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

Кафедра мережних технологій факультету інформатики

ПОБУДОВА БАГАТОРІВНЕВОГО ВЕБ-ЗАСТОСУВАННЯ НА DOCKER-ПЛАТФОРМІ

Текстова частина до курсової роботи

за спеціальністю «Інженерія програмного забезпечення» 121

Керівник курсової роботи

к.т.н. Черкасов Д.І.

_____

“____” _____ 2021 р.

Виконав студент БП ІПЗ-3

Круковська Я.В.

“____” _____ 2021 р

Київ 2021

ЗМІСТ

Календарний план виконання курсової роботи

Тема: Побудова багаторівневого веб-застосування на Docker-платформі

№ п/п	Назва етапу курсового проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на курсову роботу	жовтень 2020 р.	
2.	Огляд літератури за темою роботи	листопад – грудень 2020 р.	
3.	Огляд технологій потрібних для реалізації практичної частини	січень 2021 р.	
4.	Написання теоретичної частини роботи	лютий – березень 2021 р.	
5.	Створення практичної частини роботи	березень – травень 2021 р.	
6.	Коригування виконаної роботи	травень 2021 р.	
7.	Створення презентації	травень 2021 р.	
8.	Захист курсової роботи	Після 17.05.2021	

Студент Круковська Я.В.

Керівник Черкасов Д.І.

“_____” _____

ВСТУП

Нині стрімко зростає складність застосувань, збільшуються вимоги до навантаження на них та їх відмово стійкості. Крім того, необхідна можливість швидко додавати нові можливості і змінювати ті, що існують. Всі ці вимоги унеможливлюють використання монолітної архітектури. Одним із рішень є архітектурний патерн - багаторівневі застосування. Найпопулярніший варіант – це трирівнева архітектура, що дозволяє розділити користувацький (клієнтський) інтерфейс, бізнес логіку та базу даних, над якими можуть працювати окремі команди, або навіть різні постачальники.

Використання цього рішення дозволяє спростити розробку та підтримку функціоналу, але не вирішує проблему надійності, масштабування та швидкого розгортання. Крім того, адміністрування залишається доволі складним. Проте це можна вирішити за допомогою віртуалізації середовища виконання застосувань. Цей підхід також дозволяє позбутися необхідності мати власне апаратне забезпечення, використовуючи обчислювальні ресурси, ресурси для зберігання даних і засоби адміністрування як сервіси, що надаються іншими постачальниками. Також не можна не згадати зменшення вартості підтримки інфраструктури – власнику застосування не потрібно мати резервні обчислювальні або дискові ресурси, він платить лише за те, що використовує.

Актуальність цієї роботи визначається актуальністю описаних вище очікувань від застосунків, адже розглянуті в роботі рішення нині активно використовуються для того, щоб задовольнити ці вимоги.

Метою цієї роботи є огляд багаторівневої архітектури та порівняння застосунків з різною кількістю рівнів. Також в текстовій частині наведений порівняльний аналіз різних підходів віртуалізації, а саме: контейнеризації та віртуальних машин.

Практичною частиною роботи було визначено розробку власного трирівневого веб-застосування – книжкового сервісу. В функціональні вимоги входять можливість реєстрації, перегляду каталогу книг та відгуків до них, додавання нових книг та відгуків залежно від ролі користувача. Головною вимогою до проекту є розгортання його на платформі Docker. Для розробки користувацького інтерфейсу було використано фреймворк Angular; бізнес логіка була написана на мові програмування Java з використанням Spring Boot та Hibernate, а базою даних була обрана PostgreSQL. Кожен рівень було поміщено в окремий контейнер.

1. Огляд існуючих рішень

Для огляду багаторівневої архітектури потрібно розуміти різницю між поняттями «шар» (layer) та «рівень» (tier), які часто перекладають однаково та використовують як синоніми.

«Шаром називають функціональну складову програмного забезпечення, поки рівень – це функціональна складова програмного забезпечення, для виконання якої відведена інша інфраструктура. Наприклад, програма «Контакти» на мобільному телефоні має три шари, але є одноріневним застосунком, бо всі три шари запускаються на телефоні» [1].

Аналогічна проблема з термінами виникла в кондитерській сфері. У тортів існують шари, що відрізняються смаком чи кольором, але мають однаковий розмір. Разом вони об’єднуються у рівні, що відрізняються одне від одного розміром. Прирівняємо до архітектури застосунків – шари відрізняються «начинкою» - тим, чим вони займаються, і об’єднуються в рівні – блоки, що відрізняються фізично.

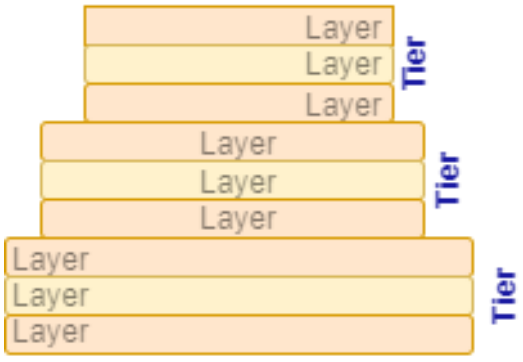

Рисунок 1.1 - Шари та рівні на прикладі торту

Отже, правильно використовувати термін «шар» для позначення логічного розподілення застосунку, а «рівень» для опису фізичного розподілення компонентів.

1.1. Огляд типів архітектури

1.1.1. Одноріvнева архітектура

Одноріvнева архітектура є найпростішою для розуміння та реалізації, бо у ній всі необхідні компоненти запускаються на одному рівні. Таке архітектурне рішення ще називають монолітом.

Проте один рівень не означає, що застосування не може бути розділеним на шари. Одноріvнева архітектура передбачає бізнес логіку, базу даних, користувацький інтерфейс чи будь-який інший шар, який може знадобитися.

Прикладом такого рішення є прості застосунки на телефоні, або ж продукти Microsoft Office. Якщо розглянути конкретно Microsoft Word, то можна побачити, що частиною одного застосунку є користувацький інтерфейс, бізнес логіка (правила перенесення слів та переходу на нову сторінку), доступ до файлів та керування даними документу [2].

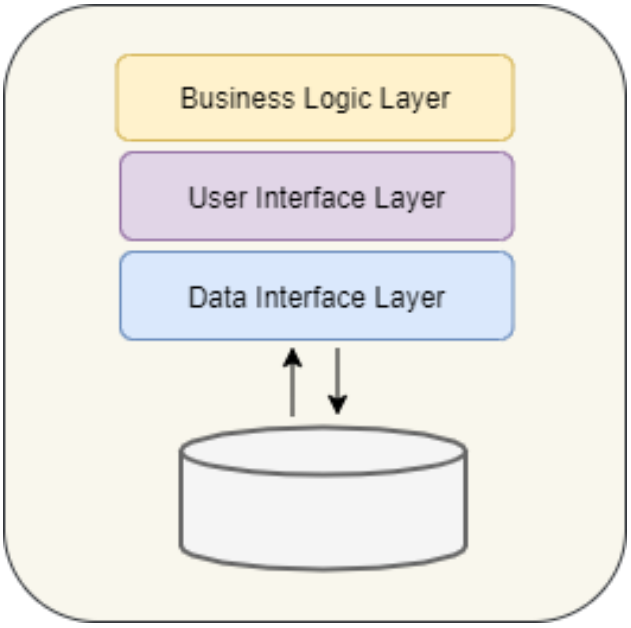

Рисунок 1.1.1.1 - Приклад структури одноріvневного застосування

1.1.2. Дворівнева архітектура

У дворівневій архітектурі з’являється новий рівень, що відокремлює дані. Один фізичний компонент містить у собі шари користувацького інтерфейсу та бізнес логіки, а інший складається з шару даних.

В застосунках з такою архітектурою зазвичай шари бізнес правил та користувацького інтерфейсу є частиною клієнтського застосунку, а маніпуляція даними відбувається в окремій фізично відокремленій системі. Прикладом такої системи може бути SQL сервер.

Ще одним варіант дворівневої архітектури - де бізнес правила виконуються на стороні рівня даних. В таких випадках процедури маніпулювання даними зберігаються в самій базі [2].

Збережена процедура – це підготовлений SQL код, який можна зберегти для повторного використання. Це зручно за наявності якогось SQL запиту, який доводиться постійно писати заново [3].

Клієнтська сторона може явно викликати збережену процедуру, яка буде виконана на сервері. Запустити її може також і якась умова. Наприклад, умова «надіслати повідомлення про критичне зменшення балансу на рахунку, якщо сума менше 50», яка запустить процедуру надсилання нотифікації користувачеві [2].

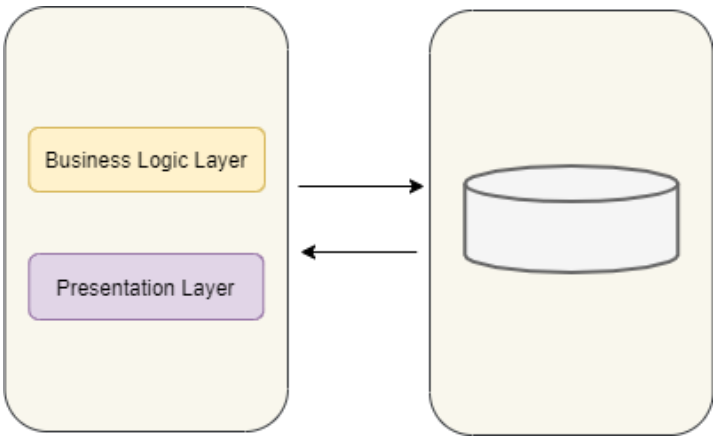

Рисунок 1.1.2.1 - Приклад структури дворівневого застосування

1.1.3. Трирівнева архітектура

Трирівнева архітектура є найпопулярнішою і найпоширенішою. Від дворівневої її відрізняє відділення бізнес логіки від користувацького інтерфейсу ще один рівень.

Розглянемо детально кожен рівень:

1. Рівень відображенні: це шар користувацького інтерфейсу та шар комунікації. За допомогою останнього можна «спілкуватися» із застосунком. Головними цілями цього рівня є показ даних, а також збирання інформації від користувача. Він може бути представлений у вигляді веб, десктоп або мобільного застосунку.
2. Рівень застосунку (також називається логічним або середнім рівнем): є зв’язком між рівнями відображення та даних. В ньому отримана інформація обробляється та відправляється до необхідного рівня. Тобто він забезпечує застосування функціональними можливостями і поєднує UI з базою даних.
3. Рівень даних: виконує роль зберігання та керування даними. Представляється у вигляді бази даних.

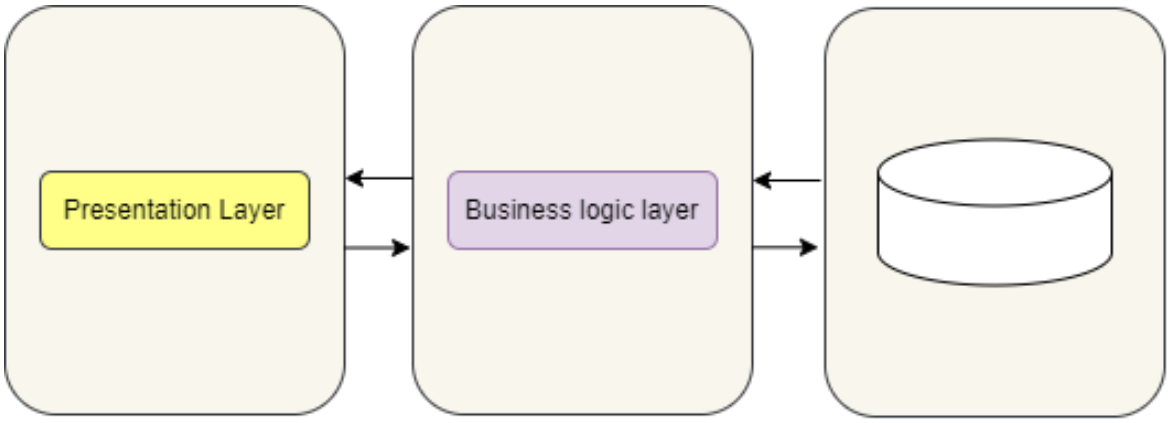

Рисунок 1.1.3.1 - Приклад структури трирівневого застосування

1.1.4. N-рівнева архітектура

Архітектура, в якій кількість рівнів перевищує 3 зустрічається зрідка. Пояснюється це ускладненням підтримки таких застосунків. Крім того, трирівнева архітектура цілком зручно і зрозуміло розподіляє рівні.

При розгляді архітектури з чотирьох рівнів з'являється поняття «тонкий клієнт» («thin client»), що розміщується на новому «клієнтському» рівні.

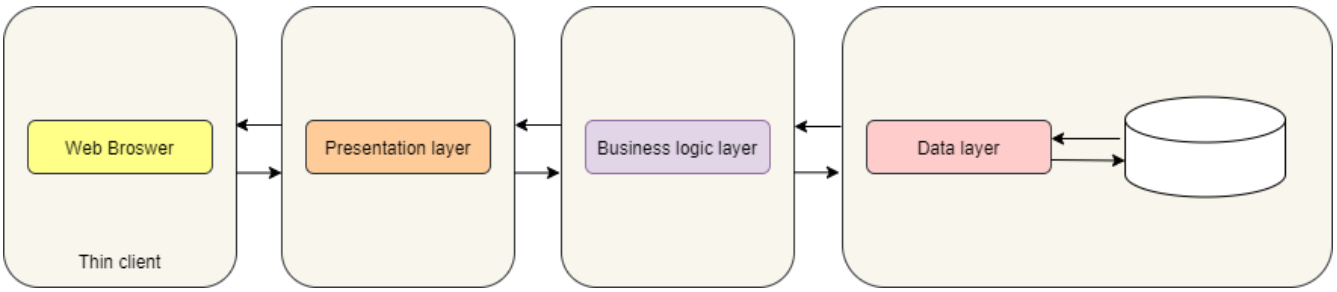

Рисунок 1.1.4.1 - Приклад структури чотирирівневого застосування

Клієнт називається тонким, якщо у нього майже повністю або повністю відсутня бізнес логіка. В іншому випадку він товстий. Найкращим прикладом є веб-браузер, що може підключатися до повністю інакших застосунків, про які не знає, і все одно надавати зручний для користувача інтерфейс [4].

Іншими словами це комп'ютер чи програма, що чиї можливості сильно залежать від інших застосунків.

Ще один варіант архітектури з чотирьох рівнів полягає в розбитті застосунку на рівні: відображення, сервіс даних, бізнес логіка, доступ до даних. З рівнями відображення та бази даних ми вже знайомі з опису інших типів архітектури.

Рівень сервісу даних (Data Service Layer, DSL) – забезпечує захищеність даних від клієнта. Головна задача рівня – посилання на дані, які обробив рівень бізнес логіки та передача їх до рівня відображення.

Рівень доступу до даних (Data Access Layer, DAL) – працює незалежно від інших рівнів і надає доступ до бази даних та операцій над нею [5].

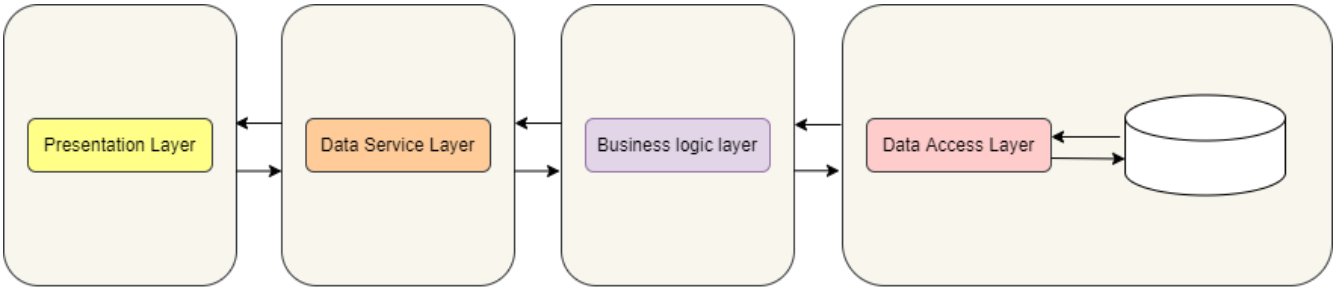

Рисунок 1.1.4.2 - Альтернативний варіант чотирирівневого застосування

Як приклад п’ятирівневої архітектури пропонуються ті самі рівні, але відображення розділене на два – саме представлення (зовнішній вигляд) та взаємодія користувача з інтерфейсом.

1.1.5. Порівняльний аналіз

Тип архітектури	Переваги	Недоліки
Однорівнева	- Легко реалізовувати та розгортати - Швидка для використання однією людиною	- Відсутність масштабування - Важче підтримувати таку монолітну структуру
Дворівнева	- Легко реалізувати та підтримувати - Краще розподілення навантаження на рівні - Рівні легко і швидко спілкуються між собою	- Не може підтримувати велику кількість користувачів: при їх збільшенні буде падати продуктивність
Трирівнева	- Легко масштабувати - Швидка розробка, бо кожним рівнем може займатися окрема команда. - Більша надійність застосунку, бо проблеми в одному рівні не будуть повністю виводити з ладу інші	- Рівням складніше спілкуватися - Важче розробляти та підтримувати
N-рівнева	- Переваги трирівневої архітектури, але продуктивність ще вища	- Набагато важче розробляти та підтримувати

Таблиця 1 - порівняння рівневих архітектур

Отже, кожен наступний рівень архітектури має кращу продуктивність за попередній, але є складнішим для реалізації та підтримки. Трирівнева архітектура є найоптимальнішою, бо її легше підтримувати за n-рівневу, а за рахунок її ефективності та масштабування вона краще за архітектури з меншою кількістю рівнів.

1.2. Огляд підходів віртуалізації

Віртуалізація використовує програмне забезпечення для того, щоб створити абстрактний рівень у апаратного забезпечення. Це дозволяє елементам апаратного забезпечення одного комп'ютера - процесору, пам'яті, тощо – ділитися на множину віртуальних комп'ютерів, які називають віртуальними машинами (virtual machines). У кожної віртуальної машини є власна операційна система, що веде себе як незалежний комп'ютер, хоча вона й запущена на частині реального апаратного забезпечення [6].

Віртуалізація набрала популярності як рішення для двох проблем. Разом з нею компанії отримали можливість розділяти їх сервери і запускати застарілі програми на різних версіях різних операційних систем. Завдяки цьому деякі сервери стали ефективнішими, що призвело до зменшення вартості їхньої покупки, встановлення та підтримки [7].

Перевагами віртуалізації є:

- Збільшення ефективності ресурсів: завдяки віртуалізації можна максимально ефективно використовувати можливості серверів. Раніше, коли кожному застосунку потрібен був окремий сервер, багато його можливостей не використовувалися.
- Легше управління: для управління віртуальними машинами можна створити автоматичні сервіси.
- Мінімальний простій: віртуалізація дозволяє паралельно запускати декілька машин, щоб на випадок проблем з однією з них, інші продовжувати працювати. З фізичними серверами це було набагато важче реалізувати.
- Швидше забезпечення: купівля, встановлення і налаштування апаратного забезпечення для кожного застосунку займає чимало часу. Оскільки багато віртуальних машин використовують одне апаратне забезпечення, це економить багато часу [6].

Існує чимало типів віртуалізації. Серед них: віртуалізація даних, десктоп віртуалізація, серверна віртуалізація, віртуалізація операційних систем, віртуалізація мережевих функцій, CPU/GPU віртуалізація, хмарна віртуалізація, Linux віртуалізація та інші.

1.2.1. Віртуальні машини

Віртуальна машина – це технологія для побудови віртуальних середовищ. По суті це емуляція фізичного комп'ютера. Дуже часто віртуальні машини ототожнюють з віртуалізацією в цілому.

Така машина не може напряму спілкуватися зі с комп'ютером, на якому запущена. Для цього використовується гіпервізор, який також називають монітором віртуальних машин. Саме він забезпечує розподіл ресурсів і перевіряє, що всі віртуальні машини не заважають роботі одне одного, використовуючи чужі ресурси [8].

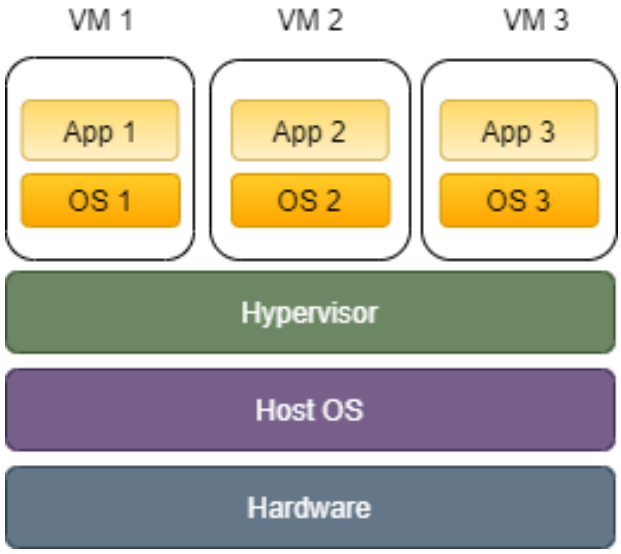

Рисунок 1.2.1.1 - Приклад віртуалізації за допомогою віртуальних машин

Гіпервізори бувають двох типів:

1) Автономні: вони встановлені безпосередньо на фізичній машині. Цей вид гіпервізорів є популярнішим за рахунок підвищеної безпеки та нижчої латентності у порівнянні з другим типом.
2) На основі операційної системи: у цього типу між фізичним сервером та гіпервізером є шар ОС. Зазвичай використовується для візуалізації робочих місць користувачів [9].

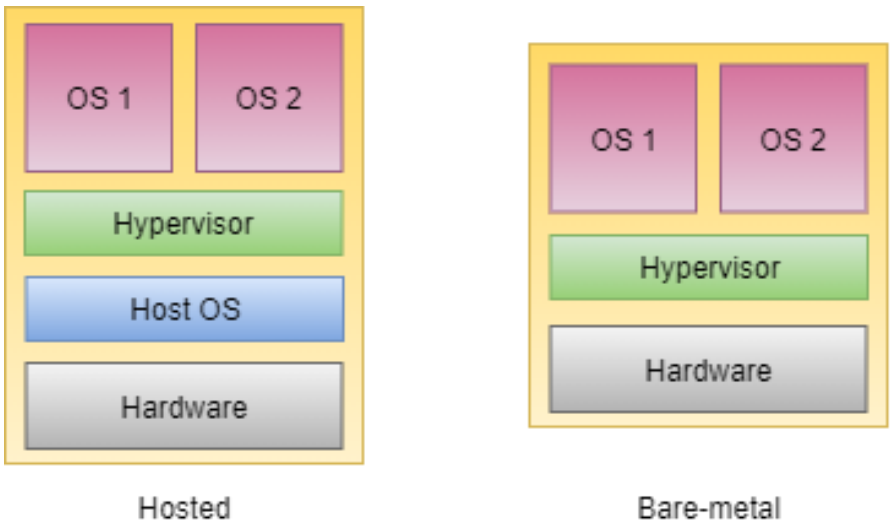

Рисунок 1.2.12 - Порівняння гіпервізорів

Сфери використання віртуальних машин:

- Хмарні обчислення
- Тестування нових операційних систем: віртуальна машина дозволяє тестувати нові ОС без впливу на основну ОС
- Дослідження небезпечного програмного забезпечення: віртуальна машина дозволяє не заражати реальні машини
- Безпечний пошук: схоже на попередній пункт. Віртуальна машина дає змогу переходити на будь-який сайт, не хвилюючись про можливі віруси. Можна робити знімок файлової системи і повертатися до нього після кожної пошукової сесії. Для цього потрібен гіпервізор другого типу.
- Запуск застосунків, що не поєднується: надає можливість користуватися програмами, що недоступні для певної операційної системи, але необхідні користувачу.
- Підтримка DevOps: така віртуалізація дає змогу налаштувати шаблони з віртуальних машин з певними налаштуваннями для процесів тестування і розробки [10].

1.2.2. Контейнеризація

Альтернативою віртуалізації за допомогою віртуальних машин є контейнеризація. Це інкапсуляція застосунку в окремий контейнер зі своїм робочим середовищем.

Контейнер – це одиниця програмного забезпечення, в які спакований код, бібліотеки, залежності так, що вони можуть бути запущеними будь-де: десктопі, хмарі, тощо. Вони маленькі та швидкі, бо на відмінну від віртуальних машин не включають у кожен свій екземпляр гостьову операційну систему. Замість цього вони використовують операційну систему хоста [11].

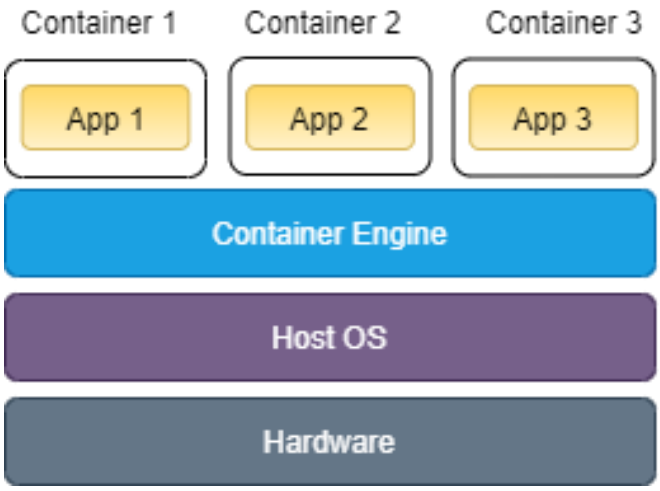

Рисунок 1.2.2.1 - віртуалізації за допомогою контейнерів

Для керування, розгортання, масштабування і підтримки контейнерних застосунків часто використовуються оркестранти (orchestrator). Прикладом таких інструментів є Kubernetes та Docker Swarm.

Оркестрування, зазвичай, починається зі створення конфігураційного файлу, де вказується:

- Де шукати контейнери
- Як налаштовувати мережу між контейнерами

- Де зберігати логи

Коли настає час розгортати новий контейнер в кластер, оркестрант планує це розгортання та шукає найкращий для нього хост, враховуючи наперед вказані обмеження (наприклад: CPU або доступність пам’яті).

Коли контейнер запущений, оркестрант керує його життєвим циклом згідно з вказаними налаштуваннями (їх можна знайти, наприклад, в Dockerfile) [12].

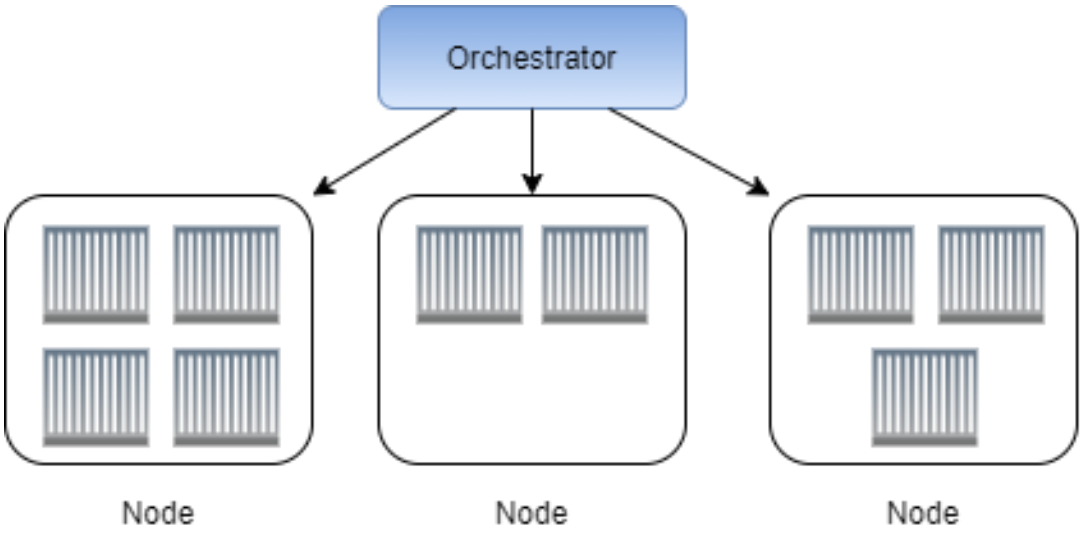

Рисунок 1.2.2.3 - Приклад схеми роботи оркестрантів

Згідно з опитуванням The New Stack, “Primary method of managing/orchestrating containers”, 45% опитуваних користуються саме оркестрантами. 16% використовують для цього shell скрипти або кастомізації, 13% - інструменти для управління конфігураціями, 12% - контейнерами як сервісами, 7% - платформою як сервісом, 1% - хмарним оркеструванням або IaaS конфігурацією.

Контейнери як сервіс або контейнери як послуга (Containers as a service, Caas) – хмарний сервіс, що надає оркестранти, що дають можливість керувати контейнерами та їх інфраструктурою.

Інфраструктура як сервіс або послуга (Infrastructure as a Service, IaaS) - сервіс, що надає інфраструктуру, якою людині не треба керувати. Наприклад: замість цілого дата центру комп'ютерного апаратного забезпечення можна використати віртуальне АЗ надане хмарною платформою.

Платформа як сервіс або послуга (Platform as a service, PaaS) – сервіс, що надає автоматизовані середовища. Він не тільки полегшує роботу з інфраструктурою, а ще й абстрагує від операційних систем та пов'язаних з цим проблем з розгортанням [13].

Сфери використання контейнерів:

- Мікросервіси: мікросервісна архітектура передбачає набір багатьох незалежних невеликих сервісів, що зручно розгортати з допомогою таких самих невеликих контейнерів.
- Модернізація та міграція застосунків: початок міграції багатьох застосунків на хмару починається саме з контейнеризації.
- DevOps: поєднання мікросервісів як архітектури та контейнерів як платформи є популярною основою для DevOps команд.
- Hybrid, multi-cloud: контейнери є зручними у ситуаціях, коли компаніями доводиться керувати комбінацією з приватних та публічних хмар, тому що вони можуть працювати у будь-якому середовищі [11].

1.2.3. Порівняльний аналіз

	Віртуальні машини	**Контейнери**
Операційна система	Потребують більше системних ресурсів (пам’ять, центральний процесор), бо використовує повноцінну ОС	Використовують лише частину ОС, бо для них можна налаштувати використання лише конкретних сервісів
Ізоляція	Повна ізоляція від ОС хоста та інших віртуальних машин	Легка ізоляція від хоста та інших контейнерів
Сумісність з гостьовою системою	Запускає будь-яку операційну систему всередині машини	Використовує операційну систему хоста
Розгортання	За допомогою Hypervisor software	За допомогою Docker або системи автоматичного розгортання, як-от: Kubernetes
Постійне сховище	Для однієї віртуальної машини використовується Віртуальний Жорсткий Диск (Virtual Hard Disk, VHD). Для декількох – спільний файловий ресурс (Server Message Block, SMB).	Для одного вузла – локальне сховище, SBM для декількох
Балансування навантаження	Забезпечується перенесенням віртуальних машин на інші сервери у відмово стійкому кластері	Оркестрант автоматично запускає або припиняє роботу контейнерів у вузлі кластера, щоб управляти змінами навантаження та доступу
Мережа	Використовують віртуальні мережеві адаптори	Використовують ізольоване представлення віртуального мережевого адаптера

Таблиця 2 - порівняння віртуальних машин та контейнеризації [14]

2. Розробка власного рішення

Метою практичної частини роботи стало розроблення власного багаторівневого застосунку – книжкового сервісу, кожний з рівнів якого розміщений в окремому Docker контейнері.

Було обрано три рівні: представлення, бізнес логіки та даних.

Рисунок 2.1 - Структура рівнів застосування та їх взаємодії

Були реалізовані наперед визначені мінімальні функціональні можливості:

- Можливість реєстрації та входу в систему
- Перегляд списку усіх книг системи
- Перегляд відгуків до кожної книги
- Додавання нових книг та їх видалення (для адміністраторів)
- Додавання нових відгуків до книг та їх видалення (для усіх авторизованих користувачів)

2.1. База даних

Першим кроком розробки власного рішення стало створення бази даних, що відповідатиме одному з рівнів застосунку. Для цього завдання була обрана реляційна база даних PostgreSQL.

Всередині неї зберігаються дані про п’ять сутностей – книга, користувач, роль користувача, зв'язок ролей та користувачів «user_roles» та «read». Остання описує зв'язок між користувачем та книгою – дата прочитання людиною того чи іншого твору, опціонально вказані оцінка та відгук.

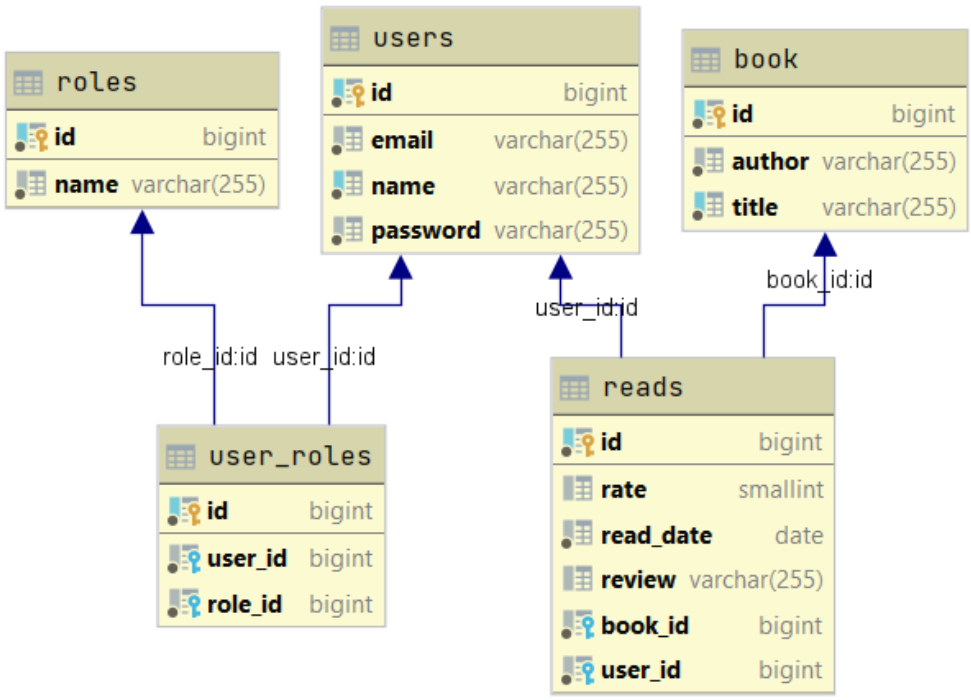

Рисунок 2.1.1 - схема таблиць реалізованої бази даних

2.2. Серверна частина

Серверна частина застосунку написана на мові програмування Java з використанням фреймворку Spring.

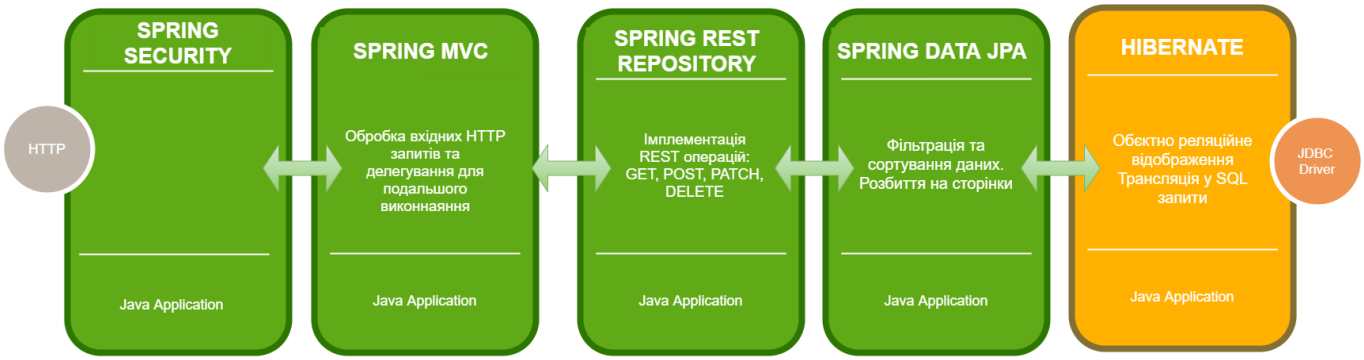

Рисунок 2.2.1 - схема організації серверної частини застосування

Джерело даних тобто базу даних було вказано в файлі application.properties.

```
spring.datasource.url=jdbc:postgresql://localhost:5432/postgres
spring.datasource.username=postgres
spring.datasource.password=example
```

Рисунок 2.2.2 - підключення бази даних до серверної частини застосування

Для отримання даних з бази даних були створені репозиторії, що є нащадками CRUD репозиторія – JpaRepository та PagingAndSortingRepository.

Отриманні дані приводяться до вказаних класів-моделей. Хоч у базі даних 5 таблиць у серверній частині вони представлені лише у 4 класах. Зв'язок користувачів та їх ролей представлений Hibernate анотацією @ManyToMany.

Запити приймають Rest контроллери.

Авторизація та автентифікація були реалізовані за допомогою Spring Security: при успішній автентифікації користувачу надається jwt токен з певним терміном дії, що буде збережений в сесії на сторони графічного інтерфейсу. При подальших запитах з необхідністю авторизації сервер перевірятиме валідність токену.

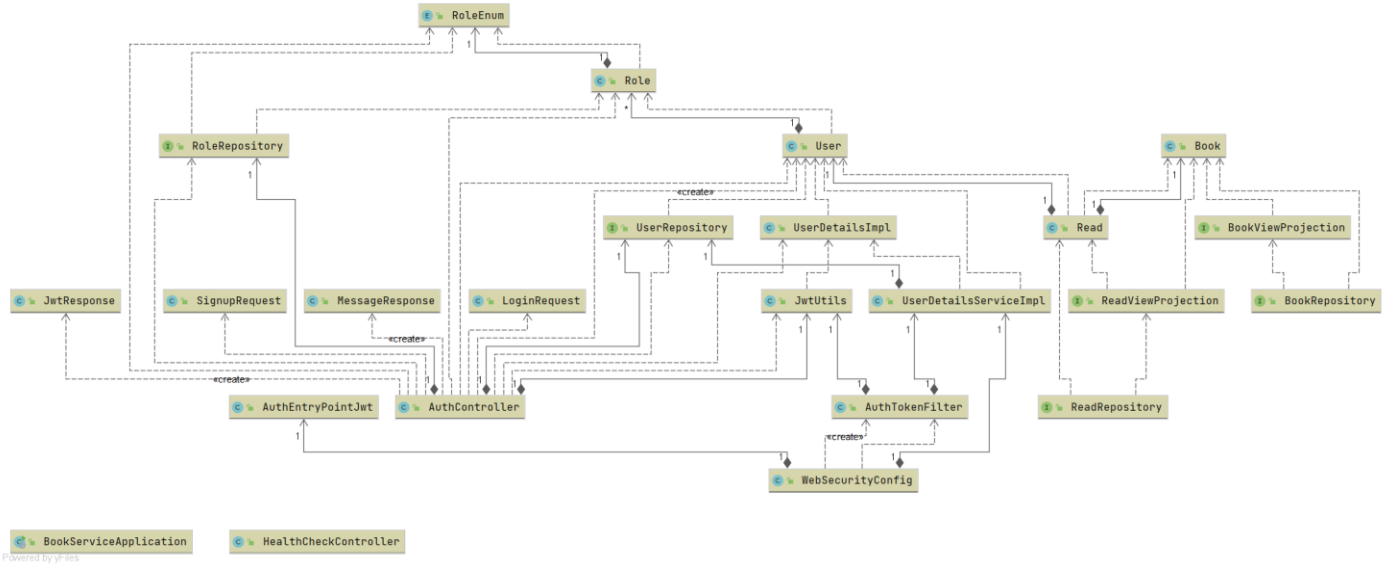

Рисунок 2.2.3 - діаграма класів серверного застосування

2.3. Інтерфейс користувача

Графічне представлення застосунку було реалізовано за допомогою front-end фреймворку Angular, що дозволяє створювати сайти з однієї сторінки. Це означає, що при переході на інший url сайту, він не оновлюється, а лише змінює поточний компонент.

Для даного застосунку були створені компоненти для «полиці» з усіма відгуками користувача, списку книг, книги як елементу списку, книги з детальною інформації та авторизації.

Отримання даних з серверу виконується за допомогою запитів класу HttpClient.

```
getBookById(id: number): Observable<Book> {
  return this.http.get<Book>( url: `${apiUrl}books/${id}`);
}
```

Рисунок 2.3.1 - приклад запиту з інтерфейсу користувача на сервер

Вигляд сайту залежить від того, чи зайшов користувач у свій обліковий запис та яка у нього роль. У не авторизованого користувача є можливість тільки переглядати список книг та відгуків до кожної.

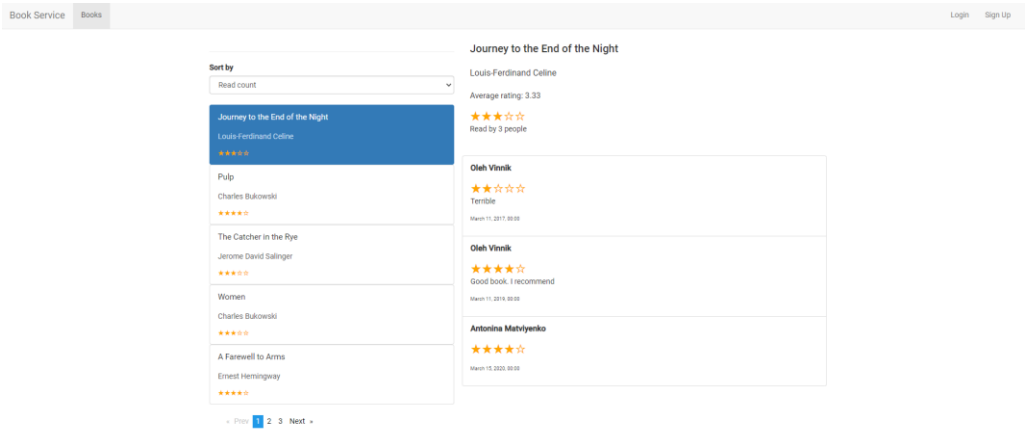

Рисунок 2.3.2 - головна сторінка неавторизованого користувача

У авторизованого користувача з’являється можливість перегляду всіх своїх відгуків, їх створення та видалення.

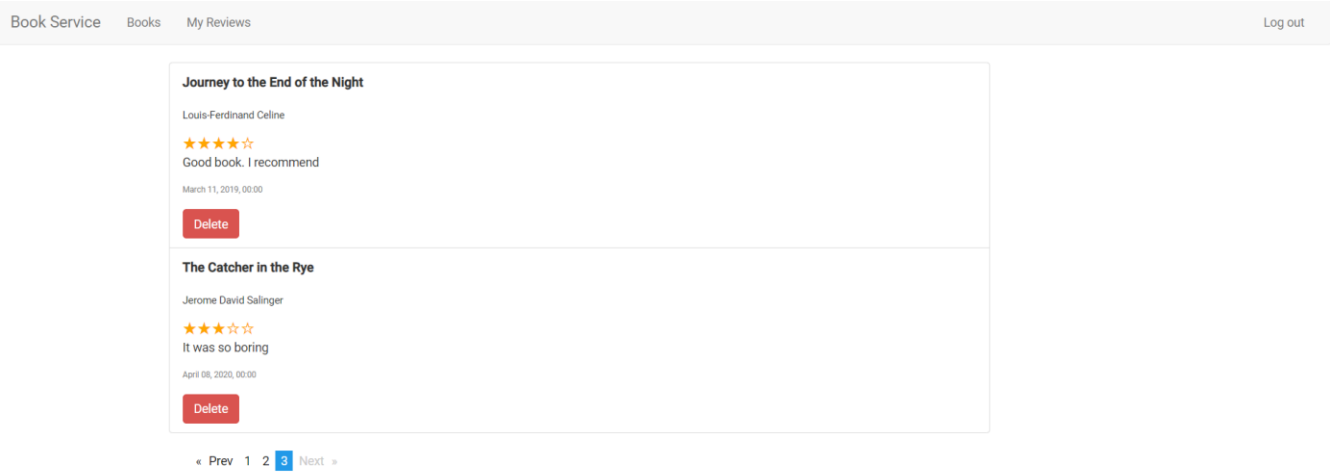

Рисунок 2.3.3 - перегляд відгуків, що надіслав користувач

У користувачів з роллю «адміністратор» також є можливість додавати книги.

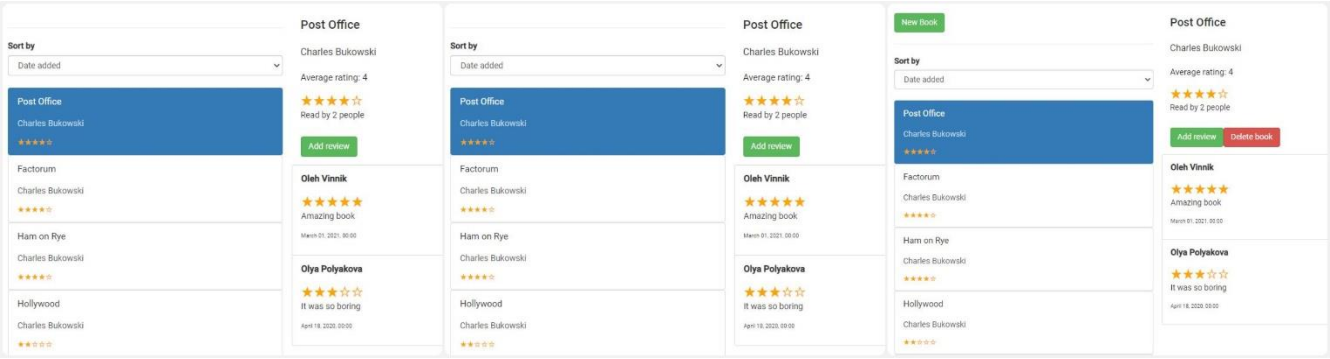

Рисунок 2.3.4 - порівняння головної сторінки залежно від ролі користувача

2.4. Контейнеризація

Останнім кроком при розробці застосунку стала його контейнеризація на основі платформи Docker.

Docker – один з найпопулярніших інструментів для управління контейнерами. Він базується на клієнт-серверній архітектурі і складається з таких елементів:

- Демон (daemon) – слухає Docker API і залежно від отриманого запиту керує образами, контейнерами та іншими Docker об’єктами.
- Клієнт – засіб спілкування користувачів з Docker, який і надсилає запити на Docker API. Прикладом слугує Docker compose.
- Реєстр – місце збереження образів. Прикладом публічного реєстру є Docker Hub.
- Образ – шаблон інструкцій для створення контейнерів. Він може бути створений на основі іншого образу. В такому випадку треба написати Dockerfile з кроками створення власного образу.
- Контейнер – екземпляр образу, що може бути запущений [15].

Для реалізації книжкового сервісу було створено 4 контейнери: база даних, сервер, інтерфейс користувача, консоль для адміністрування бази даних – pgAdmin від PostgreSQL.

Для перших трьох були створені нові образи, тож для них були написані відповідні Dockerfile.

Dockerfile бази даних вказує образ PostgreSQL останньої версії з Docker Hub, а також папку «data», в якій будуть зберігатися файли бази даних.

```
FROM  postgres
ENV PGDATA=/data
```

Рисунок 2.4.1 - dockerfile для бази даних

Після успішного створення образу бази даних, інформація про нього, параметри для доступу до бази даних, а також порт для запуску були внесені до docker-compose файлу:

```
db:
  image: postgres
  container_name: db
  build:
    context: docker/postgres-wrapper
    dockerfile: Dockerfile
  volumes:
    - bookservice_data:/data/postgres
  environment:
    POSTGRES_USER: postgres
    POSTGRES_PASSWORD: example
    PGDATA: /data/postgres
  ports:
    - 5432:5432
  networks:
    - spring-network
```

Рисунок 2.4.2 – частина docker-compose файлу про базу даних

Dockerfile серверної частини вказує образ реалізації Java, OpenJDK, 11-ої версії відповідно з версією мови, на якій був написаний проект. Після цього він копією скомпільовану версію застосунку з папки target та перейменовує її у app.jar. Як точку входу програми він визначає виконання команди «java -jar» на об’єкті, який був створений попереднім рядком.

```
FROM adoptopenjdk:11-jre-hotspot
COPY target/bookservice-1.0.1.jar app.jar
ENTRYPOINT ["java","-jar","/app.jar"]
```

Рисунок 2.4.3 - dockerfile для серверної частини

Тоді в docker-compose була додана інформація про контейнер з серверною частиною коду: були вказані залежність від контейнеру з базою даних, необхідні

для конфігураційного файлу Spring дані (адреса бази даних, ім’я користувача та пароль для доступу), а також порт.

```
book-server:
  depends_on:
    - db
  container_name: book-server
  build:
    context: ""
    dockerfile: Dockerfile
  image: bookservice:latest
  environment:
    - SPRING_DATASOURCE_URL=jdbc:postgresql://db:5432/postgres
    - SPRING_DATASOURCE_USERNAME=postgres
    - SPRING_DATASOURCE_PASSWORD=example
    - SPRING_JPA_HIBERNATE_DDL_AUTO=none
  ports:
    - 8041:8041
  networks:
    - spring-network
```

Рисунок 2.4.4 – частина docker-compose файлу для серверної частини

Dockerfile для UI частини проекту робить наступне:

1. Бере образ Node з реєстру
2. Створює нову папу «app» та робить її поточною робочою папкою
3. Копіює у неї файл package.json з проекту
4. Запускає команду «npm install» для встановлення усіх залежностей
5. Копіює папку та запускає «npm run build» для генерування збірки проекту
6. Бере образ nginx (веб-сервер)
7. Копіює результати збірки, щоб замінити початкові параметри nginx

```
FROM node:10-alpine as build-step
RUN mkdir -p /app
WORKDIR /app
COPY package.json /app
RUN npm install
COPY . /app
RUN npm run build --prod
FROM nginx:1.17.1-alpine
COPY --from=build-step /app/dist/book-service-ui /usr/share/nginx/html
```

Рисунок 2.4.5 – dockerfile для інтерфейсу користувача

Тоді за аналогією з попередніми контейнерами інформація у docker-compose включає у себе назву образа, контейнера, номер порту та залежність від інших контейнерів.

```
book-service-ui:
  image: book-service-ui:latest
  container_name: ui
  ports:
    - "4200:80"
  restart: always
  depends_on:
    - book-server
  networks:
    - angular-network
```

Рисунок 2.4.6 - частина docker-compose файлу про інтерфейс користувача

Останнім додатковим контейнером було визначено контейнер, що надає доступ до консолі бази даних pgAdmin. Для нього береться вже готовий образ з Docker hub тому в docker-compose лишається вказати дані для доступу, порт, папку для зберігання даних.

```
pgadmin:
  container_name: pgadmin_container
  image: dpage/pgadmin4
  environment:
    PGADMIN_DEFAULT_EMAIL:        @ukr.net
    PGADMIN_DEFAULT_PASSWORD: example
    PGADMIN_CONFIG_SERVER_MODE: 'False'
  volumes:
    - pgadmin:/home/user/pgadmin
  ports:
    - "${PGADMIN_PORT:-5050}:80"
  networks:
    - spring-network
  restart: unless-stopped
```

Рисунок 2.4.7 - частина docker-compose файлу для консолі адміністрування бази даних

Тепер проект з 4 контейнерів може бути запущений на Docker.

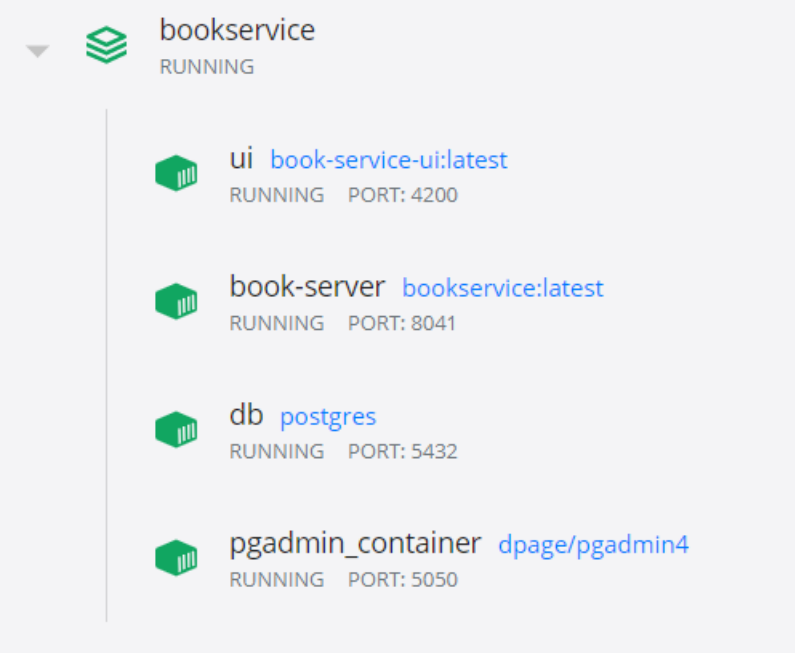

Рисунок 2.4.8 – запущений проект в Docker

ВИСНОВКИ

Під час виконання курсової роботи було розглянуто різницю між шарами та рівнями застосувань, проаналізовано різні типи багаторівневої архітектури – їх переваги та недоліки, а також оглянуто основні підходи віртуалізації – віртуальні машини та контейнеризацію.

Отримана інформація допомогла при виконанні практичної частини розробити трирівневе застосування, кожен з рівнів якого розміщений в окремому контейнері Docker. Отриманий продукт може бути корисним як приклад контейнеризації.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. What is Three-Tier Architecture [Електронний ресурс] – Режим доступу до ресурсу: https://www.ibm.com/cloud/learn/three-tier-architecture
2. Three-tier Application Model [Електронний ресурс] – Режим доступу до ресурсу: https://docs.microsoft.com/en-us/previous-versions/office/developer/server-technologies/aa480455(v=msdn.10).
3. SQL Stored Procedures for SQL Server [Електронний ресурс] – Режим доступу до ресурсу: https://www.w3schools.com/sql/sql_stored_procedures.asp.
4. TM V. Client Server Architecture [Електронний ресурс] / Vijay TM – Режим доступу до ресурсу: https://www.scribd.com/presentation/135327979/6670080-Clint-Server.
5. Cao J. Research and Application of the Four-tier Architecture / J. Cao, J. Wei, Y. Qin. // Atlantis Press. – 2013. – С. 760–761.
6. What is Virtualization? [Електронний ресурс] – Режим доступу до ресурсу: https://www.ibm.com/cloud/learn/virtualization-a-complete-guide.
7. What is virtualization? [Електронний ресурс] – Режим доступу до ресурсу: https://www.redhat.com/en/topics/virtualization/what-is-virtualization.
8. Virtual Machines [Електронний ресурс] – Режим доступу до ресурсу: https://www.ibm.com/cloud/learn/virtual-machines.
9. Introduction to Virtualization [Електронний ресурс] – Режим доступу до ресурсу: https://www.baeldung.com/cs/virtualization-intro.
10. Containerization [Електронний ресурс] – Режим доступу до ресурсу: https://www.ibm.com/cloud/learn/containerization.
11. Containers [Електронний ресурс] – Режим доступу до ресурсу: https://www.ibm.com/cloud/learn/containers.

12. What Is Container Orchestration? [Електронний ресурс] – Режим доступу до ресурсу: https://newrelic.com/blog/best-practices/container-orchestration-explained.
13. Containers As a Service: a complete guide [Електронний ресурс] – Режим доступу до ресурсу: https://www.scalyr.com/blog/containers-as-a-service-complete-guide/.
14. Virtualization vs Containerization [Електронний ресурс] – Режим доступу до ресурсу: https://www.baeldung.com/cs/virtualization-vs-containerization.
15. Docker overview [Електронний ресурс] – Режим доступу до ресурсу: https://docs.docker.com/get-started/overview/.