Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Навчально-науковий центр заочної форми навчання
(повна назва)

Кафедра *Інформаційно-мережної інженерії*
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти перший (бакалаврський)

Застосування технологій контейнеризації для створення середовищ розгортання та запуску додатків
(тема)

Виконала:
здобувачка *4* року навчання,
групи *ТРІМІз-21-1*

Вікторія РОМАНЦОВА
(власне ім'я, прізвище)

Спеціальність *172 Телекомунікації та радіотехніка*
(код і повна назва спеціальності)

Тип програми *освітньо-професійна*

Освітня програма
«Інформаційно-мережна інженерія»
(повна назва освітньої програми)

Керівник доцент *Юрій КОЛТУН*
(посада, власне ім'я, прізвище)

Допускається до захисту

Зав. кафедри _____ *Валерій БЕЗРУК*
(підпис) (власне ім'я, прізвище)

2025 р.

Не містить відомостей заборонених до відкритого публікування.

Здобувачка / *Вікторія РОМАНЦОВА* /

Керівник / *Юрій КОЛТУН* /

Харківський національний університет радіоелектроніки

Навчально-науковий центр заочної форми навчання

Кафедра *Інформаційно-мережної інженерії*
(повна назва)

Рівень вищої освіти *перший (бакалаврський)*

Спеціальність *172 Телекомунікації та радіотехніка*
(код і повна назва)

Тип програми *освітньо-професійна*
(освітньо-професійна або освітньо-наукова)

Освітня програма *«Інформаційно-мережна інженерія»*
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«*02*» *травня* 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачці *Романцовій Вікторії Віталіївні*
(прізвище, ім'я, по батькові)

1. Тема роботи *Застосування технологій контейнеризації для створення середовищ розгортання та запуску додатків*
затверджена наказом університету від «*02*» *травня* 2025 р. № 63 *Стз*
2. Термін подання студентом роботи до екзаменаційної комісії *20 червня 2025 р.*
3. Вихідні дані до роботи *Технологія контейнеризації - Docker. Віртуальне середовище – контейнери. Операційне середовище – ОС Windows та Linux. Тип додатку - To-Do. Підходи щодо розгортання та запуску додатку – створення образу та запуск додатку у контейнері.*
Надати рекомендації щодо інсталяції Docker в ОС Windows та Linux. Запропонувати практичні принципи створення і запуску додатків з використанням технології Docker.

4. Перелік питань, що потрібно опрацювати в роботі
Вступ
1. Загальні особливості технологій віртуалізаціїї на основі контейнерів в якості віртуального середовища
2. Базові поняття, архітектура, компоненти та особливості інсталяції платформи Docker
3. Практичні принципи створення і запуску додатків Docker
Висновки

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (слайдів) *Слайди у форматі Power Point (назва, мета і актуальність роботи, відмінності контейнерів від віртуальних машин, базові поняття концепції Docker, співвідношення понять «контейнер» і «образ» Docker, архітектура та ключові команди Docker, інсталяція Docker в середовищі ОС Windows, інсталяція Docker в середовищі ОС Ubuntu на прикладі ОС Ubuntu Trusty 14.04, схема створення Docker-додатку, структура Dockerfile та його вміст, образ Docker із створеного Dockerfile, запуск образу Docker як контейнера, шарова структура файлової системи Docker, варіант утворення шарів у файловій системі платформи Docker висновки)*

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	*Ознайомлення із завданням. Уточнення ТЗ.*	02.05 – 03.05.25	*виконано*
2	*Підбір літератури за темою роботи.*	04.05 – 10.05.25	*виконано*
3	*Виконання розділу 1*	11.05 – 20.05.25	*виконано*
4	*Виконання розділу 2*	21.05 – 29.05.25	*виконано*
5	*Виконання розділу 3*	30.05 – 11.06.25	*виконано*
7	*Оформлення пояснювальної записки*	12.06 – 20.06.25	*виконано*
8	*Оформлення презентаційного матеріалу,*		
	підготовка до захисту у ЕК, захист.	21.06 – 26.06.25	*виконано*

Дата видачі завдання *02 травня 2025 р.*

Здобувачка _____
(підпис)

Керівник роботи _____ *(доц. Юрій КОЛТУН)*
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка: 64 с., 22 рис., 1 табл., 23 джерел, 1 додаток.

КОНТЕЙНЕР, ВІРТУАЛЬНА МАШИНА, VM, КОНТЕЙНЕРИЗАЦІЯ, ІНСТАЛЯЦІЯ DOCKER, ОБРАЗ КОНТЕЙНЕРУ, ОБРАЗ DOCKER, DOCKERFILE, КОМАНДИ DOCKER ЗАПУСК КОНТЕЙНЕРУ, ШАРИ DOCKER

Об'єкти дослідження – платформа Docker.

Мета роботи – проведення загального аналізу можливостей технології контейнеризації та надання практичних принципів застосування платформи Docker для створення середовища розгортання додатків з використанням контейнерів.

Проведений загальний аналіз особливостей появи та напрямів розвитку технологій контейнеризації, аналіз архітектури та компонентів платформи Docker, її компонентів та елементів. Досліджені практичні принципів створення і запуску додатків на платформі Docker у вигляді контейнерів.

THE ABSTRACT

Explanatory note 64 pages, 22 fig., 1 tab., 23 sources, 1 app.

CONTAINER, VIRTUAL MACHINE, VM, CONTAINERIZATION, DOCKER INSTALLATION, CONTAINER IMAGE, DOCKER IMAGE, DOCKERFILE, DOCKER COMMANDS, CONTAINER STARTS, DOCKER TIERS

Objects of research – платформа Docker.

The purpose of work – providing a general analysis the capabilities of containerization technology and practical principles for using the Docker platform to create an application deployment environment using containers.

A general analysis of the characteristics and development trends of containerization technologies was conducted, along with an analysis the architecture and components of the Docker platform, its components, and elements. Practical principles for creating and launching applications on the Docker platform in the form of containers were investigated..

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

CI/CD (Continuous Integration / Continuous Delivery (Deployment)) – безперервна інтеграція, доставка та розгортання;

CPU (Central Processing Unit) – центральний процесор;

CRI (Container Runtime Interface) – інтерфейс середовища виконання контейнерів;

HTTP (Hyper Text Transfer Protocol) – протокол передачі гіпертексту;

LXC (LinuX Containers) – контейнери Linux;

OCI (Open Container Initiative) – відкрита контейнерна ініціатива;

RBAC (Role-Based Access Control) – управління доступом на основі ролей;

SCM (Service Control Manager) – диспетчер управління службами;

VDS (Virtual Dedicated Server) – віртуальний виділений сервер;

VM (Virtual Machine) – віртуальна машина;

VPS (Virtual Private Server) – віртуальний сервер (віртуальний приватний сервер).

ОЗП– оперативний запам’ятовуючий пристрій;

ОС – операційна система;

ПЗ – програмне забезпечення.

ВСТУП

Контейнеризація є однією з найважливіших технологій сучасної розробки програмного забезпечення та розвитку IT-інфраструктури. Її впровадження радикально змінило підходи до способів розробки, розповсюдження та розгортання, функціонування, управління та масштабування додатків і web-сервісів. Розробники можуть створювати свої програмні додатки у власній локальній системі, при цьому вони точно знають, що всі програми будуть однаково працездатними та функціональними в будь-якому операційному середовищі – від програмного комплексу невеликого IT-відділу або компанії на ноутбуці користувача, до хмарного кластера. Мережні інженери можуть зосередитися на підтримці роботи в мережному оточенні, на наданні ресурсів та на забезпеченні безперебійного функціонування як мережного обладнання, так і системного та прикладного програмного забезпечення (ПЗ). Тобто, іншими словами, вони витрачатимуть менше часу на конфігурацію мережного оточення і на «боротьбу» із системними залежностями [1].

Таким чином, технології контейнеризації дають змогу економити на апаратних обчислювальних засобах, а також вони є менш ресурсовитратними. Якщо розглядати контейнеризацію в контексті інформаційної безпеки, то можна зазначити, що вона підвищує безпеку додатків, оскільки кожен контейнер працює в ізольованому середовищі, а це знижує ризики атаки на інші додатки або хостову систему [2].

Однією з найпопулярніших платформ для контейнеризації, що широко використовується як в IT-індустрії, так і в освітніх цілях, є Docker. Ця платформа пропонує простий і потужний підхід до створення, розгортання та управління контейнерами.

Docker автоматизує створення, доставку та управління контейнерами, роблячи процес розгортання додатків швидким і надійним. Контейнери Docker запускаються за секунди, займають мало місця і легко клонуються, що особливо важливо для реалізації практик управління проектами DevOps і розробки програмного забезпечення в умовах безперервної інтеграції, доставки та розгортання CI/CD (Continuous Integration/Continuous Deployment, CI/CD), які дають змогу автоматизувати процес збірки, тестування і доставки ПЗ у виробниче середовище [2, 3].

Docker ідеально підходить для мікросервісної архітектури: кожен сервіс упаковується в окремий контейнер, що спрощує оновлення, масштабування і підтримку. Він тісно інтегрується з інструментами управління (оркестрації) контейнерами (Kubernetes, Docker Swarm), що дає змогу будувати відмовостійкі та масштабовані системи, а також реалізує контроль доступу до апаратних ресурсів (CPU, пам'ять, диск) і підтримує регулярне оновлення образів і сканування на вразливості [4, 5].

Таким чином, платформа Docker виступає практично в якості стандарту контейнеризації, пропонуючи зручні інструменти для автоматизації, управління, інтеграції та розгортання додатків у будь-яких середовищах. Тому метою цієї кваліфікаційної роботи якраз і є проведення загального аналізу можливостей технології контейнеризації та надання практичних принципів застосування платформи Docker для створення середовища розгортання додатків з використанням контейнерів. Впровадження технологій контейнеризації, і зокрема на основі Docker, дає змогу IT-компаніям прискорювати розробку програмних додатків, знижувати витрати, підвищувати безпеку та гнучко реагувати на зміни ринку, що свідчить про актуальність роботи.

1 ЗАГАЛЬНІ ОСОБЛИВОСТІ ТЕХНОЛОГІЙ ВІРТУАЛІЗАЦІЇ НА ОСНОВІ КОНТЕЙНЕРІВ В ЯКОСТІ ВІРТУАЛЬНОГО СЕРЕДОВИЩА

1.1 Етапи розвитку та загальні особливості технології контейнеризації

Ідея контейнеризації або «ізоляції просторів користувача» бере свій початок у 1979 році, коли в ядрі UNIX з'явився системний виклик `chroot`. Він вважається однією з перших технологій контейнеризації та надає змогу ізолювати процес і його дочірні елементи від інших частин операційної системи (ОС). Єдина проблема з цією ізоляцією полягає в тому, що кореневий процес може легко вийти з `chroot`. У ньому ніколи не закладалися механізми безпеки [6, 7].

Логічним продовженням `chroot` стало створення у 2000 році механізму віртуалізації FreeBSD Jail (`jail` - «в'язниця», «клітка»), що був представлений в ОС FreeBSD і який призначений для забезпечення більшої безпеки простої ізоляції файлів `chroot`. FreeBSD Jail на відміну від `chroot` ізолює процеси та їхні дії від файлової системи [6, 7].

Коли в ядро Linux було додано можливості віртуалізації на рівні операційної системи, у 2001 році був представлений механізм віртуалізації Linux VServer, який використовував як `chroot`-подібний механізм у поєднанні з так званими «security contexts» (контекстами безпеки), так і віртуалізацію на рівні ОС. Він є більш просунутим, ніж простий `chroot`, і дає змогу запускати кілька дистрибутивів ОС Linux на одному віртуальному сервері VPS (Virtual Private Server). На одному фізичному носії може працювати відразу декілька ізольованих один від одного VPS [7].

У 2004 році компанія Sun випустила продукт Solaris Containers. Він фактично являв собою реалізацію Linux-VServer для процесорів X86 і SPARC. Контейнер Solaris – це комбінація елементів управління ресурсами системи і розділення ресурсів, що забезпечуються так званими «зонами (zone)»,які працюють як повністю ізольовані віртуальні сервери всередині однієї ОС. Запускаючи додатки в одній системі та розміщуючи кожен з них у своєму віртуальному контейнері можна реалізувати на одній машині такий самий рівень захисту, який би був у тому випадку, коли всі додатки працювали б на різних машинах [7].

У 2005 році компанія Parallels Inc. представила рішення віртуалізації на рівні ОС OpenVZ, яке надавало змогу запускати на одному фізичному сервері безліч ізольованих копій ОС, що представляють VPS або контейнери. Це рішення було прийняте багатьма хостинговими компаніями для ізоляції та продажу VPS. Віртуалізація на рівні ОС має деякі обмеження, оскільки контейнери і хост використовують одну і ту ж саму архітектуру та версію ядра. Недолік виникає в ситуаціях, коли гостям потрібні версії ядра, що є відмінними від версії на хості. Linux-VServer і OpenVZ вимагають патча ядра, щоб додати деякі механізми управління, які використовуються для створення ізольованого контейнера. Зазначу, що патчі OpenVZ не були інтегровані в ядро [7].

В ОС Linux технології «ізоляції» та «віртуалізації» ресурсів вийшли на новий етап у 2002 році, коли в ядро було додано перший простір імен для ізоляції файлової системи – `mount`. У 2006-2007 роках компанією Google було розроблено механізм Process Containers (пізніше перейменований на CGroups), який дав змогу обмежити та ізолювати використання ресурсів (CPU, ОЗП та ін.) для набору процесів. У 2008 році функціонал CGroups було додано до ядра ОС Linux. Достатня функціональність для повної ізоляції та безпечної роботи контейнерів була завершена у 2013 році з додаванням до ядра простору імен користувачів – `user` [6, 7].

У 2008 році було представлено систему LXC (LinuX Containers), яка була схожа на OpenVZ, Solaris Containers і Linux-VServer. Ця система дозволила запускати декілька ізольованих Linux-систем (контейнерів) на одному сервері [6, 7].

У 2013 році було презентовано першу версію платформи Docker, завдяки якій контейнерні технології стали дуже популярними завдяки простоті використання та широкому функціоналу. Спочатку Docker використовував LXC для запуску контейнерів, проте пізніше перейшов на власну бібліотеку `libcontainer`, також зав'язану на функціонал ядра ОС Linux, а вже в жовтні 2014 року корпорація Microsoft оголосила про повну підтримку Docker у майбутніх версіях Windows Server. У серпні 2015 року Docker і Microsoft спільно випустили «попередній технічний огляд» реалізації Docker Engine для Windows Server [1, 6].

Нарешті, у червні 2015 року на конференції DockerCon у Сан-Франциско Соломон Хайкс (Docker) і Алекс Полві (Alex Polvi) (CoreOS) оголосили про створення проекту Open Container Initiative (OCI) для розроблення спільного

стандарту формату контейнерів і механізмів запуску. Цей проект пізніше було перейменовано в Open Container Project, і наразі в його рамках продовжуються роботи з регламентації та стандартизації розвитку контейнерних технологій [1, 6].

Таким чином, з вищевикладених етапів розвитку контейнеризації як механізму віртуалізація на рівні ОС, можна бачити, що це технологія, яка дає змогу запускати програмне забезпечення в ізольованих на рівні ОС просторах, а контейнери є найпоширенішою формою віртуалізації на рівні операційної системи. За допомогою контейнерів можна запустити кілька додатків на одному хості (фізичному сервері), ізолюючи їх один від одного. Інакше кажучи, процес, запущений у контейнері, виконується всередині ОС на хості, але при цьому він ізольований від інших процесів. Для самого процесу це виглядає так, ніби він єдиний працює в системі [6].

Звідси контейнер можна визначити як засіб інкапсуляції додатку з необхідним інструментарієм для його роботи. В ізольованому просторі збираються бібліотеки, бінарні файли та дані, що потрібні тільки для роботи конкретного додатку. Це, у свою чергу, надає змогу легко переносити повністю робочі продукти незалежно від пристрою, на якому вони були створені, водночас зберігаючи дуже невеликий розмір і високу швидкість запуску [1, 8].

1.2 Відмінності контейнерів від віртуальних машин та їх переваги у використанні

Контейнери у першому наближенні можуть здатися спрощеним варіантом віртуалізації на основі віртуальних машин (Virtual Machine, VM), тобто, як і VM, контейнер може містити ізольовану ОС, яку можна використовувати для запуску додатків. Однак VM є втіленням механізмів традиційної віртуалізації, що передбачає емуляцію всієї операційної системи, а контейнери спільно використовують ядро ОС хосту, що робить їх легкими, швидкими та економічними у використанні ресурсів [1, 4].

Основна ідея традиційної віртуалізації полягає в тому, щоб розгорнути кілька VM на одному фізичному комп'ютері. При цьому слід врахувати, що кожен комп'ютер має свою хостову ОС, таку як: Windows, MacOS, Linux (Ubuntu, Centos), тощо. Деякі додатки можуть працювати тільки на MacOS, а інші – виключно на ОС Windows, тому технологія, що об'єднує різні ОС на одному

фізичному сервері, набула популярності серед користувачів і компаній. VM у найпростішому її вигляді – це автономна віртуальна система, що містить у собі все, – починаючи від власної ОС (яка називається гостьовою) до оточення і самого застосунку. На хост (або фізичний сервер) можна встановити кілька VM за допомогою шару, який називається гіпервізором, що діє поверх ОС хосту [8, 9].

Гіпервізор або монітор віртуальних машин (в комп'ютерах) – це програма або апаратна схема, яка забезпечує або дозволяє одночасне, паралельне виконання кількох ОС на одному і тому ж хост-комп'ютері. Основне призначення гіпервізора полягає у забезпеченні ізольованих середовищ виконання для кожної VM та управління доступом віртуальної машини та гостьової ОС до фізичних апаратних ресурсів комп'ютера [10].

Цей гіпервізор, який також називають гіпервізором 2-го типу, функціонує як проксі-сервер для доступу до обладнання, завдяки чому гостьова ОС вважає, що вона працює на звичайному фізичному комп'ютері. Однак у реальності гостьові ОС віртуальних машин розташовуються рівнем вище, як це показано на рис. 1.1 [10].

Рисунок 1.1 – Спрощене подання VM з гіпервізором 2-го типу

Зазначимо, що існує також і гіпервізор 1-го типу, який працює безпосередньо на обладнанні (без хостової ОС) і вважається апаратним гіпервізором. Тут гостьові ОС віртуальних машин виконуються рівнем вище але

вони вважають, що працюють безпосередньо на апаратній платформі та не бачать гіпервізора (рис. 1.2) [10].

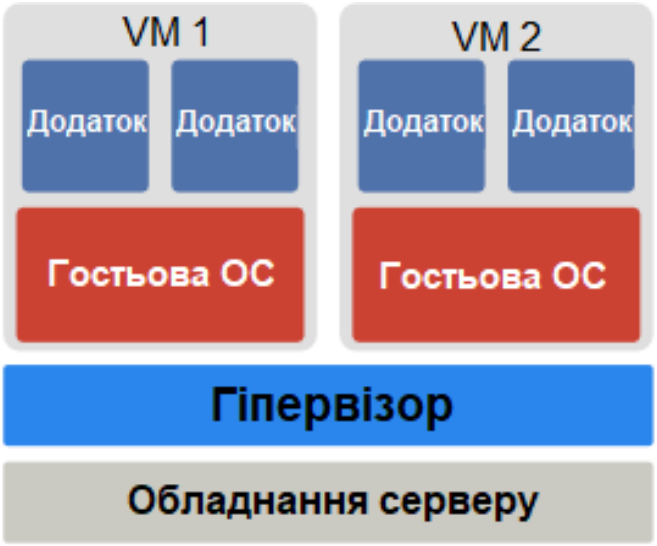

Рисунок 1.2 – Спрощене подання VM з гіпервізором 1-го типу

Великі компанії почали активно користуватися VM, розгортати свої хмарні інфраструктури (велика кількість VM, об'єднаних в одну мережу) і надавати сервіси хмарних обчислень звичайним користувачам. Використання VM дало змогу компаніям значно знизити витрати на апаратні ресурси та підвищити ефективність їх використання, оскільки до появи технологій віртуалізації на основі віртуальних машин ці компанії змушені були розгортати додатки на виділених серверах. Крім скорочення витрат використання VM надає також інші переваги, зокрема такі як [9]:

- ефективно використовувати ресурси сервера, забезпечуючи при цьому таке розділення додатків, що кожен із них може працювати на своїй власній ОС в окремій VM. Це зумовлено тим, що віртуальна машина для додатків має вигляд і діє як окрема фізична машина. Така ізоляція додатків один від одного за допомогою VM також забезпечує і додаткову безпеку;

- гнучко перерозподіляти ресурси між VM так, як це необхідно. Процесори, пам'ять та інші апаратні ресурси можна розподіляти відповідно із початковими умовами та фактичними потребами. Крім того, є можливість налаштувати автоматичний розподіл ресурсів, щоб збільшити продуктивність. Така ідея отримала надалі назву еластичності та знайшла широке застосування в хмарних технологіях;

- ефективне резервне копіювання та відновлення. VM можуть зберігатися у вигляді поодиноких файлів, резервні копії яких легко зберігати та, за потреби, переміщати. Також, у разі потреби з цих резервних копій можна легко відновити останню збережену працездатну VM у разі її збою;

- під управлінням одного і того ж гіпервізора може діяти кілька різних операційних систем, тобто є можливість підтримувати додатки, для виконання яких потрібні різні ОС;

- віртуальну машину можна легко перенести на інший, більш продуктивний хост. Більшість гіпервізорів підтримує таку можливість. Так найбільший розробник програмного забезпечення для віртуалізації американська компанія VMware пропонує, наприклад, функцію VMotion, яка дає змогу виконувати міграцію запущеної віртуальної машини з одного хоста на інший.

Однак у рішень на основі VM є значний недолік, який можна помітити на рис. 1.1. Там можна бачити хост зі своєю ОС, гіпервізор і додаткові гостьові ОС до кожної VM. Як відомо ОС споживають значні обсяги ресурсів, бо для їх підтримки необхідно мати великий обсяг пам'яті, як оперативної, так і фізичної, а також значну обчислювальна потужність. Крім того, простежується неефективність у порядку функціонування такої системи. Тут, як видно з рис. 1.1, додаток взаємодіє з гостьовою ОС, яка взаємодіє з гіпервізором, який, у свою чергу, взаємодіє з хостовою ОС, яка здійснює управління обладнанням і виконує запити, тобто через накладні витрати на роботу проміжних шарів загальна продуктивність буде зменшуватися. Навіть у разі використання гіпервізора 1-го типу (рис. 1.2), який безпосередньо здійснює управління обладнанням, зменшуються накладні витрати на взаємодію гіпервізора з хостовою ОС, але всі інші накладні витрати залишаються і продовжують у цілому негативно впливати на продуктивність системи [8, 9].

Також, у разі створення резервної копії VM, формується файл дуже великого розміру, тому що він буде містити ОС віртуальної машини, встановлений додаток з необхідними інструментами для його роботи та різні локальні дані. Зокрема деякі резервні копії VM можуть досягати обсягу понад 20 Гбайт. У результаті через великий розмір переміщення VM та організація спільного доступу до них із зовнішньої мережі може займати багато часу [9].

Контейнери також пропонують віртуальне середовище, в яке запаковуються додатки та інструментарій, що необхідний для їх функціонування: файлова

система, дерево каталогів, бібліотеки, простір процесу, тощо. На відміну від VM, контейнери не потребують власної ОС. Інакше кажучи, застосування контейнерів дало змогу позбутися проміжного шару у вигляді ОС, і у такий спосіб зменшити розмір і підвищити можливості щодо переносу додатків (рис. 1.3) [8, 9].

Рисунок 1.3 – Спрощене подання віртуального середовища на основі контейнерів

Як видно з рисунка 1.3, контейнери забезпечують повну ізоляцію додатків і процесів, коли один додаток нічого не знає про існування інших додатків. Однак усі процеси використовують одне і те ж саме ядро ОС. Для того, щоб незалежні процеси могли виконуватися під управлінням однієї хостової ОС, контейнери використовують засоби ізоляції ресурсів у її ядрі, наприклад, в ОС Linux - це групи управління та простори імен [9].

В результаті відпадає необхідність мати окремий екземпляр ОС, що працює, для кожної програми, як це було потрібно у випадку з VM. З одного боку можна зробити висновок, що VM забезпечують кращу ізоляцію, ніж контейнери. Однак, з іншого боку, контейнери є більш легкими і простішими для запуску та передачі між розробниками. Завдяки цьому на фізичному обладнанні сервера можна запустити більше контейнерів, ніж VM. Інакше кажучи, для розробників це дає можливість здійснити імітацію роботи промислової розподіленої системи на одній фізичній машині, що говорить про більш ефективне використання апаратних ресурсів [1, 9].

Крім того, контейнери надають переваги кінцевим користувачам і розробникам без необхідності розгортання додатка у хмарі. Відмінна переносимість контейнерів між різними апаратними та операційними середовищами забезпечує потенційну можливість усунення цілого класу програмних помилок, які можуть виникнути у разі навіть незначних змін робочого середовища. Користувачі можуть завантажувати і запускати складні додатки без багатогодинної роботи над конфігуруванням і можливими проблемами під час встановлення, та при цьому не турбуватися про будь-які зміни в їх локальних системах. У свою чергу, розробники подібних додатків можуть уникнути проблем, що пов'язані з відмінностями у конфігураціях середовищ користувачів та з доступністю технологічних інструментів, що необхідні для функціонування додатків [1].

З вищевикладеного можна бачити, що між застосуванням VM і контейнерів існують принципові відмінності, які стосуються мети їх використання. Зокрема метою застосування VM є повна емуляція стороннього програмного (операційного) середовища, тоді як метою застосування контейнера є необхідність зробити додатки незалежними від програмних та апаратних середовищ (властивість переносимості та самодостатності) [1].

2 БАЗОВІ ПОНЯТТЯ, АРХІТЕКТУРА, КОМПОНЕНТИ ТА ОСОБЛИВОСТІ ІНСТАЛЯЦІЇ ПЛАТФОРМИ DOCKER

2.1 Загальні особливості та базові поняття платформи Docker

Як було зазначено вище, на сьогоднішній день одним з найпопулярніших інструментів контейнеризації для розробників є платформа з відкритим вихідним кодом Docker, яка автоматизує створення, запуск і управління програмними додатками в легких контейнерах, що можна переносити між різними системами та інфраструктурами. Завдяки контейнеризації з використанням платформи Docker розробники більше не замислюються над тим, у якому середовищі функціонуватиме додаток і чи будуть у цьому середовищі необхідні для тестування опції та залежності. Достатньо упакувати застосунок з усіма інструментами (залежностями) і процесами в контейнер, щоб запускати в будь-яких ОС: Linux, Windows і MacOS. Платформа Docker дала змогу відокремити додатки від інфраструктури – оскільки контейнери не залежать від базової інфраструктури, то їх можна легко переміщати між хмарною та локальною інфраструктурами [4, 9].

Також зазначалося, що спочатку в основу платформи Docker було закладено існуючу до нього технологію Linux-контейнерів (LXC), у якій використовувалися образи, що можна було переносити, і зручний інтерфейс користувача - для створення повністю готового до застосування рішення, яке забезпечувало б створення та розповсюдження контейнерів. Зокрема Docker користується такими вбудованими технологіями ядра Linux [1, 8]:

- CGroups (Control Groups, контрольні групи) – дозволяє виділяти на кожен контейнер ресурси комп'ютера, такі як оперативна пам'ять, ядра, пам'ять, мережні ресурси тощо;

- Namespaces (простори імен) – дають змогу ізолювати такі системні ресурси, як процеси, мережі, ідентифікатори користувачів і файлову систему, надаючи змогу контейнерам працювати незалежно один від одного;

- Union File System (UnionFS) – Docker використовує багатошарову файлову систему (OverlayFS, AUFS або BTRFS (B-Tree Filesystem)), яка дозволяє створювати легкі образи контейнерів (Container images), що можна змінювати. Це дає можливість ефективно здійснювати управління шарами файлової системи, повторно використовуючи ті шари, що є незмінними, між різними контейнерами та образами.

Перш ніж запускати команди Docker, необхідно визначити базові поняття самої концепції Docker, які відображають її ключові особливості. До них належать, насамперед, поняття образів, контейнерів, шарів і процесів (рис. 2.1) [11].

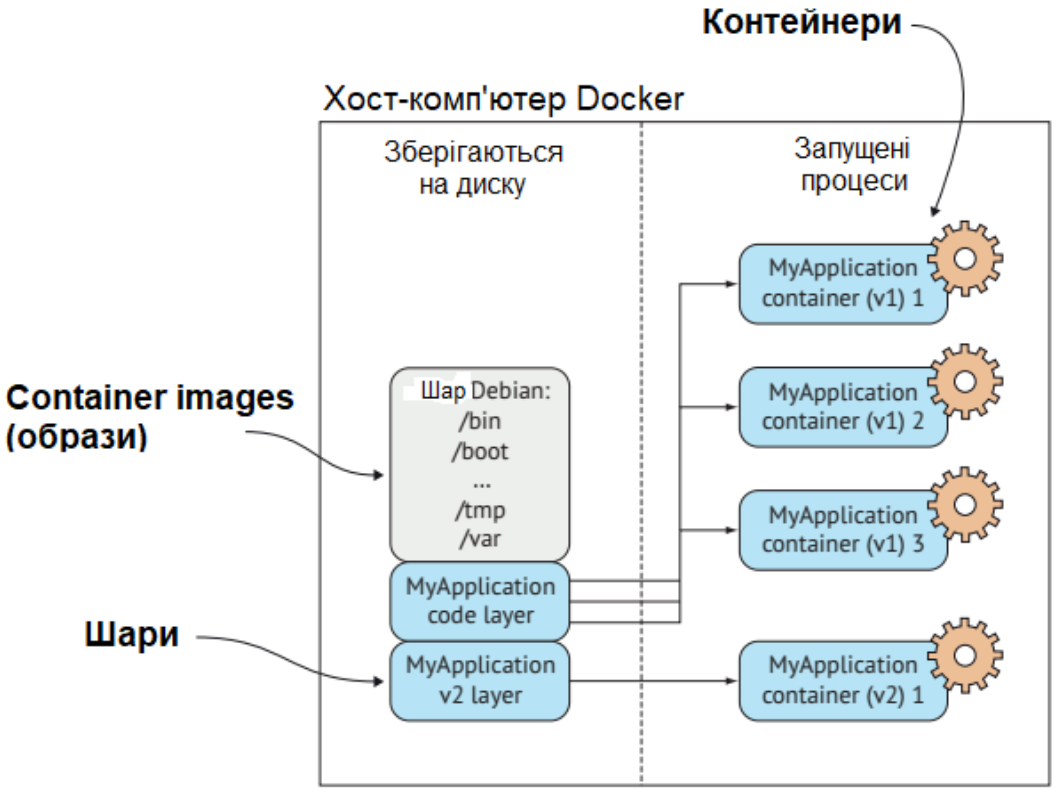

Рисунок 2.1 – Базові поняття концепції Docker

За аналогією із визначенням VM, що було надане у другому розділі, контейнер Docker також являє собою автономну віртуальну систему, що містить процес, який виконується, а також усі файли, залежності, адресний простір процесу та мережні порти, що необхідні додатку. Оскільки кожен контейнер має свій простір портів, слід організувати їх відображення у фактичні порти на рівні Docker. Контейнери запускають системи, що визначені образами. По суті, контейнер – це запущений екземпляр образу. Може бути кілька контейнерів, що запущені із одного образу [11].

Образ являє собою файл, у який упакований додаток та його середовище. Він містить файлову систему, яка буде доступна додатку, та інші метадані (наприклад, команди, які мають бути виконані під час запуску контейнера). Образи контейнерів складаються з шарів (шар – це набір змін у файлах плюс деякі метадані Docker; як правило, один шар – одна інструкція). Різні образи можуть містити одні і ті ж самі шари, оскільки кожен шар є надбудованим над іншим

образом, а два різні образи можуть використовувати один і той самий батьківський образ в якості основи. Образи зберігаються в Registry (реєстрах) і версіонуються за допомогою тегів (tag). Якщо тег не вказано, то за замовчуванням використовується тег «`latest`», який означає «останній образ, що був надісланий у цей реєстр» [6, 12].

На підставі вищевикладеного, на рисунку 2.2 проілюстроване співвідношення понять «контейнер» і «образ» Docker з використанням трьох контейнерів, що запущені з одного базового образу [11].

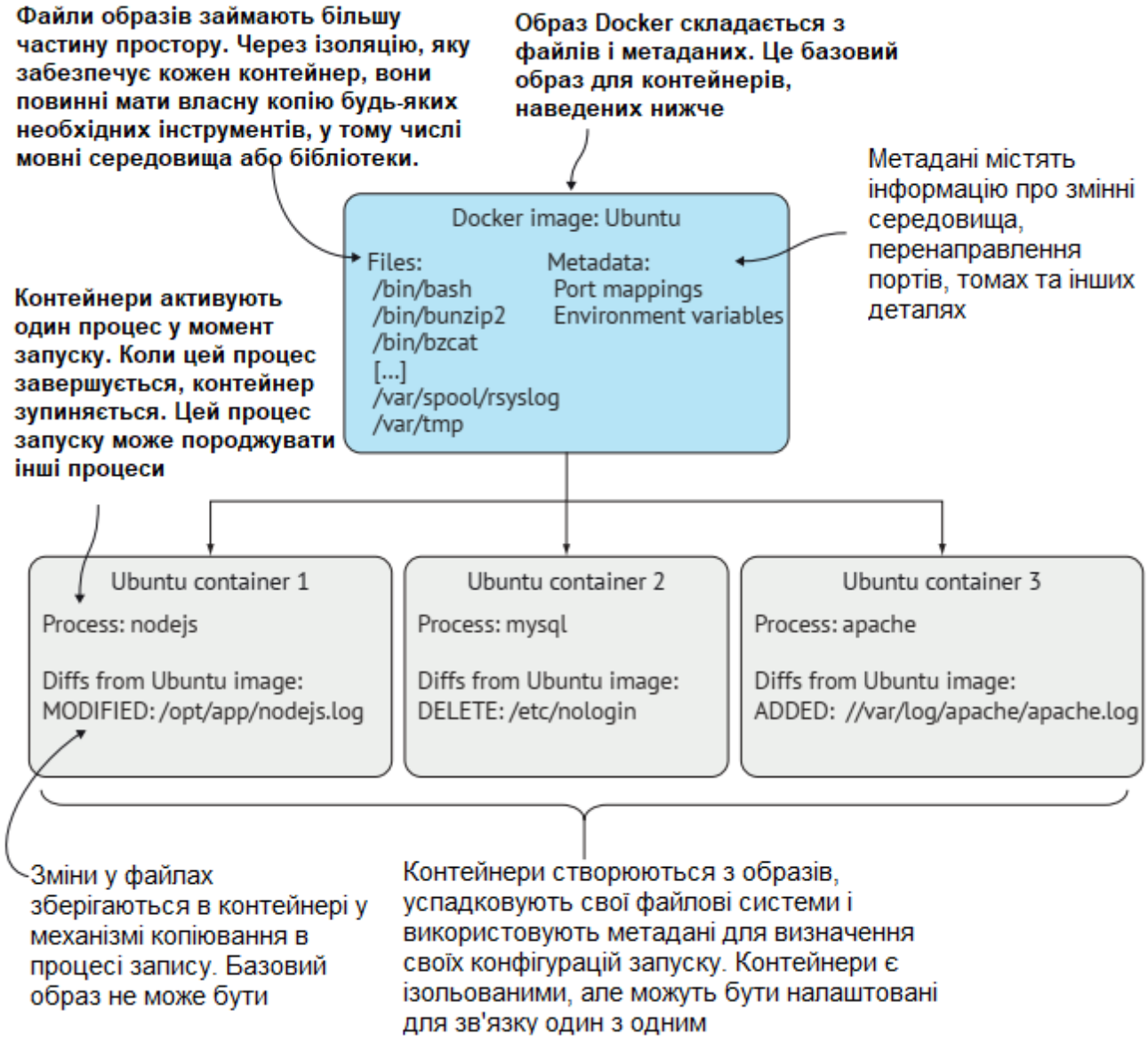

Рисунок 2.2 – Співвідношення понять «контейнер» і «образ» Docker

Один із підходів щодо трактування образів і контейнерів – це розглядати їх як програми та процеси, що у свою чергу випливає з основної функції Docker –

створювати, надсилати та запускати програмне забезпечення в будь-якому місці, де є Docker. Подібно до того, що процес можна розглядати як «додаток, який виконується», контейнер Docker можна розглядати як образ, що виконується Docker. Процес, запущений у контейнері, виконується всередині операційної системи хост-комп'ютера, але при цьому він ізольований від інших процесів. Для самого процесу це виглядає так, ніби він єдиний працює в системі. Для кінцевого користувача Docker – це програма з командним рядком, яку вони запускають, тож різноманітні операції із створення та запуску образів, перетворення їх на контейнери, збереження контейнера як образа або присвоєння тегів – робляться відповідними командами Docker. Основні команди, які необхідно використовувати на хост-комп'ютері при роботі з Docker, наведені в таблиці 2.1 [6, 11].

Таблиця 2.1 – Ключові команди Docker

Команда	**Призначення**
`docker build`	Зібрати образ Docker
`docker run`	Запустити образ Docker в якості контейнера
`docker commit`	Зберегти контейнер Docker в якості образа
`docker tag`	Присвоїти тег образу Docker

Ізоляція процесів у контейнерах здійснюється завдяки двом технологіям ядра Linux – просторам імен (Namespaces) і контрольним групам (CGroups). Namespaces гарантують, що процес буде працювати з власним поданням системи. Існує кілька типів просторів імен [6]:

- файлова система (mount, `mnt`) – ізолює файлову систему;
- UTS (UNIX Time-Sharing, `uts`) – ізолює ім'я хосту та доменне ім'я;
- ідентифікатор процесів (process identifier, `pid`) – ізолює процеси;
- мережа (network, `net`) – ізолює мережні інтерфейси;
- міжпроцесна взаємодія (`ipc`) – ізолює конкуруючу взаємодію між процесами;
- ідентифікатори користувача (`user`) – ізолюють ID користувачів і груп.

При цьому процес належить не одному Namespace, а одному Namespace кожного типу, а CGroups гарантують, що процес не буде конкурувати за ресурси, що зарезервовані за іншими процесами [6].

Слід зазначити, що під час випуску версії Docker 1.8 компанія Docker презентувала нову функцію управління надійністю вмісту контейнера, яка

перевіряє цілісність Docker-образу та справжність авторства сторони, що опублікувала цей образ. Управління надійністю вмісту є найважливішим компонентом для створення перевірених робочих процесів на основі образів, які завантажуються з реєстрів Docker [1].

2.2 Архітектура та компоненти Docker

Щоб зрозуміти, як найбільш ефективно використовувати Docker і деякі не цілком очевидні його властивості, необхідно мати уявлення про те, як влаштований Docker і яким чином організована спільна робота компонентів платформи, що приховані від користувача.

Ця платформа заснована на двох базових інструментах [1]:

- Docker Engine – відповідає за створення та функціонування контейнерів і надає ефективний і зручний інтерфейс для їх запуску;
- Docker Hub – хмарний web-сервіс для поширення контейнерів. Він надає величезну кількість образів контейнерів з відкритим доступом для завантаження, що дає змогу користувачам швидко почати роботу з ними та уникнути монотонної і одноманітної роботи, яка раніше вже була зроблена іншими людьми.

У Docker використовується архітектура клієнт/сервер, відповідно до якої клієнт взаємодіє з демоном Docker, а той надає всі необхідні клієнту послуги. На рис. 2.3 наведена архітектура встановленої та готової до використання платформи Docker, а також показані її найважливіші компоненти [1].

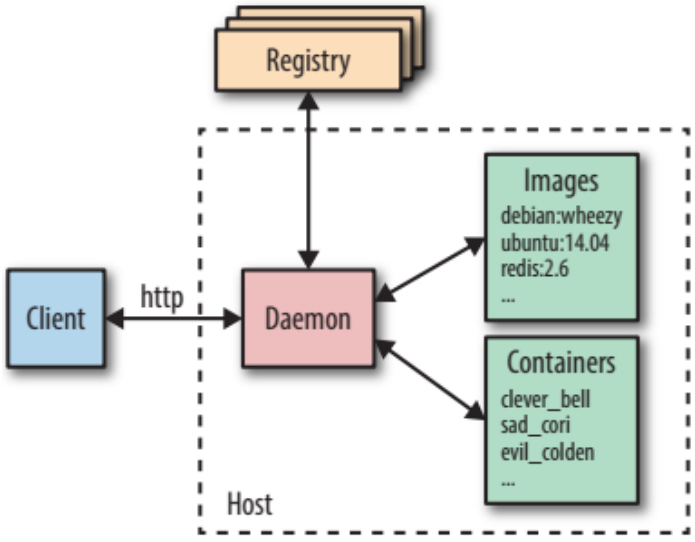

Рисунок 2.3 – Архітектура платформи Docker та її найважливіші компоненти

У центрі розташований демон Docker, який відповідає за створення, запуск і управління (контроль) роботою контейнерів, а також за створення і зберігання образів. Контейнери та образи були розглянуті вище та представлені в правій частині архітектури Docker на рисунку 2.3, а співвідношення їх понять показано на рис. 2.2. Демон запускається командою `docker daemon`, зазвичай про це дбає хостова ОС [1, 9].

Клієнт Docker є інтерфейсом користувача, або інтерфейсом командного рядка для взаємодій з демоном Docker за клієнт-серверним протоколом передачі гіпертексту (Hyper Text Transfer Protocol, HTTP). За замовчуванням це з'єднання встановлюється через сокет домену Unix/Linux, але також може використовуватися TCP-сокет для підтримки з'єднань із віддаленими клієнтами або дескриптор файлу для сокетів, якими здійснює управління система ініціалізації та менеджер служб `systemd` для ОС на базі Unix. `Systemd` призначений для управління процесами і службами, спрощуючи запуск, зупинку і моніторинг служб, а також їх конфігурацію. Оскільки всі операції обміну даними виконуються за протоколом HTTP, то можна без труднощів організувати з'єднання з віддаленими демонами Docker і розробити прив'язки (bindings) до потрібної мови програмування, але водночас слід враховувати особливості реалізації цих можливостей, наприклад обов'язкову наявність контексту створення (building context). Інтерфейси прикладного програмування, які використовуються для організації обміну даними з демоном, чітко визначені та детально документовані, що дає змогу розробникам писати програми, які взаємодіють безпосередньо з демоном, без використання клієнта Docker. Клієнт і демон Docker поширюються як окремі незалежні бінарні файли [1, 9].

Реєстри Docker – це репозитарії, які використовуються для зберігання і поширення образів. Реєстром, який обирають за замовчуванням, є Docker Hub, куди можна поміщати і звідки можна отримувати образи. Загалом на Docker Hub зберігаються тисячі загальнодоступних образів, а також керовані «офіційні» образи. Багато організацій створюють власні реєстри, які використовуються для зберігання комерційних і приватних образів і для усунення накладних витрат, що пов'язані із завантаженням образів через Інтернет. Демон Docker завантажує образи з реєстрів за командою `docker pull`. Крім того, він виконує автоматичне завантаження образів, які зазначені в команді `docker run` і в інструкції `FROM` файлу `Dockerfile`, якщо ці образи недоступні на локальній системі [1, 9].

Файл `Dockerfile` є простим текстовим файлом, що містить команди, які збирають образи Docker. За допомогою цих команд можна встановлювати додаткові програмні компоненти, налаштовувати змінні оточення, робочі каталоги і точку входу `ENTRYPOINT`, а також додавати новий код [9].

Зазначимо, що інструменти Docker Engine і реєстр Docker Hub самі по собі не надають завершеного повноцінного рішення для роботи з контейнерами. Для більшості користувачів знадобляться додаткові сервіси підтримки та допоміжне ПЗ, наприклад, система управління кластерами, інструменти виявлення сервісів, розширені мережні функціональні можливості та деякі інші [1, 9, 11].

Розглянемо додаткові найвідоміші технологічні інструменти, що надаються Docker, для управління контейнерами та їх розгортання [1, 9]:

- Docker Machine утиліта командного рядка для підтримки роботи Docker-хостів. Вона допомагає встановити і конфігурувати Docker-хости на локальних і віддалених ресурсах, а також конфігурує клієнта Docker, спрощуючи процедуру перемикання між середовищами. Управління вузлами забезпечується за допомогою команд `start`, `stop`, `inspect`, тощо;

- Docker Swarm – менеджер кластерів, дає змогу згрупувати кілька Docker-вузлів в один великий Docker-хост, після чого користувач може працювати з цим хостом як з єдиним ресурсом. Це окремий інструмент, який можна встановити за допомогою Docker Machine або вручну, отримавши образ Swarm. Наразі інструментарій Docker Swarm інтегрований у Docker Engine. Процес встановлення дуже простий, достатньо лише налаштувати диспетчер Swarm на всіх вузлах. Особливий інтерес представляє те, що розробник може просто дати команду Swarm запустити свої контейнери, а той сам вирішить, на яких вузлах їх запустити, позбавивши розробника всіх складнощів вибору. Для динамічного налаштування та управління службами в контейнерах можна використовувати службу виявлення. Інтегрований варіант називається режимом Swarm. Він діє подібно до інструменту Swarm, підтримує балансування навантаження і виявлення служб і може розглядатися як повноцінний механізм управління. Вмикається режим Swarm командою `init`, а додавання робочих вузлів здійснюється командою `join`;

- Docker Kitematic – являє собою графічний інтерфейс користувача для операційних систем Mac OS і Windows, який забезпечує запуск і управління контейнерів Docker;

- Docker Compose (компонувальник). Додатки часто складаються з безлічі компонентів, і відповідно вони будуть виконуватися в декількох контейнерах. За

допомогою інструментарію Docker Compose можна без проблем запустити додаток у декількох контейнерах. При цьому можна визначити оточення для додатку в загальному файлі `Dockerfile` та визначити перелік служб у файлі `docker-compose.yml`, після чого Docker автоматично створюватиме та запускатиме необхідні контейнери, як це визначено в даних файлах. Інструментарій Compose так само, як Docker Machine, має свій набір команд для управління службами додатку;

- Docker Trusted Registry – це локально встановлене ПЗ для зберігання та управління образами Docker. По суті це локальна версія реєстру Docker Hub, яку можна об'єднати з наявною інфраструктурою гарантування безпеки та узгодити з правилами зберігання і забезпечення захисту даних, що прийняті в конкретній організації. Усі функціональні можливості локального реєстру, зокрема різні метрики, управління доступом на основі ролей (Role-Based Access Control, RBAC) і реєстраційні журнали, контролюються через адміністративну консоль. Зазначимо, що на сьогодні це єдиний програмний додаток компанії Docker Inc., вихідний код якого є закритим.

2.3 Особливості інсталяції Docker в середовищах ОС Windows та Linux

2.3.1 Інсталяція Docker в середовищі ОС Windows

Останнім часом у сучасних версіях операційних систем Windows від компанії Microsoft, зокрема таких як Windows Pro, Windows Server 2016 та інші – з’явилася підтримка контейнерів, що працюють безпосередньо на Windows і запускають Windows всередині себе (так звана нативна підтримка). Це стало можливим завдяки тому, що Microsoft приєдналася до Open Container Initiative (OCI) – об’єднання компаній, яке розвиває спільну специфікацію формату контейнерних образів і механізму запуску контейнерів. Контейнери в Windows дають змогу запускати додатки, які ізольовані від решти системи, у переносних середовищах. Такі контейнери містять у собі усе необхідне для повноцінної роботи додатку [13].

Компанія Microsoft пропонує два типи контейнерів: контейнер Windows Server і контейнер Hyper-V. Обидва типи контейнерів можуть створюватися, функціонувати та управлятися однаковим чином. Крім того, обидва типи

контейнерів можна використовувати разом із такими технологіями Windows, як .NET або PowerShell, що раніше було неможливо. Відмінності між ними починаються на рівні ізоляції, яку кожен з них забезпечує по своєму.

Контейнери Windows Server працюють із розділенням ядра ОС на хост-машині, тобто всі контейнери, що працюють на цьому хості, використовують одне і те ж саме ядро. Водночас кожен контейнер підтримує своє власне уявлення про операційну систему, реєстр, файлову систему, IP-адреси та інші компоненти. Це поєднується з ізоляцією, яку кожному контейнеру надають за допомогою процесів, простору імен і технологій управління ресурсами [13].

Контейнер Windows Server добре підходить для ситуацій, коли і хостова операційна система, і додатки всередині контейнерів перебувають в межах однієї зони довіри – наприклад, для додатків, що охоплюють кілька контейнерів або формують єдину службу. Водночас ці контейнери залежать від процесу оновлення хостової ОС, що може ускладнити обслуговування. Наприклад, патч, застосований до хоста, може зашкодити роботі додатку в контейнері. Крім того, модель розділюваного ядра може розкрити систему для вразливості додатків і крос-контейнерних атак [13].

Контейнер Hyper-V вирішує ці проблеми, надаючи віртуальну машину, в якій потрібно запустити контейнер Windows. У цьому випадку контейнер більше не розділює ядро хосту і не має залежності від патчів хостової ОС. З одного боку використання контейнера Hyper-V призводить до деякої ефективності контейнеризації (зокрема, за швидкістю та пакуванням) у порівнянні з контейнером у Windows Server, але, з іншого боку, реалізується більш ізольоване та безпечне середовище [13].

Компанія Docker активно працює над повною інтеграцією контейнерів Windows в екосистему Docker. У межах цієї ініціативи Docker пропонує інструментарій Docker Engine для Windows, і Docker Client для Windows [13].

До того, що вже було сказано вище про Docker Engine, можна додати, що цей інструмент розширює функціональність, яка необхідна для управління оточенням Docker. Наприклад, дозволяє автоматизувати створення контейнерів з образів. Хоча можна створювати образи вручну, інструментарій Docker Engine пропонує додатково такі переваги, як можливість зберігання образів як коду, їх легкого відтворення або включення до циклу безперервної інтеграції в процесі розробки [13].

Однак Docker Engine не є компонентом Docker, який встановлюється разом із платформою під Windows Його потрібно завантажити, встановити та налаштувати окремо. Далі він буде працювати як служба Windows, для налаштування якої використовуються або файл конфігурації, або диспетчер управління службами Windows (Service Control Manager, SCM). Компанія Microsoft рекомендує використовувати файл конфігурації, а не SCM, але зазначає, що не кожен параметр конфігурації у файлі може бути застосовний до контейнерів Windows. Таким чином, на Docker Engine по суті лежить уся рутинна робота по здійсненню управління контейнером [13].

Інструмент Docker Client також було описано вище. Як зазначалося, він являє собою інтерфейс командного рядка, який надає набір команд для управління образами і контейнерами. Це ті ж самі команди, які дають змогу створювати та запускати контейнери Docker у Linux. Незважаючи на те, що не можна запустити контейнер для Windows на ОС Linux або контейнер Linux на ОС Windows, однак можна використовувати один і той же клієнт для управління як Linux-, так і Windows-контейнерами (обома типами: Windows Server або Hyper-V) [13].

Розглянемо процес інсталяції Docker версії 17.03.0 на 64-ьох розрядні версії ОС Windows 10 Pro, Enterprise або Education. Зазначимо, що в даному випадку для правильного встановлення Docker необхідно також увімкнути підтримку віртуалізації Hyper-V [9, 14].

Після завантаження необхідного інсталяційного пакета Docker for Windows з [14] запускаємо його встановлення. У результаті з’явиться вікно, яке запропонує підтвердити згоду з ліцензійною угодою, як це показано на рис. 2.4. Необхідно прийняти ліцензійну угоду, встановивши прапорець у нижній частині діалогу, а потім продовжити інсталяцію Docker натисканням на кнопку Install (Встановити) [9].

Після встановлення на екрані монітора з’явиться виринаюче повідомлення «Docker is starting» («Запуск Docker»). Після завершення запуску з’явиться діалогове вікно, де буде вказано «Docker is now up and running» («Docker зараз запущений і працює») (рис. 2.5). На цьому процес встановлення Docker вважається завершеним. Перевірити встановлену версію платформи Docker можна шляхом використання інтерфейсу командного рядка. Для цього слід запустити термінал і виконати команду `docker --version`, як це показано на рис. 2.6 [9].

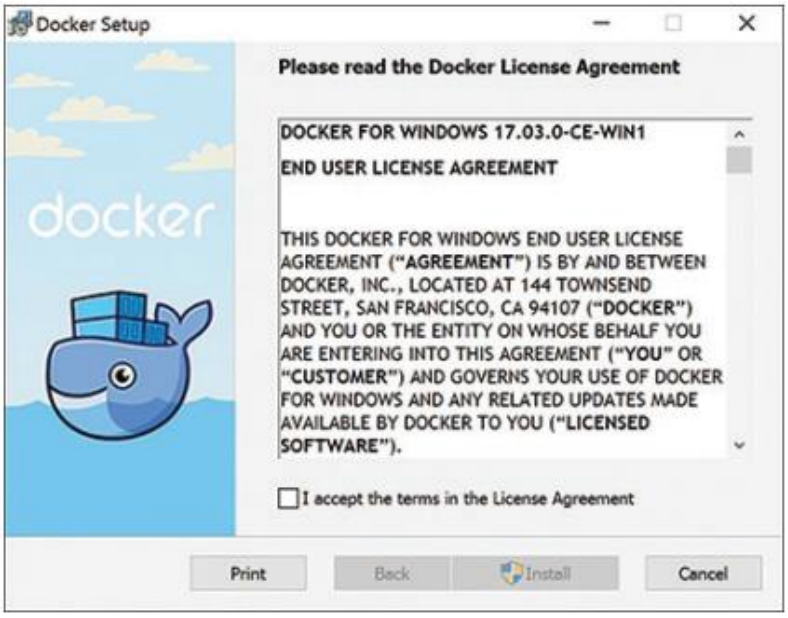

Рисунок 2.4 – Вікно початку процесу інсталяції Docker

Рисунок 2.5 – Діалогове вікно завершення установки Docker

Рисунок 2.6 – Перевірка версії встановленої платформи Docker

Як можна бачити в процесі виведення команди – була встановлена версія Docker 17.03.0. Щоб отримати список команд, що підтримуються, необхідно виконати команду `docker --help`, після чого будуть виведені всі доступні команди, як це показано на рис. 2.7 [9].

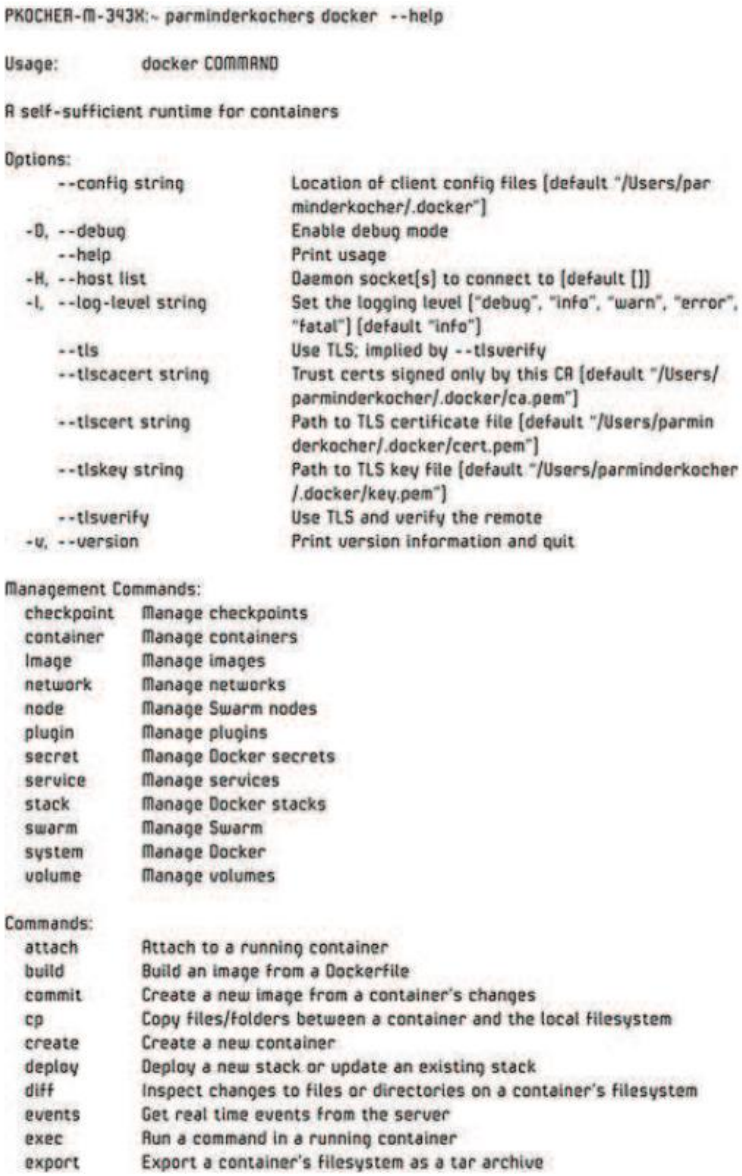

Рисунок 2.7 – Команди Docker, що є доступними

2.3.2 Інсталяція Docker в середовищі ОС Ubuntu Linux

Розглянемо процес інсталяції тієї ж самої версії Docker 17.03.0 на 64-ьох розрядні версії ОС Ubuntu, зокрема такі як: 64-розрядні версії Ubuntu: Trusty 14.04; Yakkety 16.10; Xenial 16.04 [15 - 17].

У разі необхідності, перевірити версію ОС можна шляхом виконання команди `lsb_release-a`, як це показано на рис. 2.8 [9].

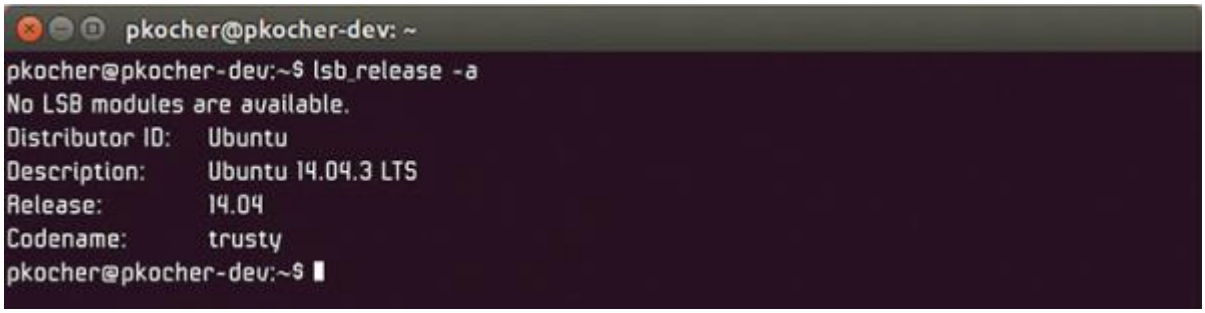

Рисунок 2.8 – Перевірка версії ОС Ubuntu

Таким чином, далі будемо орієнтуватися на встановлення платформи Docker на ОС Ubuntu Trusty 14.04, але інструкції, що тут розглядаються, також підходять і для інших двох вищезазначених версій операційних систем. При використанні ОС Ubuntu Trusty 14.04 рекомендується встановити пакети `linux-imageextra -*`, якщо вони раніше не були встановлені. Ці пакети дозволять Docker використовувати драйвери сховищ AUFS пристроїв зберігання, які за замовчуванням використовує Docker в Ubuntu. В інших дистрибутивах за замовчуванням використовується Device Mapper. Драйвери сховищ відповідають за абстракцію сховища для контейнерів. Вони працюють на рівні файлового сховища та забезпечують файлову систему контейнерів [9, 17].

Для встановлення додаткових пакетів `linux-imageextra -*` необхідно виконати команду `$ sudo apt-get update`, яка завантажить останні версії пакетів (рис. 2.9) [9].

Далі для їх інсталяції потрібно виконати команду [9, 16]:

```
$ sudo apt-get install \
    linux-image-extra-$(uname -r) \
    linux-image-extra-virtual
```

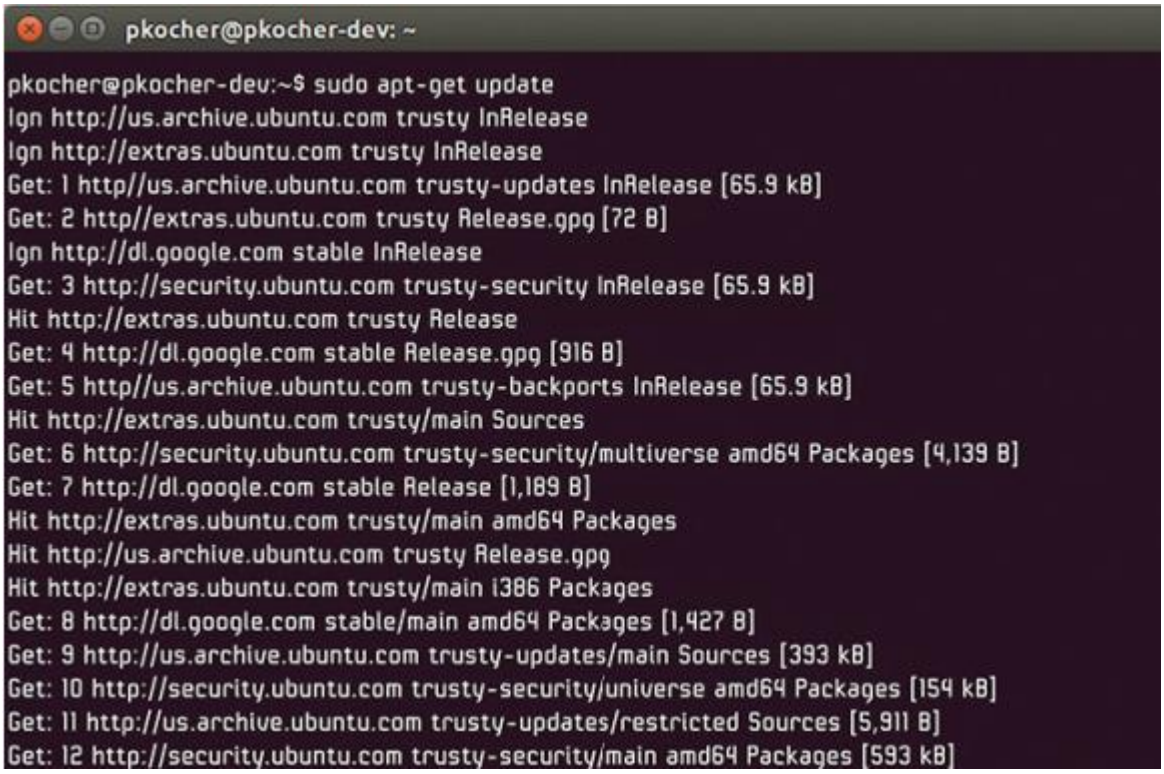

Рисунок 2.9 – Встановлення додаткових пакетів

Ця команда встановить завантажені оновлення, після чого можна розпочинати інсталяцію платформи Docker. Для інсталяції доступні дві редакції: Docker CE (Community Edition) та Docker EE (Enterprise Edition). У нашому випадку встановлюється платформа Docker CE [9].

Спочатку необхідно встановити репозиторій Docker, звідки потім можна буде виконати інсталяцію платформа Docker CE. Для цього встановлюється пакет, який необхідний для доступу до репозиторію через HTTPS. Це робиться з використанням команди [9]:

```
$ sudo apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \ software-properties-common
```

Далі потрібно додати ключ GPG (GNU Privacy Guard) для доступу до офіційного репозиторію Docker. Цей ключ є реалізацією криптографії з відкритим ключем, яку можна використовувати, як для більш стандартних операцій з ключами

шифрування (шифрування/дешифрування), так і для перевірки повідомлень за допомогою підпису. Додавання ключа здійснюється командою [9, 16]:

```
$ curl -fsSL https://download.Docker.com/linux/ubuntu/gpg | sudo apt-key add
```

Після додавання ключа необхідно перевірити його контрольну суму. Для цього використовуємо команду `$ sudo apt-key fingerprint 0EBFCD88` (рис. 2.10) [9, 16].

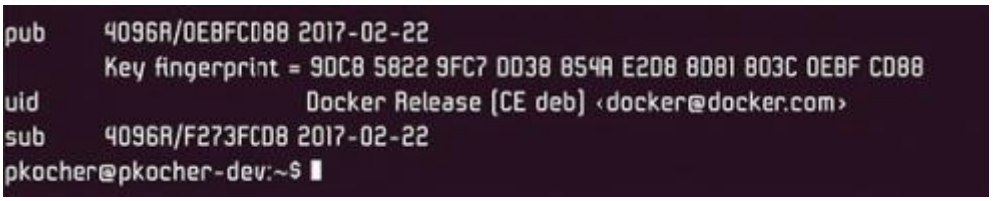

Рисунок 2.10 – Перевірка контрольної суми ключа

Далі необхідно додати репозиторій Docker до джерел утиліти APT (Advanced Packaging Tool), яка в Linux-подібних системах виконує встановлення, оновлення пакетів та відстеження їх залежностей. Для цього використовуємо команду [9, 16]:

```
$ sudo add-apt-repository "deb [arch=amd64] <-DOCKER-EE-URL> \
  $(lsb_release -cs) \ stable-"
```

Після додавання репозиторію необхідно оновити індекс пакетів [9, 16]:

```
$ sudo apt-get update
```

І нарешті, можна інсталювати потрібну версію платформи Docker CE, використовуючи для цього команду: `$ sudo apt-get install Docker-ce` (рис. 2.11) [9].

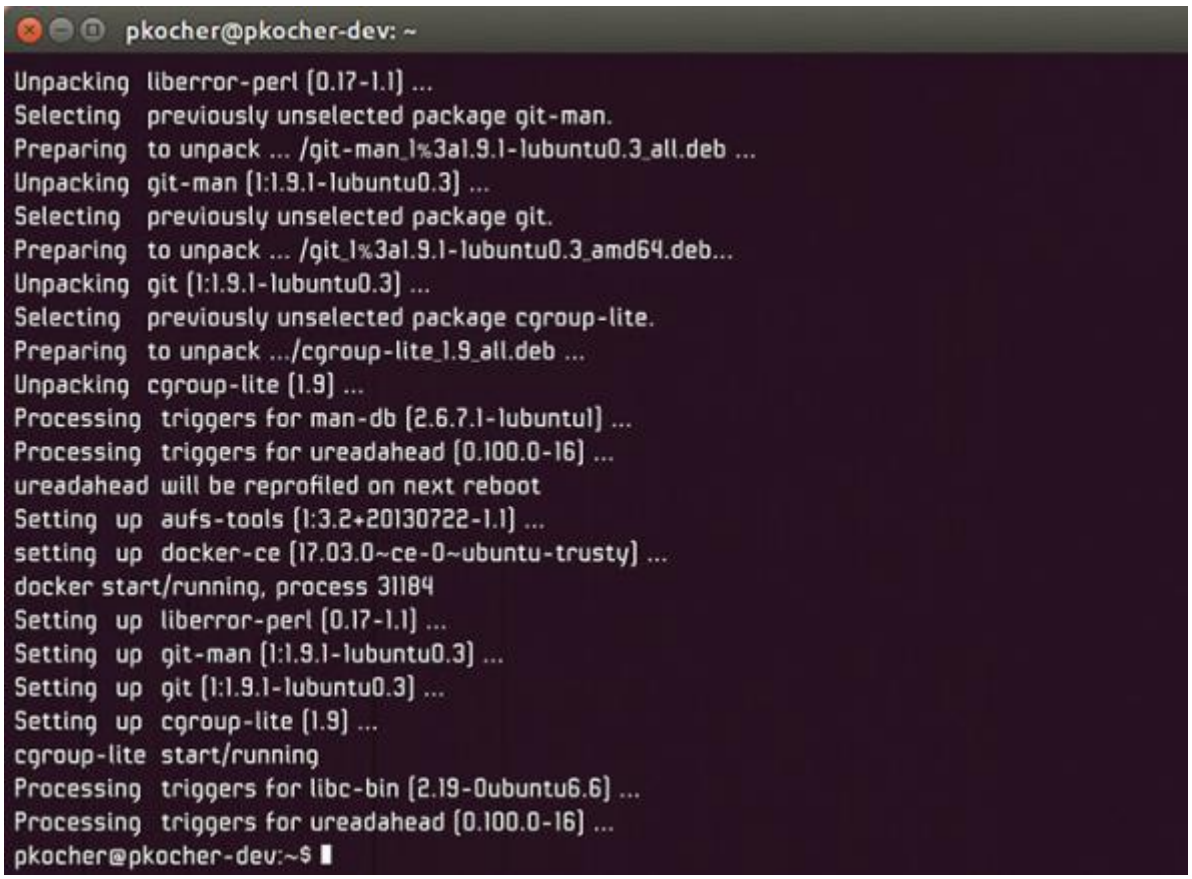

Рисунок 2.11 – Інсталяція платформи Docker CE

Після завершення інсталяції платформи Docker в ОС Ubuntu Linux можна виконати перевірку її версії, використовуючи команду `$ docker --version` (рис. 2.12) [9].

Рисунок 2.12 – Перевірка версії встановленої платформи Docker

3 ПРАКТИЧНІ ПРИНЦИПИ СТВОРЕННЯ І ЗАПУСКУ ДОДАТКІВ DOCKER

3.1 Постановка задачі та обґрунтування розробки додатку

Розглянемо принципи створення додатку у форматі To-Do із використанням образу Docker. Підхід створення додатків у форматі To-Do є зручним та ефективним засобом для забезпечення управління проектами, задачами та списками. Додатки To-Do спрямовані на те, щоб надати користувачам можливість створювати, організовувати та відстежувати свої задачі та списки справ на різних пристроях. У нашому випадку додаток буде представлений у простому web-інтерфейсі. Він має забезпечувати зберігання та відображення коротких рядків інформації, які можна у процесі позначити як виконані [11, 18].

Схематично процес створення такого Docker-додатку показаний на рис. 3.1 [11].

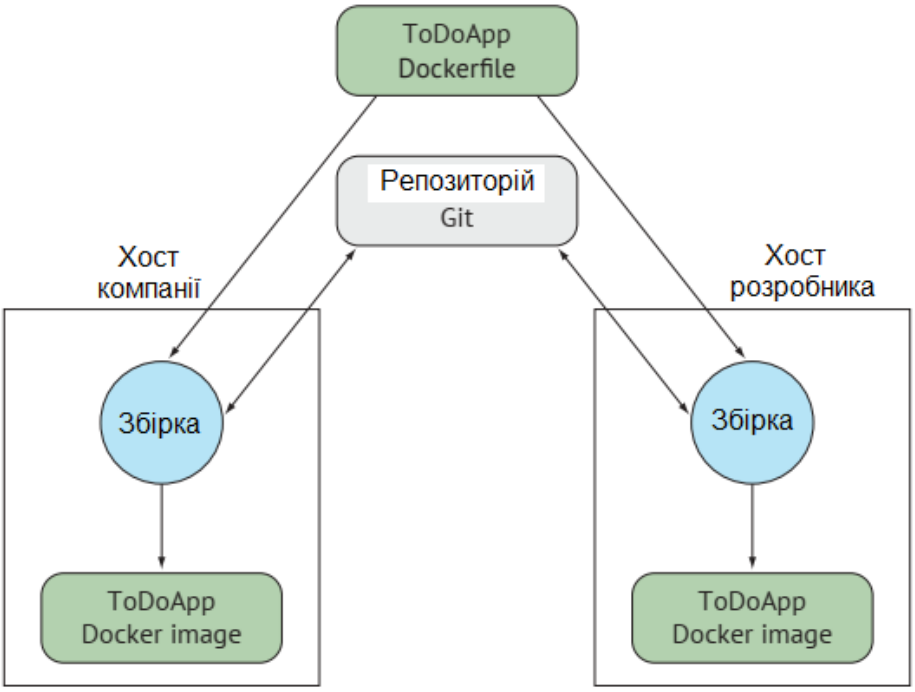

Рисунок 3.1 – Створення Docker-додатку

Далі покажемо, що з одного невеликого файлу `Dockerfile` можна впевнено створювати, запускати, виконувати та зупиняти додаток як на своєму хості, так і на хості компанії, не переймаючись проблемами з розгортанням додатку або іншими можливими залежностями. Це важлива особливість

платформи Docker – надання середовища розробки, що є надійним і спільним у використанні, та яким легко здійснювати управління. Це, у свою чергу, означає, що більше не потрібні складніші та досить часто неоднозначні інструкції з розгортання одного і того ж додатку на різних системах [11].

3.2 Базові методи створення нового образу Docker

Існує чотири базових методи створення образів Docker.

1) Із використанням команд Docker. Тут спочатку необхідно запустити контейнер за допомогою команди `docker run` , а далі, використовуючи командний рядок, ввести команду `docker commit` для створення нового образу. Ця команда створює новий образ контейнера, що редактується, у локальній системі [11, 19]:

```
docker commit <conatainer id> <username/imagename>
```

Після цього його можна завантажити на `docker hub` і використовувати в інших проектах. Цей метод підходить, коли необхідно перевірити або контролювати весь процес встановлення. Водночас потрібно вести записи про здійснені дії, щоб у разі потреби мати змогу повернутися до потрібного етапу створення образу [11, 19].

2) На основі `Dockerfile`. Тут необхідно виконати збірку з відомого базового образу і вказати її за допомогою обмеженого набору простих команд.

3) На основі `Dockerfile` та інструментів управління конфігурацією. Суть використання цього методу така ж, як і в попередньому випадку, але контроль над складнішою збіркою передається інструментарію, що здійснює управління конфігурацією. Це доцільно, коли функцій `Dockerfile` недостатньо для образу, що створюється [11].

4) Використання «нульового» образу. Тут необхідно стерти образ та із порожнього образу імпортувати `tar`-архів із потрібними файлами. Ці архіви виступають як стандартний засіб для пакування, переносу та зберігання даних. Крім збирання образів, Docker використовує `tar`-архіви також і для інших

операцій, таких як передача даних між хостами чи контейнерами, а також для тимчасового зберігання інформації [11, 20].

Операція створення tar-архіву, як правило, використовується для пакування даних перед передачею в Docker-контейнери або з них. Ця операція в Docker виконується за наступною командою [20]:

```
tar -cvf archive.tar /path/to/directory,
```

У процесі її застосування для створення tar-архіву використовуються такі ключі [20]:

- «-c» – створення нового архіву;
- «-v» – деталізований вивід (показує файли, що включаються в архів);
- «-f» – вказує ім’я архіву.

Розглянемо приклад створення tar-архіву [20]:

```
docker run --rm -v /my/data:/data alpine tar -cvf /data/my-archive.tar /data/source
```

У цьому прикладі запускається контейнер на основі мінімального образу Alpine Linux з ключем «--rm», щоб контейнер видалився після виконання. Каталог /my/data хоста монтується в контейнер на /data. За допомогою команди tar, що зазначена вище, створюється архів my-archive.tar, що містить дані з /data/source [20].

Зворотна операція вилучення даних дозволяє працювати із вмістом tar-архівів у контейнерах Docker. Ця операція виконується за командою [20]:

```
tar -xvf archive.tar
```

У процесі її застосування для вилучення даних із tar-архиву використовуються наступні ключі [20]:

- «-x» – вилучає вміст архіву;
- «-v» – деталізований вивід;
- «-f» – вказує ім’я архіву.

Розглянемо приклад вилучення даних із tar-архіву [20]:

```
docker run --rm -v /my/data:/data alpine tar -xvf /data/my-archive.tar -C /data/extracted
```

У цьому прикладі вилучаються файли з my-archive.tar у каталог /data/extracted на хості. Тут ключ «-C» вказує на призначення вилучення [20].

Як зазначалося вище, tar-архіви можуть бути корисними у разі створення образів Docker. Вони можуть використовуватись для копіювання великих обсягів даних або для роботи з вихідними кодами додатків. У цілому алгоритм роботи з tar-архівом включає таку послідовність кроків: спочатку потрібно скопіювати архів у контейнер, далі вилучити його у задану директорію і потім видалити архів для економії місця. Наведемо приклад використання tar-архіву в Dockerfile [20]:

```
FROM alpine:latest

# Копіюємо архів у контейнер
COPY ./my-archive.tar /tmp/my-archive.tar

# Вилучаємо архів
RUN tar -xvf /tmp/my-archive.tar -C /var/www

# Видаляємо архів після вилучення
RUN rm /tmp/my-archive.tar
```

Таким чином, цей варіант реалізується на основі «нульового» образу шляхом накладення набору файлів із tar-архіву, які необхідні для запуску образу. Це потрібно, якщо, наприклад, треба імпортувати набір автономних файлів, що створені в іншому місці. Слід зазначити, що цей метод у масовому використанні Дуже мало застосовується [11].

Тому далі зупинимось на методі, що заснованому на Dockerfile.

3.3 Створення та застосування `Dockerfile`

Вже зазначалося, що `Dockerfile` є текстовим файлом, який містить серію команд, що дозволяють зібрати базовий образ. На рис. 3.2 наведено структуру файлу Dockerfile та зроблено опис його вмісту, який ми далі будемо використовувати. Зазначимо, що файл із назвою «`Dockerfile`» необхідно створювати у новій папці [11].

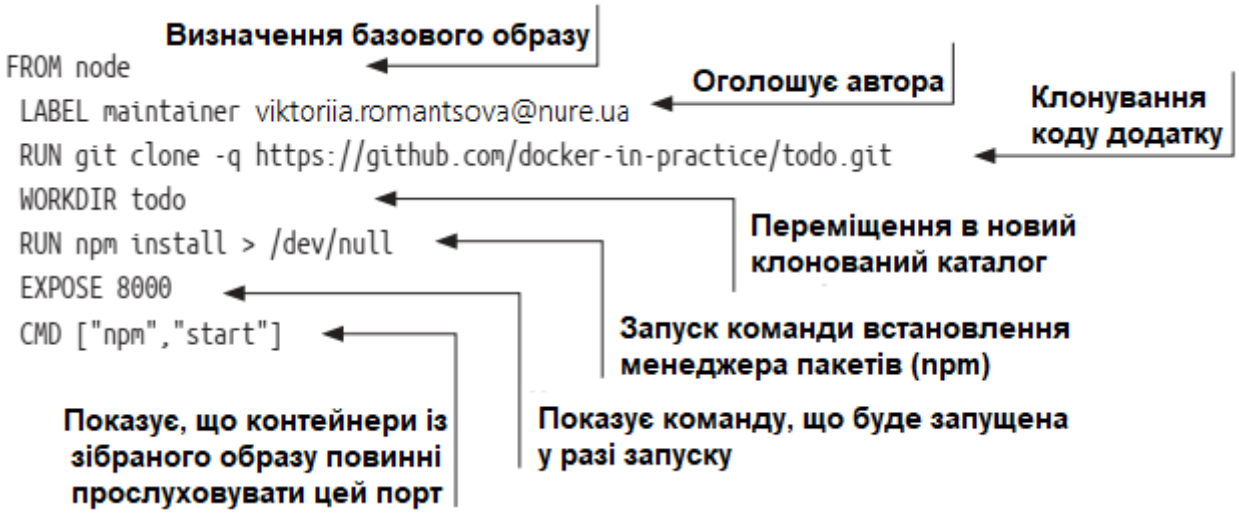

Рисунок 3.2 – Структура `Dockerfile` та його вміст

Файл `Dockerfile` починається з визначення базового образу за допомогою команди `FROM`. У наведеному на рис. 3.2 прикладі використовується образ `Node.js`, тобто в цьому випадку забезпечується доступ до двійкових файлів `Node.js`. Офіційний образ `Node.js` має назву `node` [11].

Далі йде оголошення автора за допомогою команди `LABEL`. У цьому випадку, як правило, використовується адреса електронної пошти розробника або будь-яке посилання, що вказуватиме на нього. Цей рядок не є обов'язковим для створення робочого образу Docker, але рекомендується його включити. На цьому етапі збірка успадкувала стан `node`-контейнера [11].

Після цього виконується клонування коду додатку за допомогою команди `RUN`. Ця команда використовується для отримання коду додатку, здійснюючи запит до віддаленого репозиторію `git` всередині контейнера. У цьому випадку `git` встановлюється всередині базового образу [11].

Далі переходимо до нового клонованого каталогу за допомогою команди WORKDIR. Це не лише змінює каталоги в контексті збірки, але також остання команда WORKDIR визначає каталог, у якому ми перебуватимемо за замовчуванням при запуску контейнера із зібраного образу [11].

Далі виконується запуск команди встановлення менеджера пакетів Node.js (npm). Пакетний менеджер (npm) відповідає за встановлення залежностей для нашого додатку. Для цього зазвичай виконується команда npm install. Щоб Docker зробив це автоматично, потрібно вказати команду RUN: `RUN npm install.` У розглянутому прикладі нас виведення не цікавить, тому його перенаправляють у каталог /dev/null [11].

Додаток використовує порт 8000, тому застосовується команда EXPOSE щоб повідомити Docker, що контейнери із зібраного образу мають слухати цей порт. Після цього використовуємо команду CMD щоб повідомити Docker, яка команда буде виконана під час запуску контейнера [11].

Таким чином, у наведеному на рис. 3.2 прикладі можна побачити кілька ключових особливостей, що є характерними для платформи Docker і файлів Dockerfiles. Зокрема, Dockerfile – це проста послідовність обмеженого набору команд, які виконуються в суворому порядку. Це впливає на файли та метадані створеного образу. Тут команда RUN впливає на файлову систему, перевіряючи та встановлюючи додатки, а команди EXPOSE, CMD та WORKDIR впливають на метадані образу [11].

3.4 Збірка образу та запуск контейнера Docker

Вище нами була визначена послідовність кроків щодо створення файлу Dockerfile. Тепер розглянемо, як створити з нього образ Docker, використовуючи команду docker build. Наприклад, якщо потрібно зібрати образ із поточного каталогу з назвою FROM node та тегом todoapp, формат команди буде наступним [9, 11]:

```
docker build -t FROM node:todoapp.
```

Тут [9]:

- «-t» ключ, що визначає ім'я образу, що створюється (у даному випадку FROM node), який має форму ImageName:<Tag> ;

- <Tag> – ім'я тегу, у цьому випадку todoapp;

- крапка в кінці (.) визначає каталог (у цьому випадку – поточний), де знаходяться необхідні файли.

Якщо збірка образу здійснюється з іншого каталогу my_repositoriy, який розташований на цьому ж хості, то формат команди дещо зміниться [21]:

```
docker build -t my_repositoriy/FROM node:todoapp
```

Якщо є потреба зробити збірку образу з віддаленого репозиторію Git, у якому розташований Dockerfile, тоді формат запису буде таким [22]:

```
docker build -t FROM node:todoapp https://github.com/username/repo.git#node
```

Результат виконання цієї команди у випадку використання Dockerfile, який був створений вище (рис. 3.2), буде виглядати приблизно так, як це показано на рис. 3.3 [11].

Таким чином, ми створили образ Docker із власним ідентифікатором («66c76cea05bb»). Якщо у разі звернення до нього є незручності, тоді можна для зручності присвоїти йому тег, як це показано на рис. 3.4.

Далі після збірки образу Docker з файлу Dockerfile та присвоєння йому тегу, можна запустити його як контейнер, використовуючи команду docker run (рис. 3.5). Ця команда створює та запускає контейнер із образу. Ключ «-p» перенаправляє порт контейнера 8000 на порт 8000 хост-комп'ютера, тому тепер можна перейти у своєму браузері за адресою http://localhost:8000 для перегляду додатку [11].

Ключ «--name» присвоює контейнеру унікальне ім'я, до якого можна у разі потребі звертатися. Останній аргумент «example1 todoapp» – це ім'я образу. Щойно контейнер буде запущений, то натисканням комбінації клавіш <Ctrl-C> можна завершити процес і контейнер.

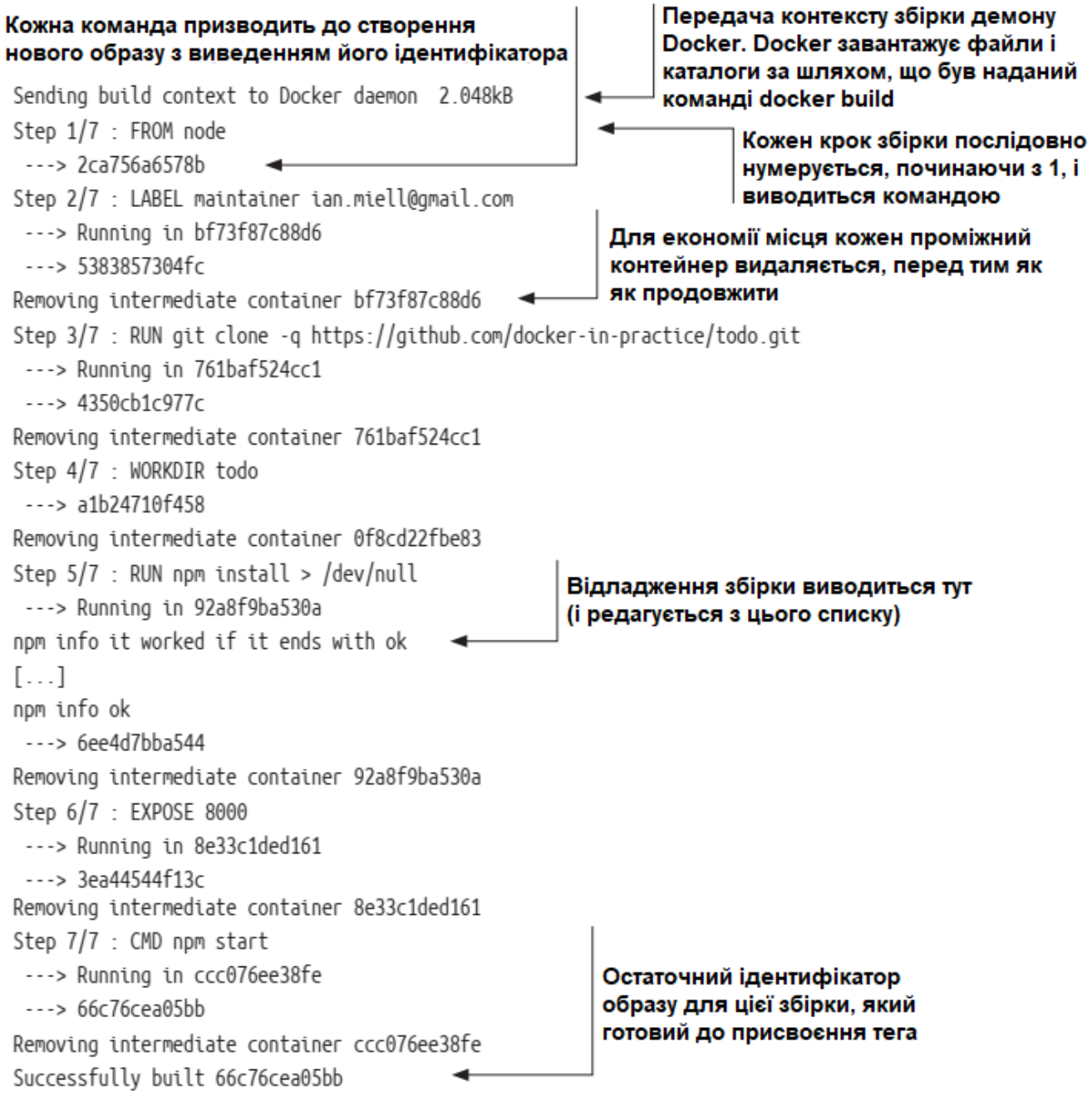

Рисунок 3.3 – Образ Docker із створеного `Dockerfile`

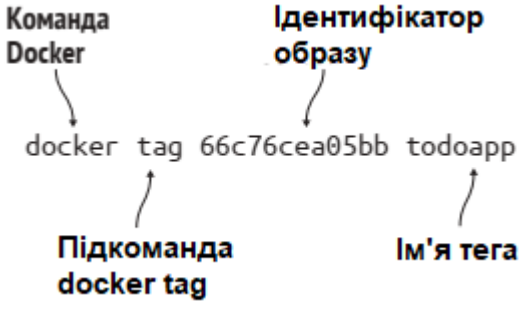

Рисунок 3.4 – Застосування команди `docker tag`

```
$ docker run -i -t -p 8000:8000 --name example1 todoapp
npm install
npm info it worked if it ends with ok
npm info using npm@2.14.4
npm info using node@v4.1.1
npm info prestart todomvc-swarm@0.0.1

> todomvc-swarm@0.0.1 prestart /todo
 > make all

npm install
npm info it worked if it ends with ok
npm info using npm@2.14.4
npm info using node@v4.1.1
npm WARN package.json todomvc-swarm@0.0.1 No repository field.
npm WARN package.json todomvc-swarm@0.0.1 license should be a valid SPDX
➥ license expression
npm info preinstall todomvc-swarm@0.0.1
npm info package.json statics@0.1.0 license should be a valid SPDX licen
➥ expression
npm info package.json react-tools@0.11.2 No license field.
npm info package.json react@0.11.2 No license field.
npm info package.json node-
     jsx@0.11.0 license should be a valid SPDX license expression
npm info package.json ws@0.4.32 No license field.
npm info build /todo
npm info linkStuff todomvc-swarm@0.0.1
npm info install todomvc-swarm@0.0.1
npm info postinstall todomvc-swarm@0.0.1
npm info prepublish todomvc-swarm@0.0.1
npm info ok
if [ ! -e dist/ ]; then mkdir dist; fi
cp node_modules/react/dist/react.min.js dist/react.min.js

LocalTodoApp.js:9: // TODO: default english version
LocalTodoApp.js:84: fwdList = this.host.get('/TodoList#'+listId);
 // TODO fn+id sig
TodoApp.js:117: // TODO scroll into view
TodoApp.js:176: if (i>=list.length()) { i=list.length()-1; } // TODO
➥ .length
local.html:30: <!-- TODO 2-split, 3-split -->
```

Команда `docker run` запускає контейнер, `-p` перенаправляє порт контейнера 8000 у порт 8000 на хост-комп'ютері, `--name` присвоює контейнеру унікальне ім'я, а останній аргумент - це ім'я образу

Виведення процесу запуску контейнера здійснюється на термінал

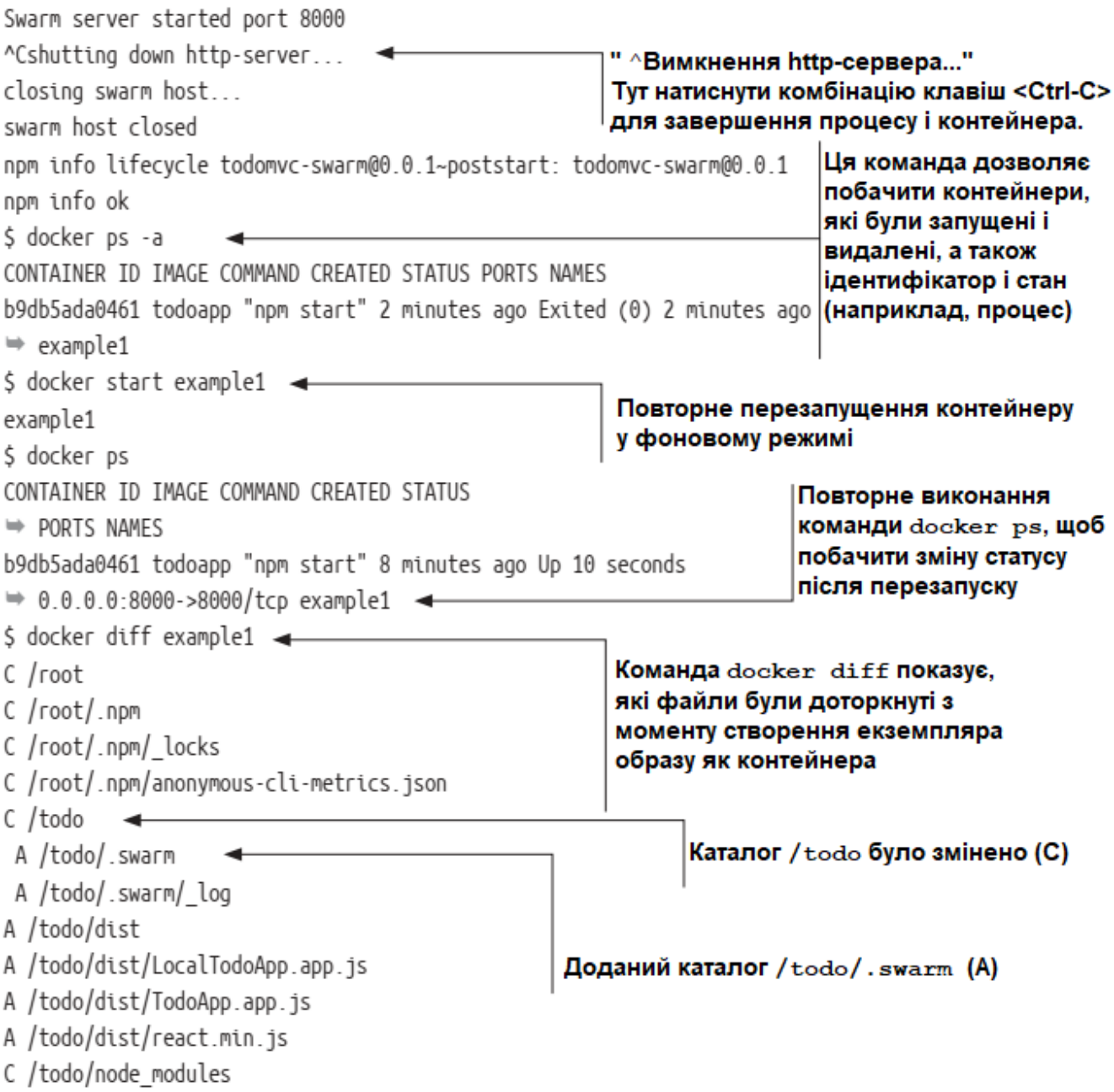

Рисунок 3.5 – Запуск образу Docker як контейнера

Для відображення всіх запущених контейнерів, які не були видалені, можна застосувати команду `docker ps`. При додаванні до команди ключа «`-a`» у список будуть включені всі контейнери, що створені у системі. Виконання команди виводить такі параметри [21]:

- `ID` (ідентифікатор), тобто кодове значення конкретного контейнера;
- `IMAGE` – образ, який використовує контейнер;
- `COMMAND` – список команд, що мають виконуватися одразу після запуску;

- CREATED – статус і час, коли контейнер був створений (наприклад, «8 minutes ago» – «8 хвилин тому»);

- STATUS – поточний стан (чи увімкнено контейнер і час його роботи з моменту останнього запуску). Тут може відображатися код виходу і час, коли це сталося;

- PORTS – порти, що використовуються контейнером зі списку;

- NAMES – більш зрозумілі у порівнянні з ID імена (в прикладі – це «example1»), які можна так само використовувати для виконання команд або пошуку.

Слід звернути увагу, що кожен контейнер має свій ідентифікатор (ID) та статус (STATUS), аналогічний процесу. У прикладі на рис. 3.5 це статус «Exited», але його можна перезапустити, як це можна бачити далі. Після перезапуску можна побачити, що статус змінився на «Up», і тепер видно порт, що перенаправляється з контейнера до хост-комп'ютера: 0.0.0.0:8000->8000/tcp example1 [11].

Контейнери є ідемпотентними, тобто внесені зміни всередині запущеного контейнера будуть втрачені під час наступного його запуску. Тому за необхідності всі внесені зміни всередині запущеного контейнера потрібно дописати у Dockerfile і перезібрати образ Docker. Список змін всередині контейнера можна отримати за допомогою команди docker diff <container_name>, яка показує, які файли були змінені з моменту створення екземпляра образу example1 як контейнера. Тут слід пам'ятати, що з точки зору Ubuntu / Linux каталог також є файлом. На рис. 3.5 можна побачити, що кожен рядок на початку містить один із трьох можливих символів – A, C, D [23]:

- A – були додані файли: .swarm, LocalTodoApp.app.js, react.min.js, та інші;

- C – файл було змінено (у прикладі на рис. 3.5 – це каталог todo);

- D – файл був видалений.

У прикладі на рисунку 3.5 нічого не видаляли, тому в результаті виводу є лише рядки з символами «A» і «C».

Таким чином, за результатами команда docker diff виведе лише ті файли, які були додані/змінені/видалені. Якщо потрібно отримати всю історію дій

у запущеному контейнері Docker, то у цьому випадку необхідно використати команду [23]:

```
docker exec difftest cat /root/.ash_history
```

Таким чином, завдяки команді `diff` можна швидко отримати уявлення про те, що змінювалося, а завдяки файлу з історією команд – побачити кожен крок змін, що вносилися [23].

З вищенаведеного можна побачити, що Docker «містить» середовище розробника для розгортання та запуску додатків, що свідчить про його здатність розглядати це середовище як сутність, над якою можна передбачувано виконувати певні дії. Це надає платформі Docker можливість впливати на життєвий цикл ПЗ: від початку процесу розробки до введення його в експлуатацію і далі – на процеси обслуговування [11].

Ще однією ключовою концепцією платформи Docker є його так звані шари, які розглядаються далі.

3.5 Шари Docker

При використанні Docker у глобальному масштабі виникає проблема, що пов’язана з розгортанням великої кількості додатків, оскільки кожному з них потрібна копія файлів для зберігання у певному місці. Тобто ця проблема буде пов’язана з наявністю фізичного дискового простору. За замовчуванням платформа Docker для зменшення обсягу необхідного дискового простору використовує свій спеціальний механізм копіювання у процесі запису (див. рис. 3.6). Тут щоразу, коли працюючому контейнеру потрібно виконати запис у файл, він фіксує зміну шляхом копіювання елемента в нову область диску. Після завершення фіксації нова область диска «заморожується» і записується як шар із власним ідентифікатором [11].

Це деяким чином дає розуміння того, чому контейнери Docker швидко запускаються – їм нічого копіювати, оскільки всі дані вже збережені у вигляді образу [11].

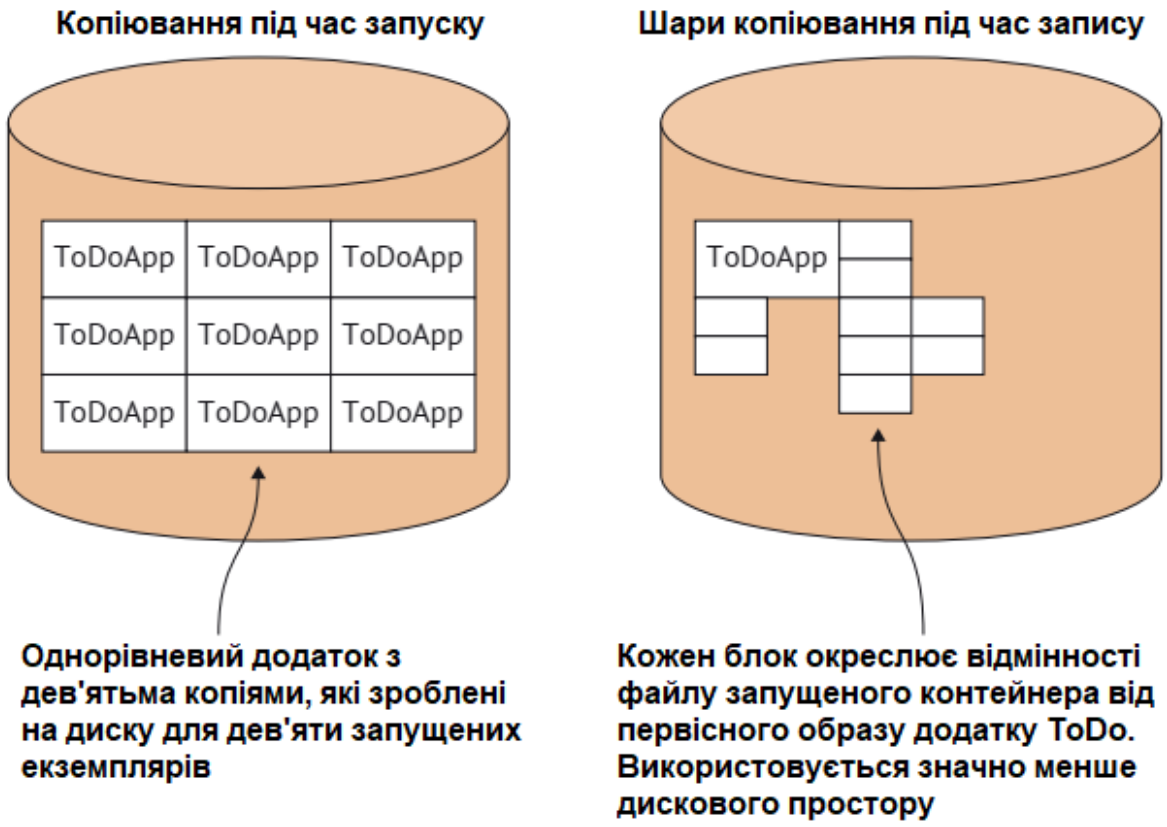

Рисунок 3.6 – Шарова структура файлової системи Docker

Така стратегія оптимізації, як копіювання у процесі запису, є стандартною стратегією, що використовується в обчислювальній техніці. У разі створення нового об’єкта (будь-якого типу) з шаблону, а не простого копіювання всього необхідного набору даних, копіювання даних здійснюється лише після їх зміни. Такий підхід, залежно від варіанта використання, може зекономити значні апаратні ресурси [11].

На рис. 3.7 показано, що створений додаток `ToDo` містить три шари, на які слід звернути увагу. Шари статичні, тому, якщо необхідно щось змінити у вищому шарі, можна просто виконати збірку поверх образу, який ви хочете взяти за посилання. У додатку був створений загальнодоступний `node`-образ і поверх нього зроблені багаторівневі зміни зверху [11].

Усі три шари можуть спільно використовуватися кількома запущеними контейнерами, так само як загальна бібліотека може спільно використовуватися в пам’яті кількома запущеними процесами. Це дуже важлива функція для операцій, яка дозволяє запускати численні контейнери на основі різних образів на хостах, не відчуваючи нестачі дискового простору.

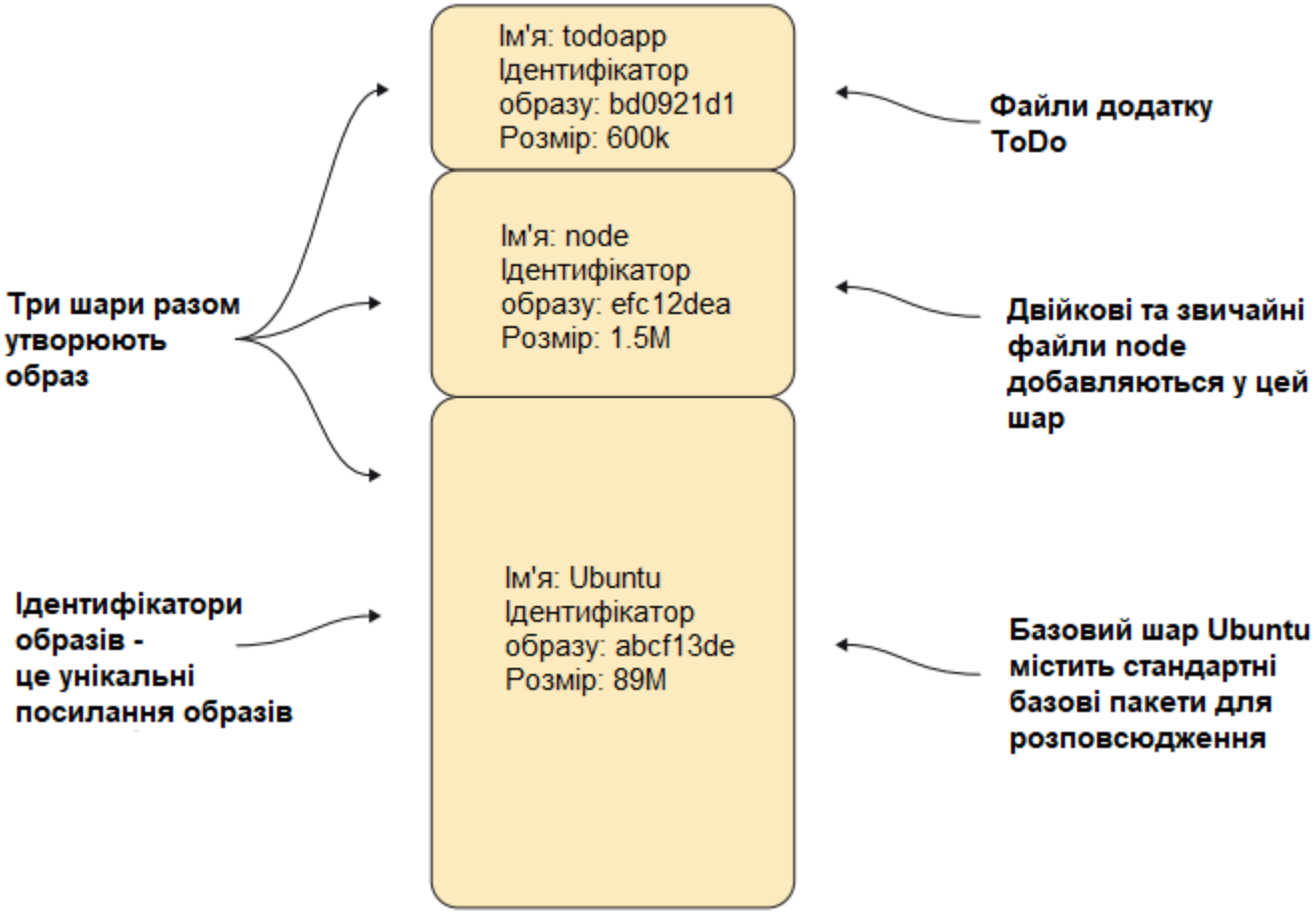

Рисунок 3.7 – Варіант утворення шарів у файловій системі платформи Docker

Наприклад, припустимо, що додаток ToDo запускається як сервіс для платних клієнтів у режимі реального часу. Його можна масштабувати для великої кількості користувачів. Або ж, якщо цей додаток буде використаний розробником ПЗ, то можна запустити багато різних програмних середовищ на локальному комп’ютері одночасно. Якщо використовувати додаток для проходження тестування, то можна одночасно виконувати набагато більше тестів і значно швидше, ніж раніше. Усі ці приклади демонструють важливу роль використання концепції Docker-шарів.

ВИСНОВКИ

В бакалаврській кваліфікаційній роботі розглянуті і проаналізовані загальні принципи застосування платформи Docker для реалізації процесів контейнеризації стосовно створення і контролю середовища для розгортання та запуску додатків. Зазначено, що контейнеризація, як механізм віртуалізація на рівні ОС – це технологія, яка дає змогу запускати програмне забезпечення в ізольованих на рівні ОС просторах. У процесі виконання роботи основна увага приділялась, як загальному аналізу особливостей появи та напрямів розвитку технологій контейнеризації, так і безпосередньо аналізу архітектури та компонентів платформи Docker, її компонентів та елементів. Особлива увага у роботі була приділена дослідженню практичних принципів створення і запуску додатків на платформі Docker у вигляді контейнерів.

Так у першому розділі кваліфікаційної роботи описані етапи розвитку та загальні особливості технології контейнеризації. Показано, що процеси контейнеризації беруть свій початок з 1979 року минулого сторіччя з появи у ядрі UNIX системного виклику `chroot`. Він надав змогу ізолювати процес і його дочірні елементи від інших частин ОС і саме тому вважається однією з перших технологій контейнеризації. А технології, що орієнтовані на контейнери, як одну із форм віртуалізації, що на цей час є найпоширенішою, набули своєї популярності лише в 2013 році, коли було презентовано першу версію платформи Docker [1, 6].

Контейнери – це досить новий підхід реалізації процесів віртуалізації, що почав конкурувати із традиційними технологіями на основі віртуальних машин. Причому з першого погляду досить часто вважається, що контейнери це спрощений варіант віртуалізації на основі VM. Тому у роботі приділена увага розгляду відмінностей контейнерів від VM та виділені їх переваг у використанні. Показано, що між застосуванням VM і контейнерів існують принципові відмінності, які стосуються мети їх використання. Механізми використання VM орієнтовані на емуляцію всієї ОС, тоді як контейнери, що розгорнуті на хост-комп’ютері, спільно використовують ядро його хостової ОС, що робить їх більш ефективними, швидкими та зручними у використанні ресурсів.

У другому розділі кваліфікаційної роботи розглянуті базові поняття, архітектура, компоненти та особливості інсталяції платформи Docker. Зазначено, що до базових понять Docker, які відображають його ключові особливості, відносяться поняття образів, контейнерів, шарів і процесів. Показано, що контейнери запускають системи, що визначені образами. Образи зберігаються в Registry (реєстрах) і версіонуються за допомогою тегів (tag). Один із підходів щодо трактування образів і контейнерів – це розглядати їх як програми та процеси. Процес, запущений у контейнері, виконується всередині операційної системи хост-комп'ютера, але при цьому він ізольований від інших процесів [11].

Під час аналізу архітектури Docker показано, що ця платформа заснована на двох базових інструментах: Docker Engine і Docker Hub. Також розглянуті та описані і інші інструменти що надаються Docker, для управління контейнерами та їх розгортання, такі як: Docker Machine, Docker Swarm, Docker Kitematic, Docker Compose, Docker Trusted Registry. Серед компонентів платформи виділені демон Docker, клієнт Docker, реєстри або репозитарії Docker, образи (`Dockerfile`).

Також у другому розділі розглянуті і проаналізовані особливості інсталяції Docker в середовищах ОС Windows та Linux. Зокрема наданий процес інсталяції платформи Docker версії 17.03.0 на 64-розрядні версії ОС Windows 10 Pro, Enterprise або Education [9, 14]. А також процес інсталяції Docker 17.03.0 на 64-розрядні версії ОС Ubuntu Linux [15 - 17].

У третьому розділі кваліфікаційної роботи запропоновані практичні принципи створення і запуску додатків Docker. Зокрема розглядалися принципи створення із використанням Docker-образу додатку у форматі To-Do. У процесі послідовного створення середовища для розгортання і запуску додатку в якості контейнера були спочатку проаналізовані базові методи створення нового образу Docker. Для подальшої роботи був вибраний метод на основі `Dockerfile`, у разі застосування якого необхідно виконати збірку з відомого базового образу і вказати її за допомогою обмеженого набору простих команд. Були описані основні практичні принципи та команди створення та застосування `Dockerfile`. Далі був із `Dockerfile` створений образ Docker, а після збірки образу Docker з файлу `Dockerfile` та присвоєння йому тегу, були надані практичні рекомендації щодо запуску запустити його як контейнеру. На всіх кроках процесу створення і запуску додатка як контейнера Docker описувалися всі основні команди і їх ключі, та надавалися рекомендації щодо їх застосування.

Таким чином, створивши і запустивши додаток на платформі Docker, можна бачити ефективність її використання для створення середовища розгортання додатку на всіх етапах робочого процесу. Відтворення та спільне використання таких середовищ і можливість їх розміщення в різних місцях надають гнучкість розробнику не лише в процесі створення додатків, а також дозволяють контролювати самі середовища в процесі розгортання та здійснення запуску додатків.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Моуэт Э. Использование Docker. Разработка и внедрение программного обеспечения при помощи технологии контейнеров / Эдриен Моуэт – М.: ДМК Пресс, 2017. – 354 с.

2. Путеводитель по Docker. От основ контейнеризации до создания собственного докера [Електронний ресурс] // Habr. – 2024. – Режим доступу до ресурсу: https://habr.com/ru/articles/810777/.

3. Волгин В. Про использование CI/CD в Docker и Jenkins для тестировщика 2023 [Електронний ресурс]. – 2023. – Режим доступу до ресурсу: https://vc.ru/dev/659295-pro-ispolzovanie-ci-cd-v-docker-i-jenkins-dlya-testirovshika-2023.

4. Siddiqui A. Контейнеризация с помощью Docker и Kubernetes [Електронний ресурс] / Areeba Siddiqui. – 2024. – Режим доступу до ресурсу: https://open.zeba.academy/konteynerizatsiya-docker-kubernetes/.

5. Как использовать Docker при разработке программного обеспечения? [Електронний ресурс] // < / >itProger: онлайн школа обучения IT профессиям. – 2024. – Режим доступу до ресурсу: https://itproger.com/news/kak-ispolzovat-docker-pri-razrabotke-programmnogo-obespecheniya.

6. Основы контейнеризации (обзор Docker и Podman) [Електронний ресурс] // – Режим доступу до ресурсу: https://habr.com/ru/articles/659049/.

7. Amri Aymen Недостающее введение в контейнеризацию [Електронний ресурс] / Aymen Eon Amri // Habr. – 2021. – Режим доступу до ресурсу: https://habr.com/ru/articles/541288/.

8. Введение в Docker и Kubernetes: основы контейнерных технологий. Часть 1 [Електронний ресурс] // Habr: Блог Цифровой СИБУР Kubernetes. – 2024. – Режим доступу до ресурсу: https://habr.com/ru/companies/sibur_official/articles/826964/.

9. Кочер П.С. Микросервисы и контейнеры Docker / П.С. Кочер. М.: ДМК Пресс, 2019. – 240 с.

10. Савельев Алексей Решения Microsoft для виртуализации ИТ-инфраструктуры предприятий [Електронний ресурс] / Алексей Савельев // Учебный Internet-курс. – 2012. – Режим доступу до ресурсу: https://intuit.ru/studies/courses/2324/624/info.

11. Милл И. Docker на практике / Иан Милл, Эйдан Хобсон – М.: ДМК Пресс, 2020. – 516 с.

12. Schmidt J. Docker Best Practices: Using Tags and Labels to Manage Docker Image Sprawl [Електронний ресурс] / Jay Schmidt // Docker Inc. – 2024. – Режим доступу до ресурсу: https://www.docker.com/blog/docker-best-practices-using-tags-and-labels-to-manage-docker-image-sprawl//

13. Windows контейнеры и Docker [Електронний ресурс] // K2 Cloud. – 2019. – Режим доступу до ресурсу: https://k2.cloud/blog/about-technologies/windows-konteynery-i-docker/.

14. Install Docker Desktop on Windows [Електронний ресурс] / dockerdocs // Docker Inc. – Доступ здійснено 26.05.2025. – Режим доступу до ресурсу: https://docs.docker.com/desktop/setup/install/windows-install/.

15. Install Docker Engine on Ubuntu [Електронний ресурс] / dockerdocs // Docker Inc. – Доступ здійснено 26.05.2025. – Режим доступу до ресурсу: https://docs.docker.com/engine/install/ubuntu/#install-docker-ce-1.

16. Установка Docker на Ubuntu [Електронний ресурс] // GitHub, Inc. – Доступ здійснено 26.05.2025. – Режим доступу до ресурсу: https://gist.github.com/ashtaev/38f25e922d356bf0de8cc51421aa8021.

17. Марков О. Управление драйверами Docker [Електронний ресурс] / Олег Марков // PurpleSchool. – 2025. – Режим доступу до ресурсу: https://purpleschool.ru/knowledge-base/article/driver.

18. Что такое Microsoft To Do и зачем оно нужно? [Електронний ресурс] // Skypro wiki. – Доступ здійснено 27.05.2025. – Режим доступу до ресурсу: https://sky.pro/wiki/lifestyle/chto-takoe-microsoft-to-do-i-zachem-ono-nuzhno/

19. Важные команды для работы с Docker [Електронний ресурс] // Zomro. – Доступ здійснено 27.05.2025. – Режим доступу до ресурсу: https://zomro.com/rus/blog/faq/303-spisok-vazhnyh-komand-dlja-raboty-s-docker.

20. Петров И. Работа с tar-архивами в Docker [Електронний ресурс] / Иван Петров // PurpleSchool. – 2025. – Режим доступу до ресурсу: https://purpleschool.ru/knowledge-base/article/tar.

21. Space Police Основные команды Docker [Електронний ресурс] / Police Space // АО «ТаймВэб». – 2020. – Режим доступу до ресурсу: https://timeweb.com/ru/community/articles/osnovnye-komandy-docker.

22. Андреев Р. Основные команды Docker, которые вам необходимо знать для эффективной работы [Електронний ресурс] / Роман Андреев // ООО «ТАЙМВЭБ.КЛАУД». – 2023. – Режим доступу до ресурсу: https://timeweb.cloud/tutorials/docker/komandy-docker-spisok.

23. Yevhen Lebid Docker совет №22: Получаем список изменений в контейнере [Електронний ресурс]. – 2018. – Режим доступу до ресурсу: https://ealebed.github.io/posts/2018/docker-совет-22-получаем-список-изменений-в-контейнере/.