують концепцію віддаленої роботи працівників саме через

«віддалений робочий стіл», тобто за допомогою інструментів, які надають можливість працювати з віртуальною машиною так, ніби вона встановлена на локальному комп’ютері користувача. При такій роботі роботодавці можуть краще моніторити переміщення даних із та в систему, а також можуть не переживати за небезпеку викрадення фізичного пристрою робітника, що могло б привести до втрати конфіденційних даних компанії, якби вони фізично зберігались на викраденому комп’ютері.

У класичному вигляді віртуалізацію можна зобразити за допомогою наступної схеми:

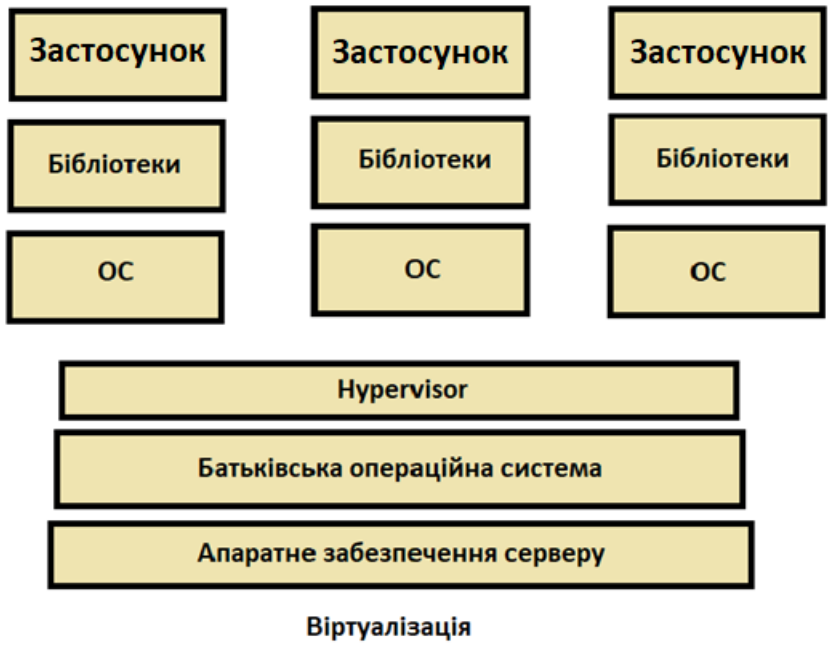


Рисунок 1. Віртуалізація у класичному розумінні

Найцікавішим елементом на цій схемі являється гіпервізор (Hypervisor), який є спеціальним програмним забезпеченням, апаратним забезпеченням або прошивкою, який використовується для роботи з віртуальними машинами. Через гіпервізор кожній віртуальній машині надаються не тільки необхідні бібліотеки та залежності, а і віртуалізований стек обладнання, включаючи ядра процесора, сховище та мережеві адаптери. Для того щоб це все запрацювало, віртуальній машині необхідна своя повноцінна гостьова операційна система. Гіпервізор

може бути запущений з батьківської операційної системи пристрою, або як вбудоване в «залізо» застосування. [6]

2.2 Контейнеризація та її відмінність від віртуалізації

Під визначенням контейнеризації можна розуміти форму віртуалізації в якій застосунки запускаються в ізольованих середовищах, хоча поділяють одну операційну систему. Усе що потрібно застосунку: скомпільовані бібліотеки, конфігурації та залежності – все це інкапсулюють в контейнері. [7]

Кожен контейнер це виконуваний пакет програмного забезпечення, який запускається над батьківською операційною системою. Зверху над нею працює контейнерна платформа зі своєю мінімалістичною операційною системою, яка залежить від технології контейнерної платформи, що використовується. Над платформою знаходяться бібліотеки та скомпільований програмний код для кожного застосування в окремих контейнерах.

Також відомо, що контейнеризація еволюціонувала з інструменту cgroups, призначеному для ізоляції та контролю використання ресурсів процесами у ядрі Linux. Із cgroups було розроблено Linux containers (LXC), з більш прогресивним функціоналом для ізоляції компонентів за просторами імен, таких як таблиці роутінгу та файлові системи. Як результат, LXC контейнер може приєднуватись до файлової системи, запускати системні команди від імені адміністратора та отримувати IP адресу. [7]

При тому що контейнери потребують необхідних бібліотек всередині себе для запуску застосувань, LXC контейнер не упаковує в себе повноцінне ядро операційної системи, що дозволяє йому споживати набагато менше обчислювальних потужностей ніж повноцінна віртуальна машина. Саме технологія LXC стала основою для платформи Docker.

До моменту появи платформи Docker, розробники використовували віртуалізацію для розробки застосувань. Однією з проблем, які з’являлися при досягненні моменту запуску застосування для використання кінцевими

користувачами (фаза production), була проблема невдалого використання віртуальних машин для запуску мікросервісної архітектури [8]. Кожен мікросервіс (або просто сервіс у сервіс-орієнтованій архітектурі) розгортався на окремій віртуальній машині, і на одному фізичному пристрої запускалось декілька віртуальних машин. Головним недоліком такого підходу було неефективне використання обчислювальних ресурсів пристрою. Мікросервіси не використовували всі ресурси (оперативна пам’ять, навантаження процесора, дисковий простір) віртуальної машини які були їм надані, а при великій кількості таких мікросервісів втрати відчувалися в тому числі й на загальній вартості такого проекту, а управління такими застосунками було досить складним.

З появою Docker розробники змогли запускати застосування в окремих оточеннях – контейнерах. Це дозволило бути певним що кожен однаковий екземпляр застосунку в контейнері буде запускатись однаково не залежно від того, в якому оточенні або на якому пристрої він запущений, без будь-яких проблем із сумісністю. Docker не вимагає конкретного попереднього визначення кількості ресурсів для контейнеру та його виділення розробником, а навпаки, використовує стільки обчислювальних потужностей пристрою, скільки йому потрібно, що і стало великим кроком в сторону підтримки мікросервісної архітектури.

Якщо підсумувати вищесказане: схоже до контейнеризації, традиційна віртуалізація дозволяє повністю ізольовувати застосування, при цьому воно буде використовувати ресурси інфраструктури під ним. Головні ж відмінності заключаються у наступному: оскільки віртуальні машини вимагають для своєї роботи повноцінну гостьову операційну систему та її ядро, а також досить масивний по навантаженню на ресурси пристрою прошарок (гіпервізор) між ними та батьківською операційною системою, сильно зменшується ефективність використання обчислювальних потужностей; через цю проблему пристрій, який би міг комфортно працювати з 10 і більше контейнерами, може відчувати труднощі з підтримкою однієї віртуальної машини.

2.3 Архітектура платформи Docker та його інструментарій

Відповідно до попереднього пункту, можна зобразити наступну схему архітектури Docker контейнерів:

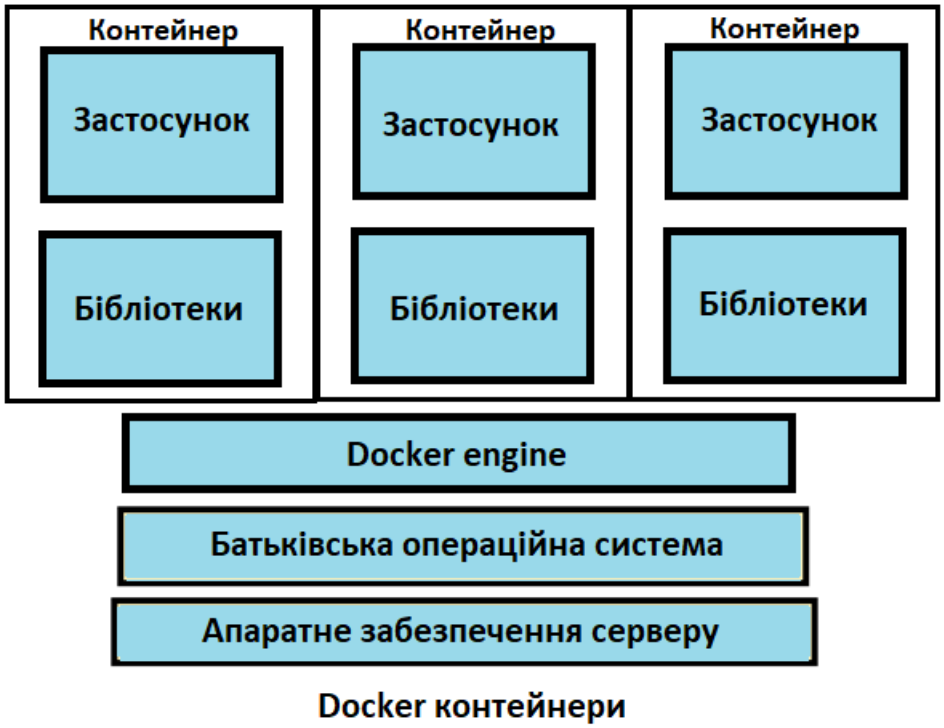


Рисунок 2. Архітектура контейнерів Docker

Проміжним шаром між батьківською операційною системою та елементами віртуалізації в даному випадку виступає Docker engine. Docker engine, в свою чергу, це комбінація Docker daemon, REST API, та інструменту Docker CLI для командного рядка.

Перейдемо до компонентів платформи Docker. Серед них можна виділити наступні [9]:

- Docker Client – інструмент CLI (інтерфейсу командного рядка), який використовується для управління та взаємодії з платформою, наприклад за допомогою команди «docker run» та заданих параметрів створює та запускає контейнер.

- Docker Daemon – сервіс який працює як сервер докеру, запускається у вигляді daemon-процесу, тобто сервіс який працює на задньому фоні у батьківській операційній системі. Він, власне, й приймає команди від CLI, а також слухає на запити до його API, та управляє об’єктами платформи (контейнерами, образами, томами пам’яті та мережами). Він також може комунікувати з іншими daemon процесами для свого управління;
- образи (images) контейнерів – як вже було раніше описано, готові зразки контейнерів, які можна також публікувати на публічних або приватних репозиторіях Docker Hub;
- Docker Registries – це компонент, який являється центральним репозиторієм образів Docker та включає в себе Docker Hub. За замовчуванням, публічні репозиторії Docker Hub встановлені в конфігурації користувача Docker, а при використанні команд “docker pull” або “docker run” Daemon використовуватиме репозиторії зі своєї конфігурації для викачування необхідних образів;
- Docker Engine – комбінація Docker Daemon, його REST API та Docker Client.
- Dockerfile – файл, який використовується для опису образів контейнерів.
- власне, контейнери.

Існує також хмарний провайдер Docker Cloud, за допомогою потужностей якого можна розгортати та управляти контейнерами.

Варто згадати також інструмент Docker Compose, який за допомогою спеціального файлу дозволяє створювати та конфігурувати одразу багато контейнерами. Така команда до клієнту як “docker-compose up” дозволяє запустити контейнери описані у файлі.

РОЗДІЛ 3. ОПИС ПРАКТИЧНОГО ДОСЛІДЖЕННЯ

3.1 Використання Docker у робочому процесі розробки застосунків

Оскільки серед цілей даної курсової роботи являється опис платформи і варіантів її використання, для реалізації практичної частини було вирішено сконцентрувати увагу саме на корисності Docker-контейнеризації для процесів які виконуються розробниками у їх роботі.

Часто інженерам в IT, які розробляють застосунки що зберігають інформацію в базах даних, в процесі розробки доводиться виконувати одноманітні дії із її налаштування, встановлення, запуску, очищення та наповнення даними. Такі дії необхідні часто одночасно для різних цілей, наприклад:

- для ручного тестування роботи застосунку на локальному пристрої розробника під час написання коду;
- для відтворення помилок («багів»), які виявлені під час роботи застосунку;
- для запуску наборів інтеграційних тестів, які, до того ж, зазвичай запускаються під час кожного фінального етапу будови коду застосунку, та перед внесенням змін на загальну гілку розробки;
- для запуску наборів інтеграційних тестів для перевірки роботоздатності застосунку під час виконання CI/CD процесів перед розгортанням нової версії застосунку.

Таку локальну копію тієї бази даних, що працює на різних оточеннях та використовується часто навіть різними командами (наприклад, “dev” використовується командою розробки, а “test” використовується командою тестувальників, тоді як “production” використовується кінцевими користувачами застосунку), досить зручно запускати саме у Docker контейнерах, адже це сильно економить час та зусилля інженерів.

Також при застосуванні підходу до розробки “database first” (підхід до проектування архітектури веб-застосунку, у якому спочатку розробляється схема

бази даних, а потім до неї пристосовують код застосунку), до якого часто вдаються розробники проектів, прийнято використовувати системи міграцій скриптів до баз даних, такі як Flyway або Liquibase. Такі скрипти завжди знаходяться в окремій директорії проекту та застосовуються до бази даних по черзі, в залежності від її поточного стану. Таким чином, базу даних тримають у необхідному стані, відповідно до версій файлів із SQL скриптами, які пишуть розробники. До згаданої локальної бази даних завжди потрібно перед початком роботи з нею застосовувати міграції SQL скриптів, що також можна робити в окремому контейнері, або з локального оточення розробника.

Більше того, інженерам часто необхідно аби всі необхідні кроки для налаштування такої бази даних могли бути зроблені швидко та з найменшою кількістю необхідних дій, адже це прямо впливає на швидкість вирішення задач, що, в свою чергу, на найнижчому рівні впливає на успішність проекту. Також важливо щоб кожен раз база даних була абсолютно однакова, з однаковими версіями, конфігурацією, структурою та наповненням, незалежно від того, на якій системі або у якому оточенні така база дана запускається.

Саме тому, на основі практичного прикладу, у наступних пунктах даного розділу буде розглянуто спосіб використання контейнеризації для полегшення та пришвидшення робочого процесу інженерів програмного забезпечення, які працюють з базою даних у своїх проектах.

3.2 Постановка та аналіз технічного завдання

Відповідно до попереднього пункту, завданням практичної частини цієї курсової роботи будуть наступні пункти:

- розробити приклад конфігурації та запуску контейнеру з реляційною базою даних та застосуванням SQL міграцій із локального середовища розробника;
- розробити варіант попереднього пункту використовуючи додатковий контейнер для міграцій;

- проаналізувати обидві реалізації та виділити переваги обраних підходів перед ручним налаштуванням бази даних на локальному пристрої розробника.

Обрані технології реалізації, пояснення їх вибору, та, власне, реалізація буде розглянуто у наступному пункті даного підрозділу.

3.3 Огляд результатів реалізації першого практичного прикладу

Переходячи до практичних прикладів варто зазначити, що обрані технології проекту (СУБД, система автоматичного збирання проекту, мови програмування та скриптів) можуть бути замінені на будь-які інші, підходящі під конкретний проект. Головною ціллю практичної частини у цій роботі – показати саме зручність використання контейнерів у процесі розробки програмного забезпечення.

Отже, варто розпочати з огляду проекту з SQL міграціями. Часто такий проект для зручності відділяють в окремий репозиторій. Таким чином, цей проект відповідає лише за все, що пов’язано зі схемою бази даних. На Рисунку 3 зображений скріншот структури реалізованого проекту для даного практичного прикладу, файли якого будуть розглядатися впродовж цього розділу.

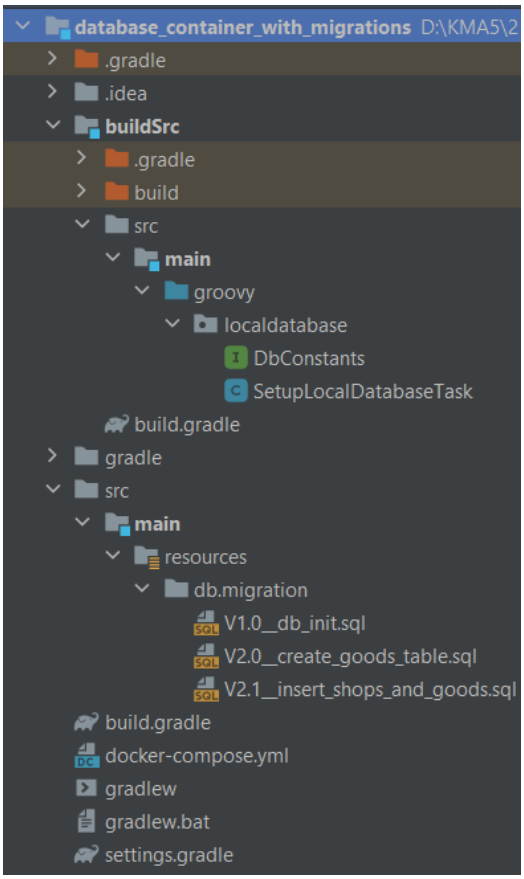


Рисунок 3. Структура практичного проекту

У даному випадку, в якості системи автоматичного збирання проекту обрана система Gradle. За принципами вона схожа на Apache Ant та Apache Maven, але використовує об’єктно-орієнтовану мову Groovy в якості бази свого синтаксису. За допомогою Gradle у даному проекті визначено батьківський модуль, конфігурація якого (як і будь-якого іншого модуля в системі) описується файлом з назвою «build.gradle». Також у проекті, окрім батьківського модуля, присутній модуль з назвою «buildSrc», назву і зміст якого буде розглянуто трішки пізніше. У батьківському ж модулі у директорії «src» можна побачити набір директорій в яких містяться файли з кодом проекту. У даному випадку, оскільки цей проект призначений для міграцій схеми бази даних, у них міститься лише папка «resources» з файлами міграцій.

Для цього проекту обрано поширену СУБД Postgres, яка є реляційною та використовує мову SQL. Саме тому файли міграцій мають формат «.sql». Варто також зазначити, що у якості системи міграцій у даному проекті обрано систему

Flyway. Відповідно до конфігурації за замовчуванням, файли міграцій розміщуються у директорії з ресурсами «db/migration». На згаданому рисунку можна побачити три файли з міграціями, які додані для прикладу. Їх зміст можна побачити у **Додатках 1, 2 та 3**. В них представлено створення деяких схеми з назвою «test_shop», та таблиць «shop» та «goods», а також додавання деяких початкових даних до цих таблиць.

Отже, у першому варіанті практичного прикладу у технічному завданні визначено використання вбудованої системи міграцій одразу із проекту. Такий варіант обраний для реалізації саме тому, що часто у проекті необхідний більший функціонал системи міграцій, ніж просто процес відправки скриптів міграцій до бази даних. Наприклад, часто у проекті використовуються додаткові скрипти з використанням функцій Flyway для відновлення певної версії схеми бази даних, очищення її під час тестування, та інші. Саме тому цю систему підключають одразу в проект за допомогою того ж Gradle.

Оскільки Gradle дозволяє користувачам описувати свої «завдання» (tasks), які можуть бути використані в життєвому циклі проекту, ця можливість і буде використана для даного проекту. Як можна побачити у **Додатку 4**, який містить файл конфігурації батьківського модуля проекту, окрім визначення необхідних змінних, у ньому визначено також декілька корисних «завдань» життєвого циклу, таких як «flywayLocalClean», «flywayLocalMigrate» та «flywayLocalRepair», а також, що найцікавіше, серед усіх «завдань» проекту реєструється також завдання «startLocalDatabase», яке описано не у цьому файлі, а в окремому модулі **buildSrc**. Це можна побачити на наступному скріншоті із цього додатку:

```
65  tasks.register('startLocalDatabase', SetupLocalDatabaseTask) { Task t ->
66      t.finalizedBy(flywayLocalClean, flywayLocalMigrate)
67  }
```

Рисунок 4. Реєстрація "завдання" startLocalDatabase у проекті

Відповідно до розширеного функціоналу системи Gradle, у проекті використовується модуль з такою назвою саме тому, що усі класи з цього модуля доступні у файлі «build.gradle» напряму, що не є можливим без використання модуля з такою назвою. [10]

Отже, у згаданому модулі міститься клас SetupLocalDatabaseTask мовою Groovy, який розширює клас стандартних «завдань» з бібліотеки gradle. Код цього класу можна побачити у **Додатку 5**, де головним методом являється initializeContainerIfNotExists(). Саме в ньому визначаються дії, які будуть запускатись зареєстрованим раніше «завданням». У ньому використовується бібліотека Testcontainers, яка була вже згадана в цій роботі. Бібліотека дозволяє створювати Docker контейнери та конфігурувати їх в коді. Це аналог тих дій, що можна було б описати в Dockerfile, або за допомогою CLI команди до Docker Client, але виконується вона під час виконання коду застосування, або скрипту (у даному випадку, скрипту gradle). Спочатку метод за допомогою приватного метода configureTestContainers() налаштовує бібліотеку на перевикористання контейнеру з такими параметрами, якщо він вже існує. Таким чином це економить кількість новостворених контейнерів при наступних запусках скрипту. Далі в методі створюється об’єкт класу PostreSQLContainer, в якому використовуються константи з інтерфейсу DbConstants, який представлений в **Додатку 6**. Цікаво також, що в кінці цього методу передаються дані про створену базу даних до наступних «завдань», виконання яких після закінчення даного було вказано на **Рисунку 4**. Конфігурація та доповнення цих завдань, які надаються бібліотеками Flyway, була вказана у файлі «build.gradle» у батьківському модулі, що дозволяє використати передані попереднім завданням дані для підключення до БД.

Отже, після огляду всього коду цього прикладу, можна перейти до його виконання. Після виконання побудови проекту командою gradle build, IDE підказує, що тепер у скомпільованому проекті зареєстровано завдання startLocalDatabase (Рисунок 5):

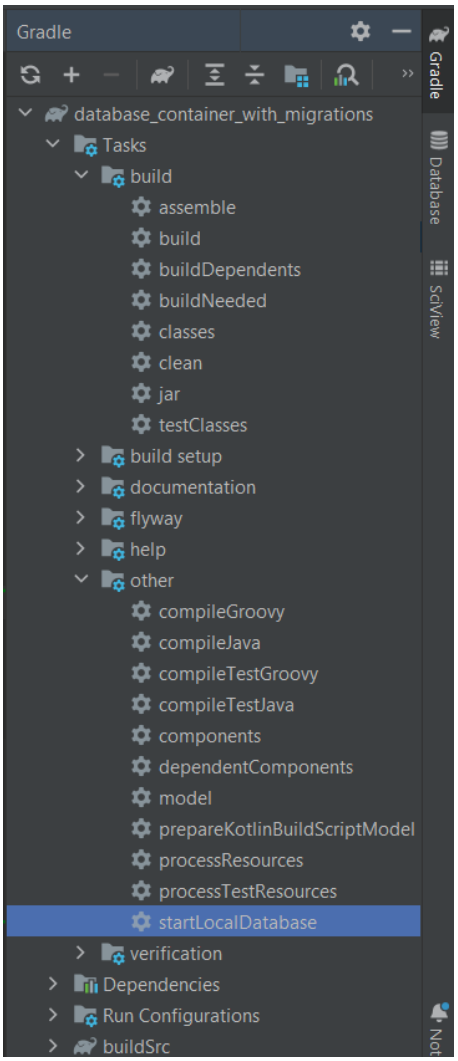


Рисунок 5. Список завдань Gradle зареєстрованих у проекті

При виконанні цього завдання за допомогою команди «gradle setupLocalDatabase», можна побачити у списку запущених контейнерів (Рисунок 6), що перед створенням цільового контейнеру із СУБД Postgres було запущено ще додатковий контейнер із образом «testcontainers/ryuk…». Як можна здогадатись, цей контейнер використовується згаданою бібліотекою для управління всіма процесами запуску та налаштування контейнерів. До речі, навіть у цій ситуації можна помітити використання платформи Docker для виконання системних програм, на цей раз бібліотекою.

На Рисунку 6 зображений список запущених контейнерів в результаті виконання коду, що був описаний вище. Цей список показано у програмі Docker Desktop для ОС Windows, що є графічним клієнтом до платформи Docker.

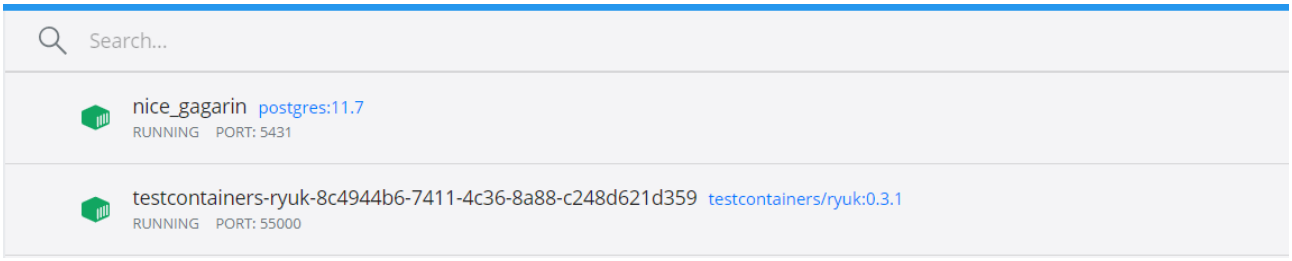


Рисунок 6. Список запущених контейнерів у Docker Desktop

Створивши об’єкт Data Source у IDE, та підключившись до бази даних, що запущена у верхньому контейнері, бачимо, що після виконання завдання startLocalDatabase було автоматично виконано і завдання flywayLocalClean та flywayLocalMigrate – схема та таблиці з міграцій присутні (Рисунок 7):

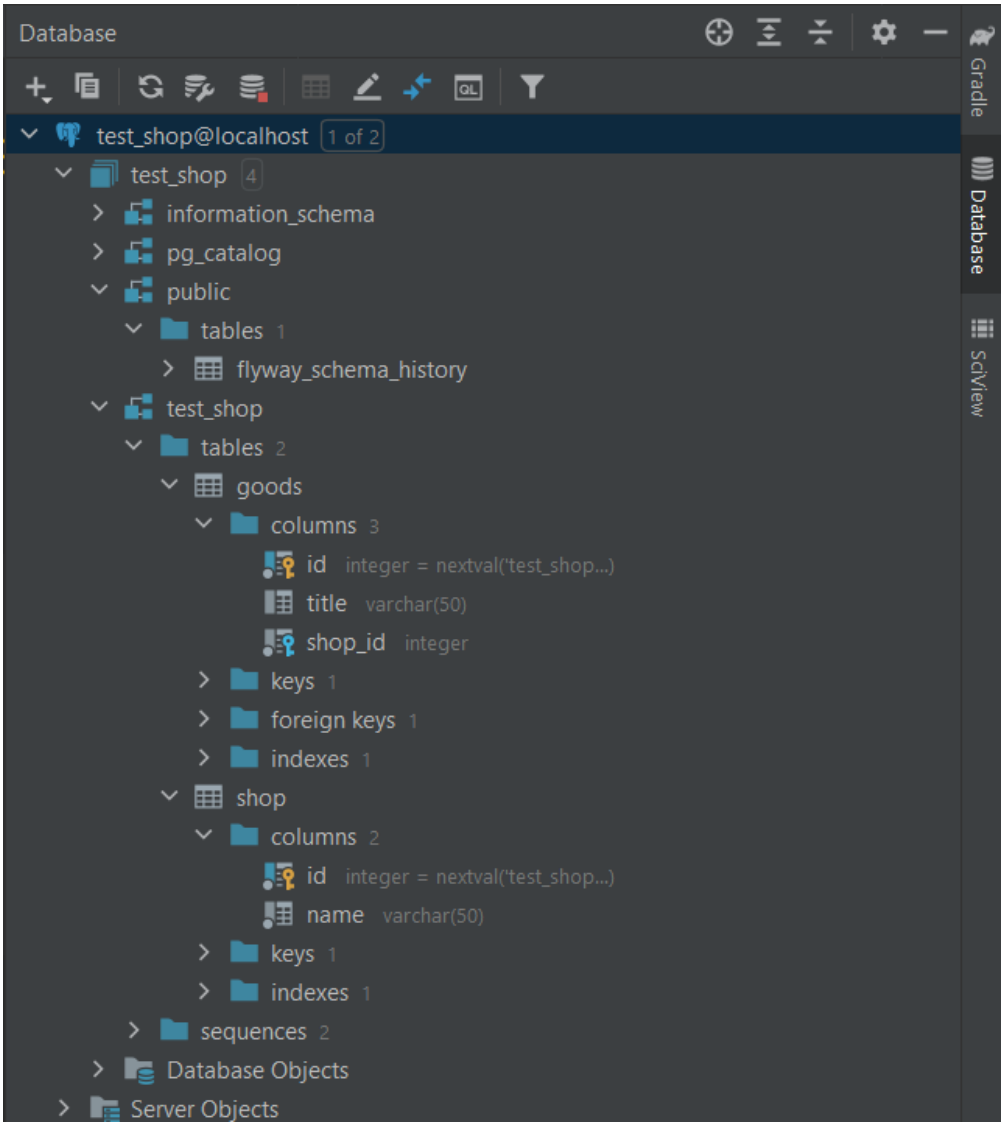
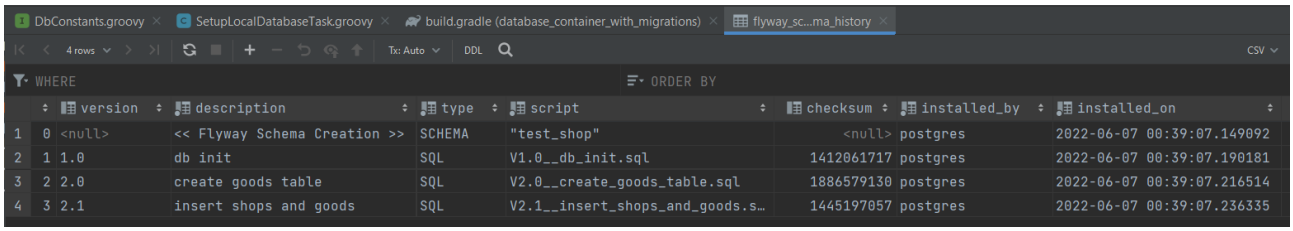


Рисунок 7. Вміст схеми бази даних у контейнері

Про це також говорять дані у таблиці public.flyway_schema_history, яка зберігає метадані про виконання міграцій системою Flyway (Рисунок 8):



DbConstants.groovy | SetupLocalDatabaseTask.groovy | build.gradle (database_container_with_migrations) | flyway_sc...ma_history

4 rows | Tx: Auto | DDL | CSV

WHERE | ORDER BY

	version	description	type	script	checksum	installed_by	installed_on
1	0	<null>	<< Flyway Schema Creation >>	SCHEMA	"test_shop"	<null>	postgres	2022-06-07 00:39:07.149092
2	1	1.0	db init	SQL	V1.0__db_init.sql	1412061717	postgres	2022-06-07 00:39:07.190181
3	2	2.0	create goods table	SQL	V2.0__create_goods_table.sql	1886579130	postgres	2022-06-07 00:39:07.216514
4	3	2.1	insert shops and goods	SQL	V2.1__insert_shops_and_goods.s…	1445197057	postgres	2022-06-07 00:39:07.236335

Рисунок 8. Метадані в таблиці public.flyway_schema_history

Отже, завдання виконано: локальна версія бази даних, що має ту ж схему, що використовується на робочих оточеннях розгорнуто, для чого у проекті в системі Gradle було зареєстровано скрипт написаний на мові Groovy.

3.4 Огляд результатів реалізації другого практичного прикладу

Переходячи до наступного прикладу, необхідно розглянути, що існує і інший варіант вирішення поставленої задачі. Систему flyway можна і не підключати до проекту, а запускати окремий контейнер для неї, який буде робити міграції звертаючись до тієї ж бази даних, що розгортається у своєму окремому контейнері, як у попередньому прикладі, беручи файли міграцій з директорії, яка підключається до нього як volume.

Отже, тепер необхідно два робочих контейнери, один – для системи flyway, а інший – для бази даних. Таку реалізацію можна зробити за допомогою окремого файлу docker-compose.yml, вміст якого представлений на Рисунку 7**:**

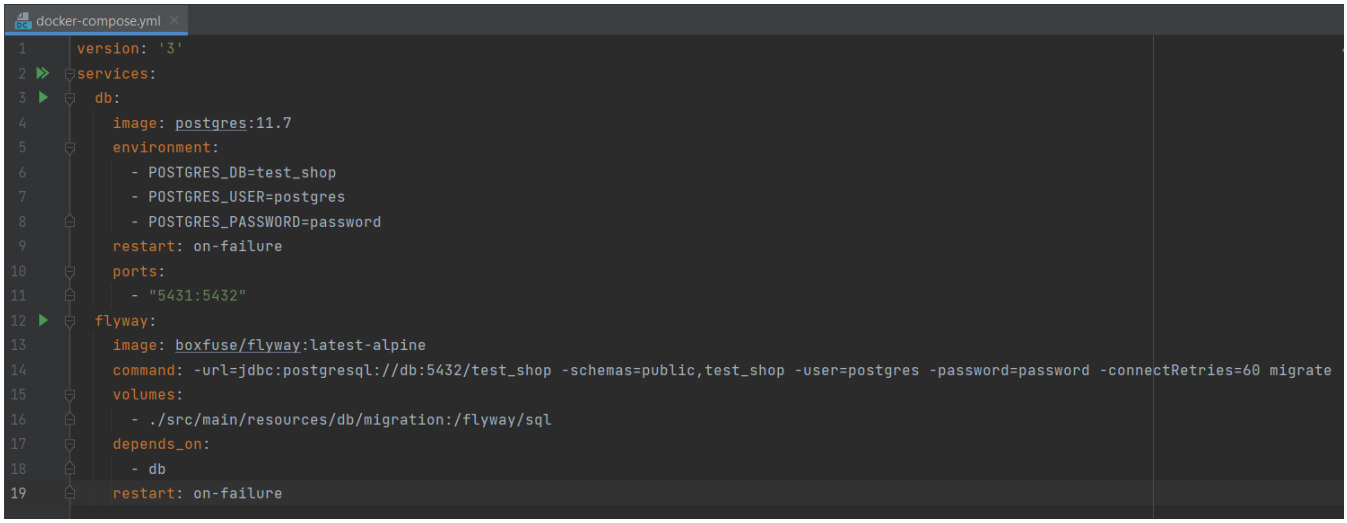


Рисунок 9. Файл docker-compose.yml

За допомогою цього файлу визначаються два контейнери, один з назвою «db», а другий з назвою «flyway». Використовуючи секції image вказується образ для кожного з контейнерів, секція environment визначає необхідні змінні оточення для контейнера. Для контейнера flyway також необхідно вказати команду "*migrate*" із параметрами у секції command, яка буде виконана одразу після запуску контейнера. За допомогою додаткових ключів вказуються URL бази даних, яка міститься у іншому контейнері, а також дані для доступу до неї, аналогічно тому, як це було вказано в попередньому прикладі. Також тут важливим є параметр «-connectRetries=60», який вказує, що цей контейнер буде очікувати успішної відповіді від бази даних 60 секунд, тому що попри те, що контейнер flyway запускатиметься лише після запуску контейнеру db (про що вказує секція depends_on), не гарантує того, що сама СУБД встигне запуститись і налаштуватись на прийом запитів до неї до моменту виконання команди міграції контейнером flyway. Важливим є також те, що оскільки ці два контейнери перебувають в одній стандартній мережі контейнерів, то при підключенні до бази даних необхідно використовувати порт контейнера, а не так званий EXPOSED_PORT, який вказується першим у секції ports (у даному випадку він дорівнює 5431) та використовується для доступу до контейнеру ззовні цієї мережі.

Таким чином, розглянуто ще один варіант запуску контейнеру з базою даних, який міг бути використаний замість бібліотеки Testcontainers, або ж замість виконання наступної команди, що виконує аналогічну дію щодо контейнера db:

“docker run –name db %параметри та змінні оточення% -d postgres:11.7”

Отже, після виконання команди до CLI:

“*docker-compose up*”,

спочатку запускається контейнер db з базою даних postgres, і одразу після нього – контейнер flyway, який починає виконувати команду “*migrate*” з очікуванням робочої СУБД, яка також незабаром починає приймати запити. Після відпрацювання команди контейнером flyway, він автоматично вимикається. Початок цього процесу можна спостерігати у логах терміналу, на Рисунку 10:

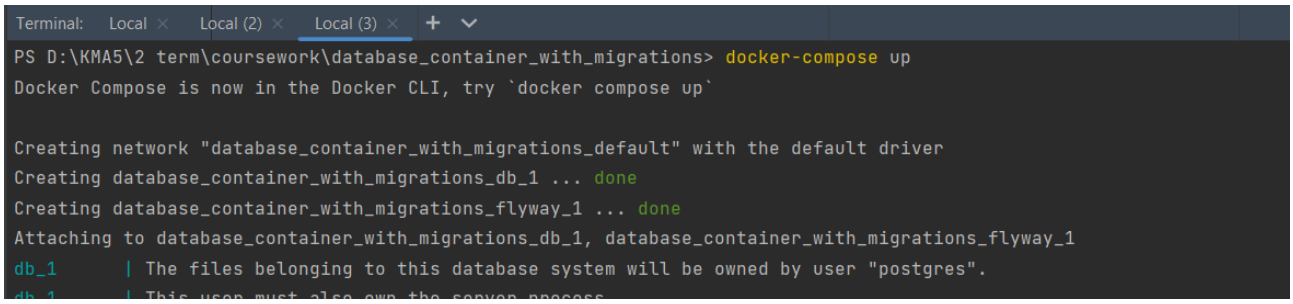


Рисунок 10. Логи під час виконання команди "docker-compose up"

Результат також можна спостерігати в інтерфейсі Docker Desktop (Рисунок 11):

Рисунок 11. Результат виконання команди "docker-compose up"

Відповідно, при підключенні до цього контейнеру, можна побачити абсолютно ідентичну до попереднього прикладу схему бази даних (Рисунки 12 та 13):

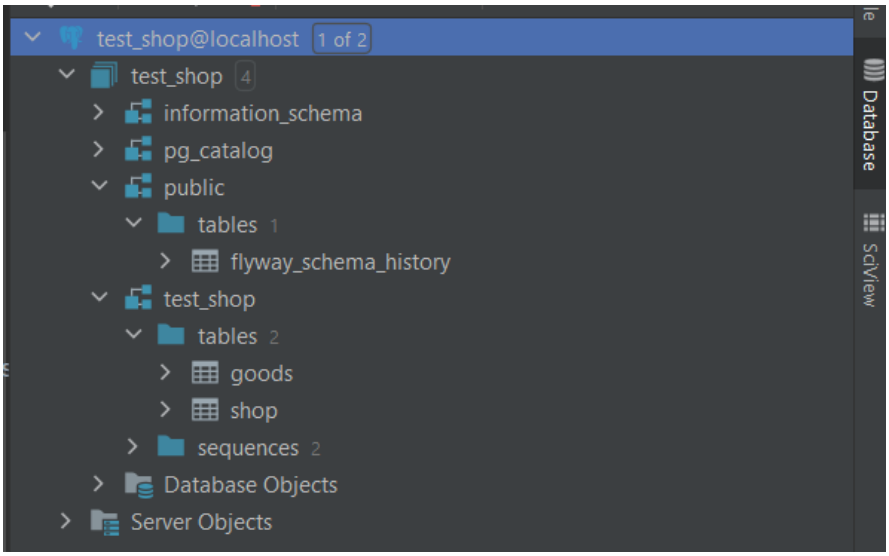


Рисунок 12. Схема бази даних у контейнері db

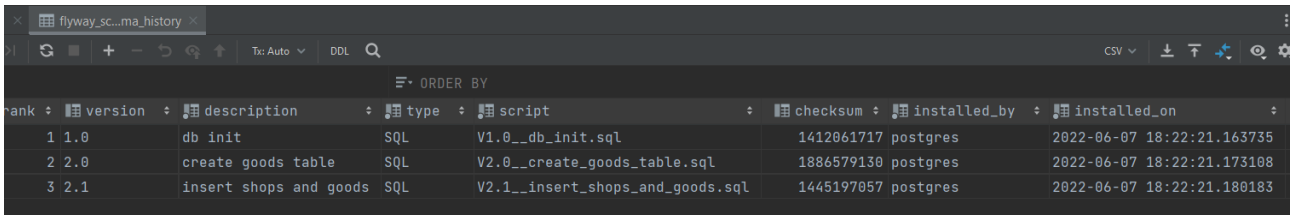


Рисунок 13. Вміст таблиці flyway_schema_history у контейнері db

Відмінність цього методу запуску та виконання міграцій від попереднього заключається у тому, що таке рішення являється менш гнучким в плані використання додаткових користувацьких команд flyway у майбутньому: за відсутності встановленої системи flyway в поточному середовищі, без додаткового запуску контейнера flyway, або використання віддаленого середовища, де система встановлена, для виконання таких команд не обійтися. У будь-якому разі, завдання також виконано.

3.5 Аналіз можливостей роботи з Docker на основі практичного прикладу

Аналізуючи розглянуті приклади, можна сказати що з платформою Docker можна знаходити безліч варіантів виконання різних задач. Як було розглянуто,

навіть бібліотека Testcontainers використовує додатковий контейнер для своєї роботи, таким чином позбавляючи користувача необхідності встановлювати якісь додаткові інструменти та програмне забезпечення на свою локальну систему, що часто може викликати зайві труднощі у нестандартних оточеннях та версіях операційної системи.

У даному випадку, за допомогою контейнеризації було вирішено задачу зручного та ефективного запуску бази даних, що часто використовується розробниками під час їх буденного робочого процесу. Такий контейнер можна швидко вимкнути та звільнити ресурси системи, він не потребує встановлення бази даних на локальний пристрій, а робота з підключенням до бази даних (що стосується і міграцій) не відрізняється від варіанту встановлення такої бази даних у середовищі. Більше того, використання платформи Docker для вирішення даної задачі дозволяє запускати такий контейнер і під час процесів CI/CD, для проведення додаткового інтеграційного тестування застосунку.

ВИСНОВКИ

Отже, результатом роботи стало дослідження концепцій віртуалізації та контейнеризації, архітектури платформи Docker, способів використання платформи для виконання різних типів задач. На практичних прикладах доведено зручність використання платформи розробниками програмного забезпечення, зокрема при запуску локальної версії бази даних для подальшої роботі з нею у вигляді тестування, випробувань коду, відтворення «багів» та ін.

Відповідно до результатів цього дослідження, серед переваг використання даної платформи можна виділити:

- незалежність від операційної системи: для роботи контейнерів необхідна наявність лише середовища, в якому можна запустити сам Docker;
- ефективне використання ресурсів пристрою та системи;
- зручне управління життєвим циклом контейнерів, що відкриває можливості для їх оркестрації;
- велика різноманітність варіантів застосування контейнеризації у різних галузях комп’ютерних технологій;
- наявність інструментів та API які дозволяють легко взаємодіяти з платформою;
- постійна підтримка розробників платформи, та інші.

Як висновок, можна із впевненістю сказати, що платформа Docker стала однією з провідних ланок галузі IT, а популярність її використання доводить, що її зручність, ефективність та функціонал вже роками завойовує серця інженерів програмного забезпечення.

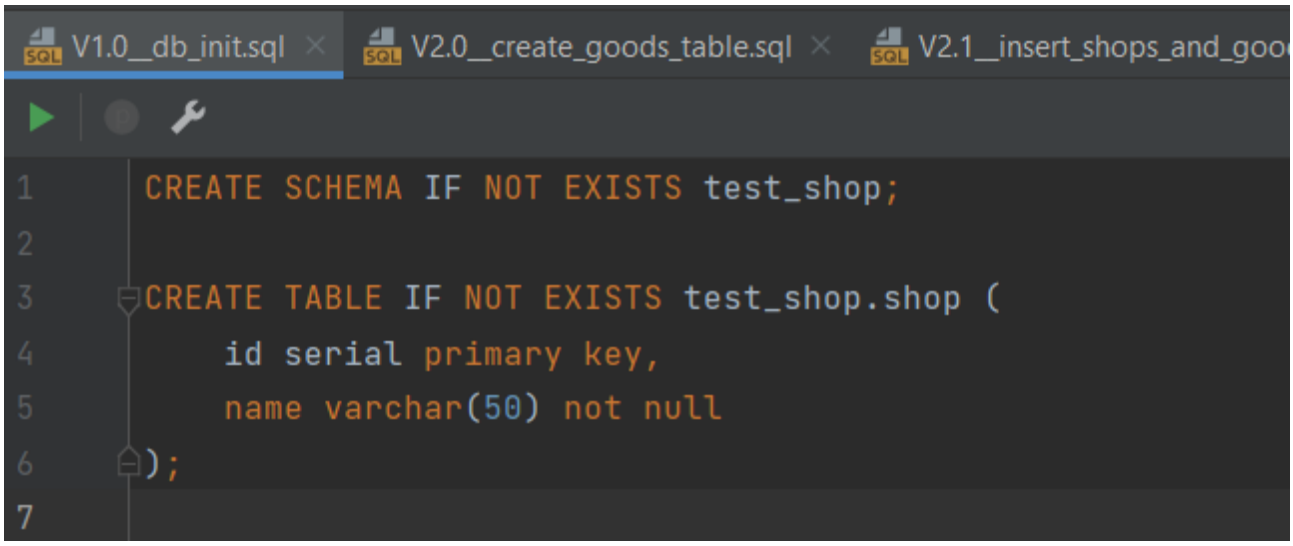
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Docker.com – Docker Overview [Електронний ресурс] https://docs.docker.com/get-started/overview/
2. Insights Stackoverflow – Developer Survey 2021 [Електронний ресурс] https://insights.stackoverflow.com/survey/2021#most-popular-technologies-tools-tech
3. Docker.com – Docker Use Cases [Електронний ресурс] https://www.docker.com/use-cases/
4. Oztec – Virtualization: concepts and terminologies, 2018 [Електронний ресурс] https://ostec.blog/en/general/virtualization-concepts-and-terminologies/
5. GeeksforGeeks – Virtualization In Cloud Computing and Types – Namrata Bisht, 2021 [Електронний ресурс] https://www.geeksforgeeks.org/virtualization-cloud-computing-types/
6. Onbiz – Детально про віртуалізацію: типи, переваги, рішення [Електронний ресурс] https://onbiz.biz/about-virtualization/
7. Citrix – What is containerization [Електронний ресурс] https://www.citrix.com/solutions/app-delivery-and-security/what-is-containerization.html
8. Section – Why is Docker so Popular – Ahmad Mardeni, 2021 [Електронний ресурс] https://www.section.io/engineering-education/why-is-docker-so-popular/
9. Dzone – Docker Containers and Kubernetes: An Architectural Perspective – Rabi Prasad Padhy, 2018 [Електронний ресурс] https://dzone.com/articles/docker-containers-and-kubernetes-an-architectural
10. JRebel – Using BuildSrc for Custom Logic in Gradle Builds, 2017 [Електронний ресурс] https://www.jrebel.com/blog/using-buildsrc-custom-logic-gradle-builds

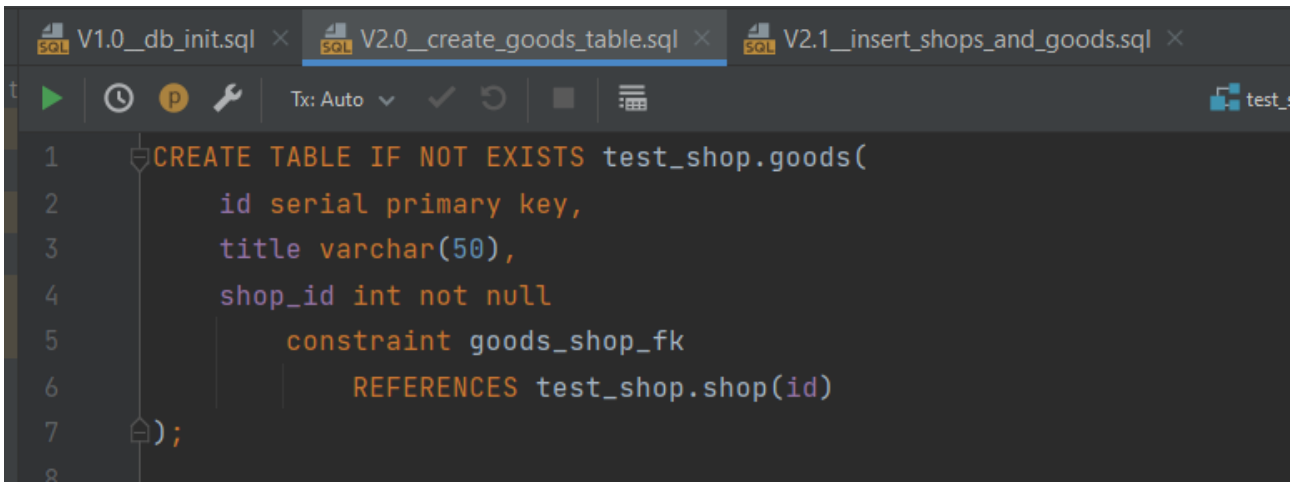
11. Github – flyway-docker – rd-buildmonkey [Електронний ресурс] https://github.com/flyway/flyway-docker
12. Flywaydb – Gradle Plugin [Електронний ресурс] https://flywaydb.org/documentation/usage/gradle/

ДОДАТКИ

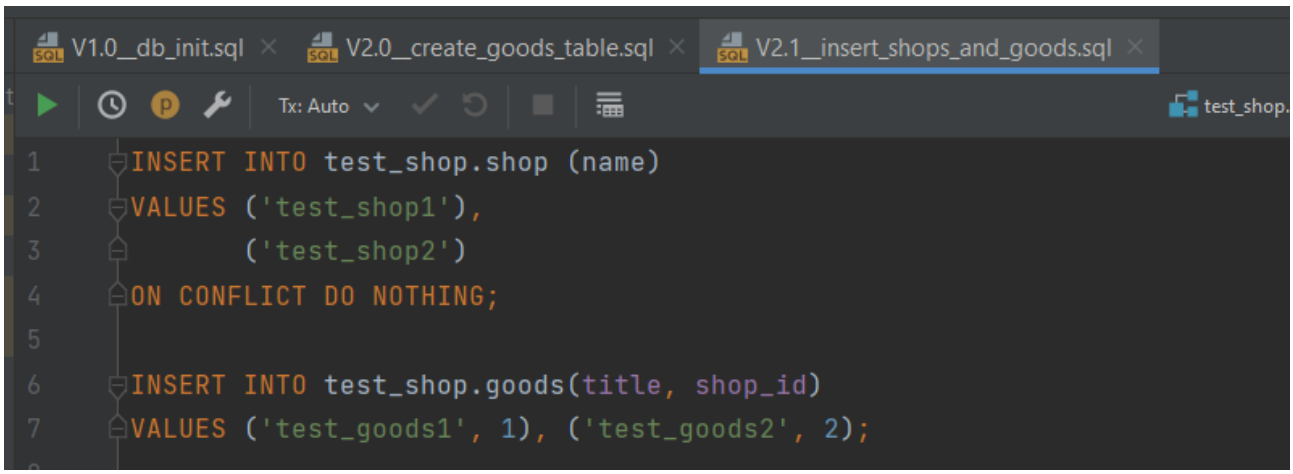
Додаток 1. Файл міграції «V1.0__db_init.sql»



Додаток 2. Файл міграції «V2.0__create_goods_table.sql»



Додаток 3. Файл міграції «V2.1__insert_shops_and_goods.sql»



Додаток 4. Файл конфігурації батьківського модуля «build.gradle»

build.gradle (database_container_with_migrations)

```
import localdatabase.DbConstants
import localdatabase.SetupLocalDatabaseTask
import org.flywaydb.gradle.task.FlywayCleanTask;
import org.flywaydb.gradle.task.FlywayRepairTask;
import org.flywaydb.gradle.task.FlywayMigrateTask;

plugins {
    id 'idea'
    id 'groovy'
    id 'java'
    id 'org.flywaydb.flyway' version '8.5.12'
}

ext {
    localDbSchemas = ['public', 'test_shop']
}

group 'org.example'
version '1.0-SNAPSHOT'
final String localDbUrl = System.getenv('DB_LOCAL_URL') ?: DbConstants.DB_URL
final String localDbUsername = System.getenv('DB_LOCAL_USERNAME') ?: DbConstants.DB_USERNAME
final String localDbPassword = System.getenv('DB_LOCAL_PASSWORD') ?: DbConstants.DB_PASSWORD

repositories {
    mavenLocal()
    mavenCentral()
}

dependencies {
    implementation localGroovy()
    implementation 'org.postgresql:postgresql:42.3.4'
    implementation 'org.flywaydb:flyway-core:8.5.12'
    implementation 'log4j:log4j:1.2.17'
}

task flywayLocalClean(type: FlywayCleanTask) {
    url = localDbUrl
    user = localDbUsername
    password = localDbPassword
    schemas = localDbSchemas
    cleanDisabled = false
}
```

Прим. (продовження на наступній сторінці)

```
task flywayLocalMigrate(type: FlywayMigrateTask) {
    url = localDbUrl
    user = localDbUsername
    password = localDbPassword

    schemas = localDbSchemas
    defaultSchema = 'public'

    baselineVersion = 1.0
    baselineOnMigrate = true

    shouldRunAfter(flywayLocalClean)
}
task flywayLocalRepair(type: FlywayRepairTask) {
    url = localDbUrl
    user = localDbUsername
    password = localDbPassword
    schemas = localDbSchemas
}

tasks.register('startLocalDatabase', SetupLocalDatabaseTask) { Task t ->
    t.finalizedBy(flywayLocalClean, flywayLocalMigrate)
}
```

Додаток 5. Клас SetupLocalDatabaseTask

DbConstants.groovy | SetupLocalDatabaseTask.groovy

```groovy
class SetupLocalDatabaseTask extends DefaultTask {

    private static final int CONTAINER_EXPOSED_PORT = 5432

    /*
     * Creates postgresql container if it not exists, provides db credentials to flyway tasks
     */
    @TaskAction
    void initializeContainerIfNotExists() {
        configureTestContainers()
        Consumer<CreateContainerCmd> hostPortBinding = { CreateContainerCmd c ->
            c.hostConfig.withPortBindings(new PortBinding(Ports.Binding.bindPort(DbConstants.HOST_PORT),
                    new ExposedPort(CONTAINER_EXPOSED_PORT)))
        }
        try {
            new PostgreSQLContainer( dockerImageName: 'postgres:11.7')
                    .withUsername(DbConstants.DB_USERNAME)
                    .withPassword(DbConstants.DB_PASSWORD)
                    .withDatabaseName(DbConstants.DATABASE)
                    .withExposedPorts(CONTAINER_EXPOSED_PORT)
                    .withCreateContainerCmdModifier(hostPortBinding)
                    .withReuse( reusable: true)
                    .start()
        } catch (Exception e) {
            String rootCauseMsg = ExceptionUtils.getRootCauseMessage(e)
            if (!rootCauseMsg.contains('port is already allocated')) {
                throw e
            }
        }
        configureFlywayTasks()
    }

    /*
     * Loads file ".testcontainers.properties", creates a new one if it not exists.
     * Sets the property 'testcontainers.reuse.enable' to true,
     * this configures Testcontainers to reuse the existing containers.
     */
    private void configureTestContainers() {
        File propsFile = new File(
                System.getProperty('user.home'), child: '.testcontainers.properties')
        configurePropertyForFile(propsFile)
    }

    private void configurePropertyForFile(File file) {
        Properties props = new Properties()
        if (file.exists()) {
            file.newInputStream()
                    .withCloseable { BufferedInputStream inputStream -> props.load(inputStream) }
        }
        if (props['testcontainers.reuse.enable'] != 'true') {
            ant.propertyfile(file: file) {entry(key: 'testcontainers.reuse.enable', value: 'true')}
        }
    }
```

```
70      /*
71       * Provides db credentials to flyway tasks
72       */
73      private void configureFlywayTasks() {
74          def provideDatabaseCredentials : Closure<String> = {
75              url = DbConstants.DB_URL
76              user = DbConstants.DB_USERNAME
77              password = DbConstants.DB_PASSWORD
78          }
79          project.tasks.named( s: 'flywayLocalClean', provideDatabaseCredentials)
80          project.tasks.named( s: 'flywayLocalMigrate', provideDatabaseCredentials)
81      }
82  }
```

Додаток 6. Інтерфейс DbConstants

DbConstants.groovy | SetupLocalDatabaseTask.groovy

```
1   package localdatabase
2
3   interface DbConstants {
4
5       public static final String DB_URL = String.format(
6               'jdbc:postgresql://localhost:%d/test_shop?serverTimezone=UTC', HOST_PORT);
7
8       public static final int HOST_PORT = 5431
9
10      public static final String DB_USERNAME = 'postgres'
11
12      public static final String DB_PASSWORD = 'password'
13
14      public static final String DATABASE = 'test_shop';
15  }
16
```