Технології створення Web-застосунків

Модуль 5.

Доц. Попівщий В.І. ЗНУ каф.ПЗАС 2020

ОСНОВНІ ДЖЕРЕЛА

1. Хорсдал К. Микросервисы на платформе .NET. – СПб. : Питер, 2018. 352 с.
2. Ньюмен С. Создание микросервисов. – СПб.: Питер, 2016. – 304 с.
3. Фаулер М. Шаблоны корпоративных приложений. – М.: Вильямс, 2016. – 544 с.
4. Ричардсон К. Микросервисы. Паттерны разработки и рефакторинга. – СПб.: Питер, 2019. 544 с.
5. Cole Matt R. Hands-On Microservices with C#. – Packt Publishing, 2018. – 234 p.
6. Newman Sam Monolith to Microservices. – O'Reilly Media, 2020. – 256 p.
7. Gaurav Aroraa, Ed Price Hands-On Microservices with C# 8 and .NET Core 3 Third Edition. – Packt Publishing, 2020. – 451 p.

8. Morgan Bruce, Paulo A. Pereira **Microservices in Action**. – Manning Publications, 2019. – 366 p.
9. Essentials of Microservices Architecture. Paradigms, Applications, and Techniques. - CRC Press: 2020. - 293 p.
10. Tayo Koleoso Beginning Quarkus Framework: Build Cloud-Native Enterprise Java Applications and Microservices. – Apress, 2020. – 297p. https://scanlibs.com/beginning-quarkus-framework-cloud-native-apps/
11. Binildas Christudas Practical Microservices Architectural Patterns. – Apress, 2019. – 902 p.
12. Моуэт Т. Использование Docker – М.: ДМК Пресс, 2017. – 354 с.
13. .NET Microservices: Architecture for Containerized .NET Applications. – Microsoft Corporation, 2020

ТРЕНІНГИ

- Тренінг «Мікросервіси для Java розробників» https://itcluster.lviv.ua/paged/training-microservices-for-java-developers/
- Курси Microservices https://www.nobleprog.com.ua/kursy-microservices

Огляд відеокурсів

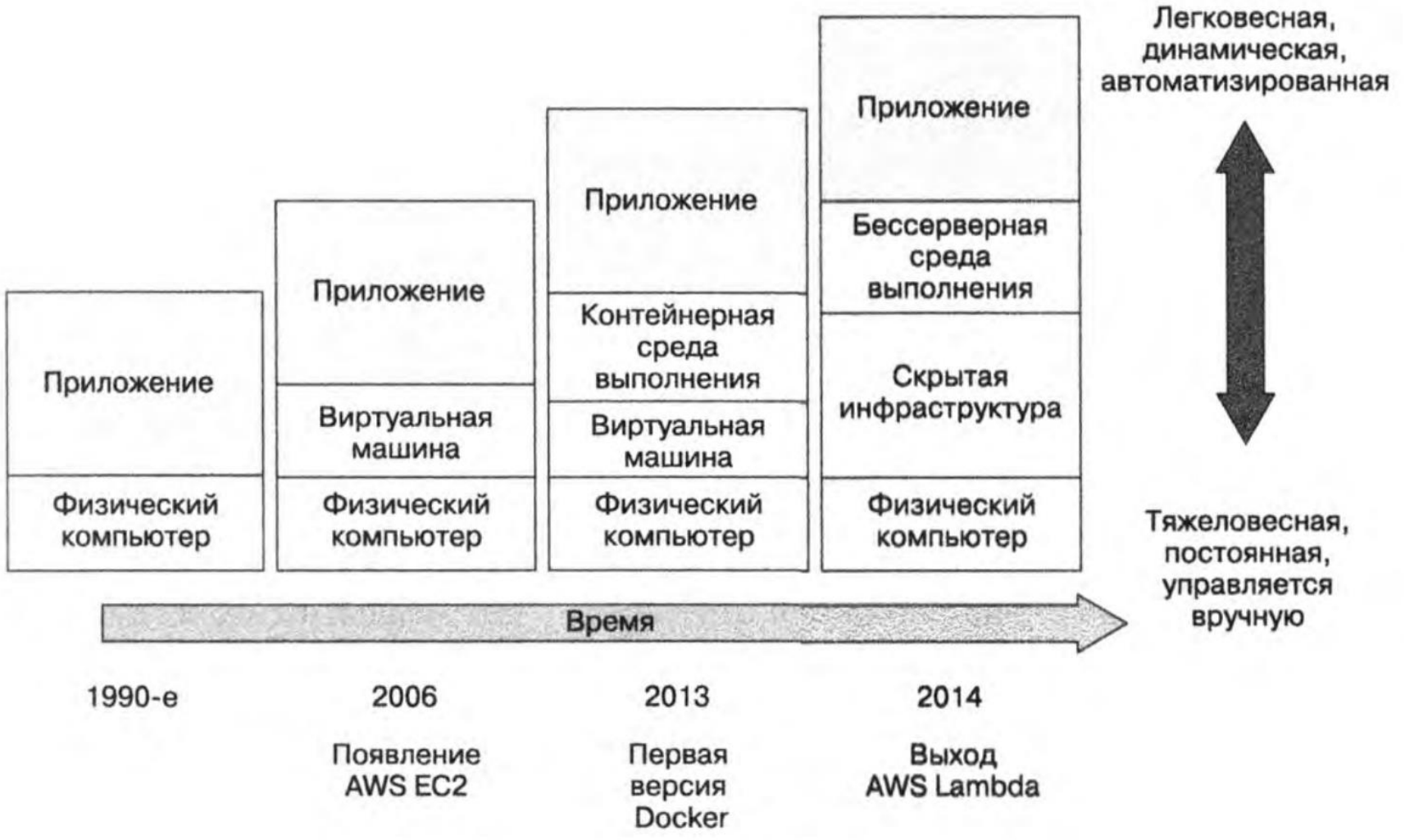
1. Linkedin - Learning Creating Your First Spring Boot Microservice 2019
2. **Lynda - Azure Microservices with .NET Core for Developers 2020**
3. Pluralsight - Building Microservices (2019)
4. Pluralsight – Microservices The Big Picture (2018)
5. ASP.NET Core 2.0 E-commerce Web Site Based on Microservices (2019)
6. Lynda - Microservices Design Patterns (2020)
7. Packt – A Beginner's Guide to a Microservices Architecture
8. O'Reilly - Building Microservice Systems with Docker and Kubernetes
9. Lynda – Java Microservices with GraalVM (2020)

10. Lynda - Software Architecture - Domain-Driven Design (2019)
11. Lynda – Microservices - Asynchronous Messaging (2019)
12. Lynda - Azure Service Fabric for Developers (2020)
13. Pluralsight - Microservices Fundamentals (2019)
14. Udemy - Build Microservices with NET Core & Amazon Web Services (2019)

Модуль 5. Розгортання мікросервісів. Azure DevOps. Docker.

- **Розгортання** - це поєднання двох взаємопов'язаних концепцій - процесу і архітектури.
- **Процес** розгортання полягає в доставці коду в промислове середовище і складається з етапів, які повинні виконати люди - розробники або системні адміністратори.
- **Архітектура** розгортання визначає структуру середовища, в якому цей код буде виконуватися.

Процес перекидання розробниками коду на робочий сервер вручну став високоавтоматизованим.



Вступ до контейнерів та Docker

(.NET Microservices: Architecture for Containerized .NET Applications. – Microsoft Corporation, 2020)

- Контейнеризація - це підхід до розробки програмного забезпечення, при якому програма чи сервіс, її залежності та конфігурація (абстраговані як файли маніфесту розгортання) упаковуються разом як образ контейнера.
- Контейнеризовану програму можна протестувати як одиницю та розгорнути як екземпляр образу контейнера в головній операційній системі (ОС).
- Подібно до того, як транспортні контейнери дозволяють перевозити товари кораблем, поїздом або вантажівкою незалежно від вантажу всередині, контейнери програмного забезпечення виступають як стандартна одиниця розгортання програмного забезпечення, яка може містити різні коди та залежності.

- Контейнеризація програмного забезпечення таким чином дозволяє розробникам та IT-спеціалістам розгортати їх в різних середовищах з незначними або зовсім без змін.
- Контейнери також ізолюють програми один від одного на загальній ОС. Контейнерні програми працюють поверх хоста контейнера, який, у свою чергу, працює в ОС (Linux або Windows). Отже, контейнери мають значно менший розмір, ніж образи віртуальних машин (VM).
- Кожен контейнер може запускати цілу веб-програму або сервіс, як показано на Рис. 2-1. У цьому прикладі хост Docker є хостом контейнера, а App1, App2, Svc 1 і Svc 2 - контейнеризовані програми або сервіси.

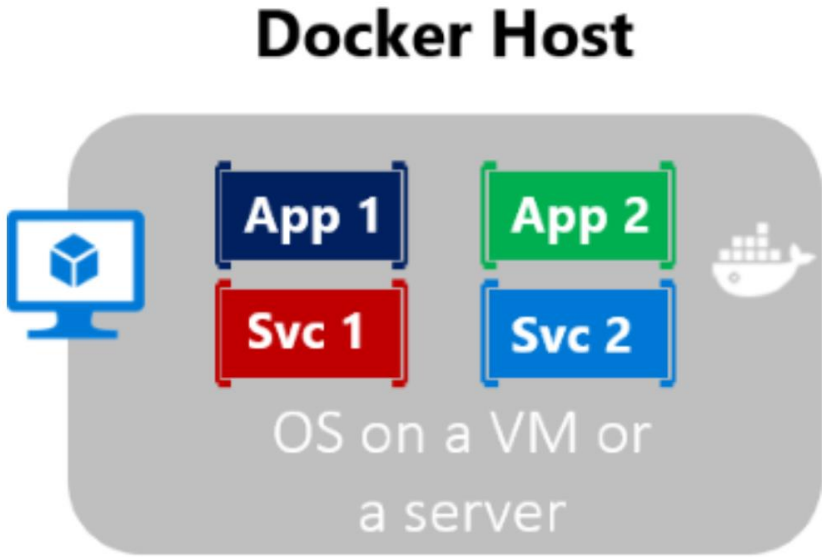


Figure 2-1. Multiple containers running on a container host

Ще однією перевагою контейнеризації є масштабованість. Ви можете швидко масштабуватися, створюючи нові контейнери для короткочасних завдань. З точки зору програми, створення екземпляра образу (створення контейнера) подібне до створення такого процесу, як сервіс або веб-програма.

- Коротше кажучи, контейнери пропонують переваги ізоляції (isolation), портативності (portability), гнучкості (agility), масштабованості (scalability) та контролю протягом усього робочого циклу програми.
- Найважливішою перевагою є ізоляція навколишнього середовища від Dev до Ops.

Що таке Docker?

- Docker - це проект з відкритим кодом для автоматизації розгортання додатків як портативних, самодостатніх контейнерів, які можуть працювати в хмарі або локально.
- Docker - це також компанія, яка просуває та розвиває цю технологію, працюючи у співпраці з постачальниками хмар, Linux та Windows, включаючи Microsoft.
- Контейнери Docker можуть працювати де завгодно, локально в центрі обробки даних клієнта, у зовнішнього постачальника послуг або в хмарі на Azure. Контейнери образів Docker можуть запускатися в Linux і Windows. Однак образи Windows можуть працювати лише на хостах Windows, а образи Linux - на хостах Linux та хостах Windows (на сьогоднішній день за допомогою віртуальної машини Hyper-V Linux), де host означає сервер або віртуальну машину.

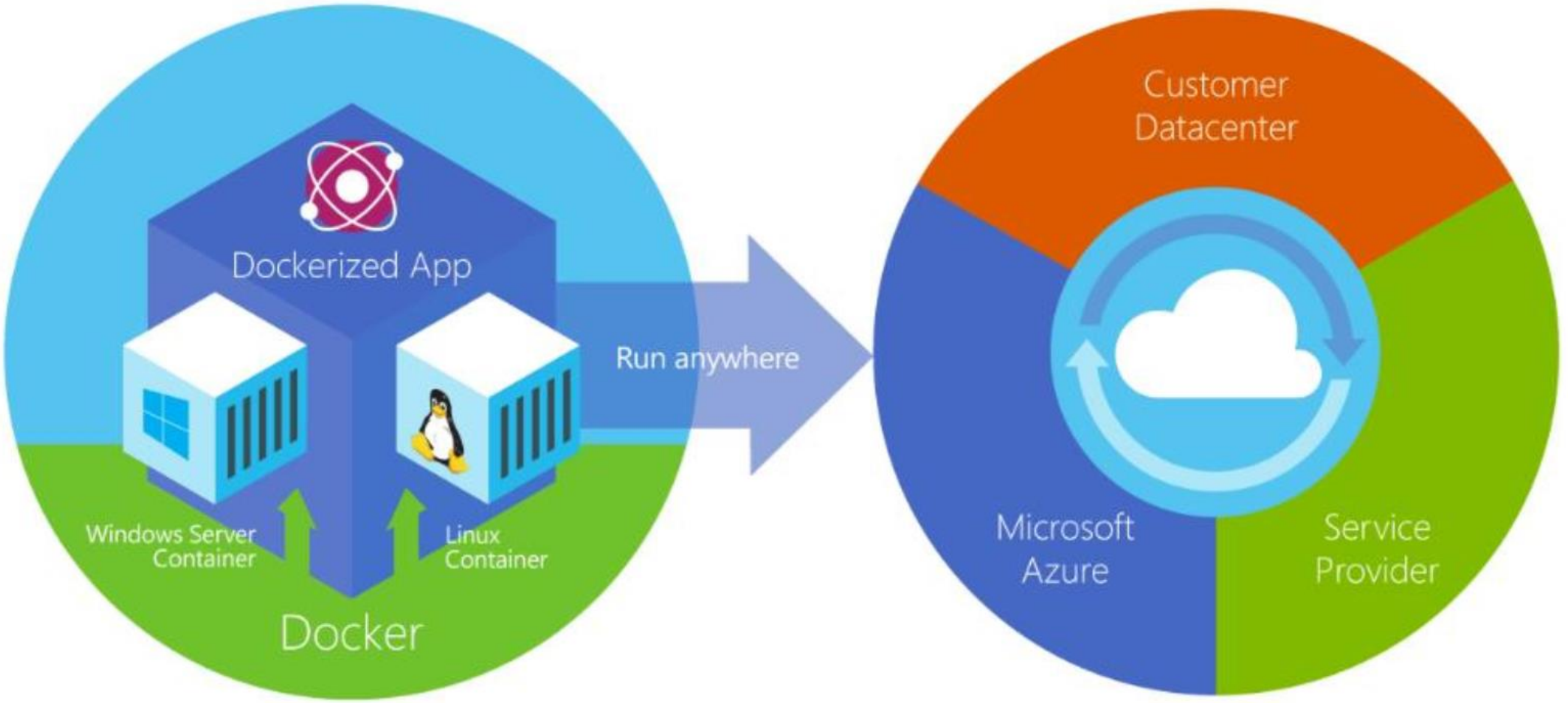


Figure 2-2. Docker deploys containers at all layers of the hybrid cloud.

- Розробники можуть використовувати середовища для розробки в Windows, Linux або macOS.
- На комп'ютері розробника розробник запускає хост Docker, де розгортаються образи Docker, включаючи програму та її залежності.
- Розробники, які працюють на Linux або macOS, використовують хост Docker на базі Linux, і вони можуть створювати образи лише для контейнерів Linux.
- Розробники, які працюють у Windows, можуть створювати образи для Linux або контейнерів Windows.

- Для розміщення контейнерів у середовищах розробки та надання додаткових інструментів розробника, Docker постачає **Docker Community Edition** (CE) для Windows або macOS.
- Ці продукти встановлюють необхідну віртуальну машину (хост Docker) для розміщення контейнерів.
- Docker також надає **Docker Enterprise Edition** (EE), який призначений для корпоративної розробки і використовується IT-командами, які створюють, постачають та запускають великі критичні для бізнесу додатки у виробництво.

Порівняння контейнерів Docker з віртуальними машинами

- На Рис. 2-3 показано порівняння між віртуальними машинами та контейнерами Docker.
- Віртуальні машини включають додаток, необхідні бібліотеки або двійкові файли та **повноцінну гостьову операційну систему**. Повна віртуалізація вимагає більше ресурсів, ніж контейнеризація.
- Контейнери включають програму та всі її залежності. Однак вони **діляться ядром ОС з іншими контейнерами**, виконуючись як окремі процеси в просторі користувача в хост-операційній системі.

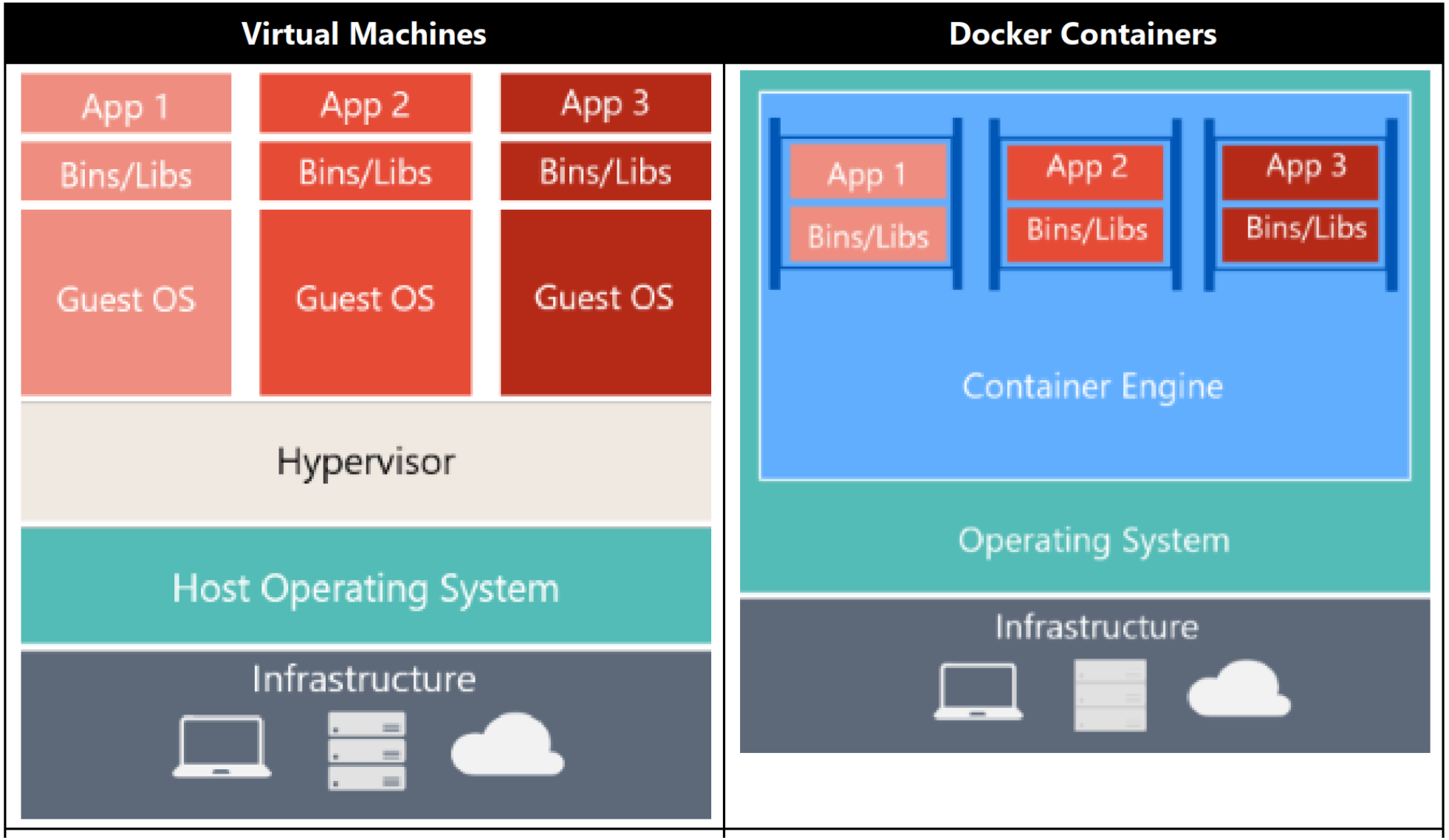


Рис. 2-3 Порівняння традиційних віртуальних машин з контейнерами Docker

- **Для віртуальних машин** на хост-сервері є три базових шари, знизу вгору: інфраструктура, операційна система хосту та гіпервізор, а на додачу кожна віртуальна машина має власну ОС та всі необхідні бібліотеки.
- **Для Docker** хост-сервер має лише інфраструктуру та ОС, а на додачу - механізм контейнерів, який тримає контейнер ізольованим, але спільно використовує базові служби ОС.
- Оскільки контейнери вимагають набагато менше ресурсів (наприклад, їм не потрібна повна ОС), їх легко розгорнути і швидко розпочати. Це дозволяє мати більш високу щільність, а це означає, що це дозволяє запускати більше сервісів на одному апаратному блоці, тим самим зменшуючи витрати.
- Як побічний ефект від роботи на одному ядрі ви отримуєте менше ізоляції, ніж віртуальні машини.

- Основна мета образу (image) полягає в тому, що він робить середовище (залежності) однаковим для різних розгортань. Це означає, що ви можете налагодити його на своїй машині, а потім **розгорнути на іншій машині** з тим самим гарантованим середовищем.
- Образ контейнера (container image) - це спосіб упакувати програму чи послугу та розгорнути її надійним та відтворюваним способом. Можна сказати, що Docker - це не лише технологія, а й філософія та процес.

- Використовуючи Docker, ви не почуєте, як розробники кажуть: **"Це працює на моїй машині, чому не працює у виробництві?"**
- Вони можуть просто сказати: **"Він працює на Docker"**, оскільки упаковану програму Docker можна виконати в будь-якому підтримуваному середовищі Docker, і вона працює так, як це було призначено для всіх цілей розгортання (таких як Dev, QA, staging та production).

Термінологія Докера

- У цьому розділі перелічені терміни та визначення, з якими слід ознайомитись, перш ніж глибше вивчати Docker.
- **Container image** (образ контейнера): Пакет (package) із усіма залежностями та інформацією, необхідними для створення контейнера. Образ включає всі *залежності* (наприклад, фреймворки), а також *конфігурацію розгортання та виконання*, які будуть використовуватися середовищем виконання контейнера. Зазвичай образ походить від декількох базових образів, які є шарами, складеними один над одним, щоб сформувати файлову систему контейнера. Образ незмінний після його створення.

- **Dockerfile**: Текстовий файл, що містить інструкції щодо створення образу Docker. Це як пакетний сценарій, у першому рядку для початку вказано базовий образ, а потім слідують інструкції щодо встановлення необхідних програм, копіювання файлів тощо, до отримання необхідного робочого середовища.
- **Build**: Дія побудови образу контейнера на основі інформації та контексту, наданих його Dockerfile-лом, а також додаткових файлів в папці, де побудовано образ. Ви можете створювати образ за допомогою команди Docker **docker build**.
- **Container**: Екземпляр образу Docker. Контейнер являє собою виконання однієї програми, процесу або сервісу. Він складається із вмісту образу Docker, середовища виконання та стандартного набору інструкцій. Під час масштабування сервісу ви створюєте кілька екземплярів контейнера з одного образу. Або пакетне завдання може створити кілька контейнерів з одного образу, передаючи різні параметри кожному екземпляру.

- **Volumes**: Файлова система для запису, яку контейнер може використовувати. Оскільки образи доступні лише для читання, але більшості програм потрібно писати у файлову систему, *томи* додають шар для запису, поверх зображення контейнера, тому програми мають доступ до файлової системи, в яку можна записувати. Програма не знає, що вона отримує доступ до шарованої файлової системи, це просто файлова система, як зазвичай. ***Томи*** живуть у хост-системі та керуються Docker.
- **Tag**: Позначка або мітка, яку ви можете застосувати до образу, щоб можна було ідентифікувати різні образи або версії одного образу (залежно від номера версії або цільового середовища).

- **Multi-stage Build**: Це функція, починаючи з Docker 17.05 або новішої версії, яка допомагає зменшити розмір кінцевих образів. У кількох реченнях, за допомогою *багатоступеневої збірки* ви можете використовувати, наприклад, великий базовий образ, що містить SDK, для компіляції та публікації програми, а потім за допомогою папки публікації з невеликим базовим образом, лише для виконання, для створення набагато меншого кінцевого образу.
- **Repository (repo):** Колекція відповідних образів Docker, позначена тегом, що вказує на версію образу. Деякі репозиторії містять кілька варіантів конкретного образу, наприклад образ, що містить SDK (важче), образ, що містить лише час виконання (легше) тощо. Ці варіанти можуть бути позначені тегами. Одне репо може містити варіанти платформи, такі як образ Linux та образ Windows.

- **Registry**: Сервіс, який забезпечує доступ до сховищ. Реєстром за замовчуванням для більшості загальнодоступних зображень є Docker Hub (належить Docker як організації). Реєстр зазвичай містить сховища від декількох команд. Компанії часто мають приватні реєстри для зберігання та управління створеними ними образами. Реєстр контейнерів Azure (Azure Container Registry) - ще один приклад.
- **Multi-arch image**: Для мультиархітектури це функція, яка спрощує вибір відповідного образу відповідно до платформи, на якій працює Docker. Наприклад, коли файл Docker запитує базовий образ **FROM mcr.microsoft.com/dotnet/core/sdk:3.1** з реєстру, він фактично отримує **3.1-sdk-nanoserver-1909**, **3.1-sdk-nanoserver-1809** або **3.1-sdk -buster-slim**, залежно від операційної системи та версії, де працює Docker.

- **Docker Hub**: Публічний реєстр для завантаження образів та роботи з ними. Docker Hub забезпечує розміщення образів Docker, публічні або приватні реєстри, тригери побудови та веб-хуки, а також інтеграцію з GitHub та Bitbucket.
- **Azure Container Registry**: Публічний ресурс для роботи із образами Docker та їх компонентами в Azure. Це забезпечує реєстр, близький до ваших розгортань в Azure і надає вам контроль над доступом, що дає можливість використовувати ваші групи та дозволи Azure Active Directory.
- **Docker Trusted Registry** (**DTR**): Служба реєстру Docker (від Docker), яку можна встановити локально, щоб вона була в центрі обробки даних та мережі організації. Це зручно для приватних образів, якими слід керувати на підприємстві. Довірений реєстр Docker входить до складу продукту Docker Datacenter.

- **Docker Community Edition (CE):** Засоби розробки для Windows та macOS для побудови, запуску та тестування контейнерів локально. Docker CE для Windows забезпечує середовища розробки для Linux і Windows контейнерів. Хост Linux Docker в Windows заснований на віртуальній машині Hyper-V. Хост для Windows Containers безпосередньо базується на Windows. Docker CE для Windows та Mac замінює Docker Toolbox, що базується на Oracle VirtualBox.
- **Docker Enterprise Edition (EE):** Корпоративна версія інструментів Docker для розробки Linux та Windows.

- **Compose**: Інструмент командного рядка та формат файлу YAML з метаданими для визначення та запуску багатоконтейнерних програм. Ви визначаєте одну програму на основі декількох образів з одним або кількома файлами .yml, які можуть замінювати значення залежно від середовища. Після створення визначень ви можете розгорнути цілу багатоконтейнерну програму за допомогою однієї команди (**dockercompose up**), яка створює контейнер для кожного зображення на хості Docker.
- **Cluster**: Колекція хостів Docker виставляється так, ніби це єдиний віртуальний хост Docker, так що програма може масштабуватися до декількох екземплярів служб, розподілених між кількома хостами в кластері. Кластери Docker можна створити за допомогою **Kubernetes**, **Azure Service Fabric**, **Docker Swarm** та Mesosphere DC/OS.

- **Orchestrator**: Інструмент, який спрощує управління кластерами та хостами Docker. Оркестратори дозволяють управляти їхніми образами, контейнерами та хостами за допомогою CLI або графічного інтерфейсу. Ви можете керувати мережею контейнерів, конфігураціями, балансуванням навантаження, виявленням послуг (service discovery), високою доступністю (high availability), конфігурацією хоста Docker тощо. Оркестратор відповідає за запуск, розподіл, масштабування та зцілення (healing) робочих навантажень (workloads) у колекції вузлів. Зазвичай продукти оркестратора - це ті самі продукти, що забезпечують кластерну інфраструктуру, такі як **Kubernetes** та **Azure Service Fabric**, серед пропонованих на ринку.

Контейнери, образи та реєстри Docker

- Використовуючи Docker, розробник створює програму чи сервіс та упаковує їх та їх залежності в образ контейнера. Образ - це статичне представлення програми чи сервісу та її конфігурації та залежностей.
- Щоб запустити програму або сервіс, образ програми інстанціюється, щоб створити контейнер, який працюватиме на хості Docker. Контейнери спочатку перевіряються в середовищі розробки або на ПК.

- Розробники повинні зберігати образи в **реєстрі**, який виконує функцію **бібліотеки образів** і який необхідний при розгортанні у виробничих оркестраторах.
- Docker веде загальнодоступний реєстр через **Docker Hub**; інші постачальники надають реєстри для різних колекцій образів, включаючи **Azure Container Registry**. Крім того, підприємства можуть мати приватний реєстр локально для власних зображень Docker.
- На Рис. 2-4 показано, як образи та реєстри в Docker відносяться до інших компонентів. Він також відображає різноманітні пропозиції реєстру (registry offerings) від постачальників.

Basic taxonomy in Docker

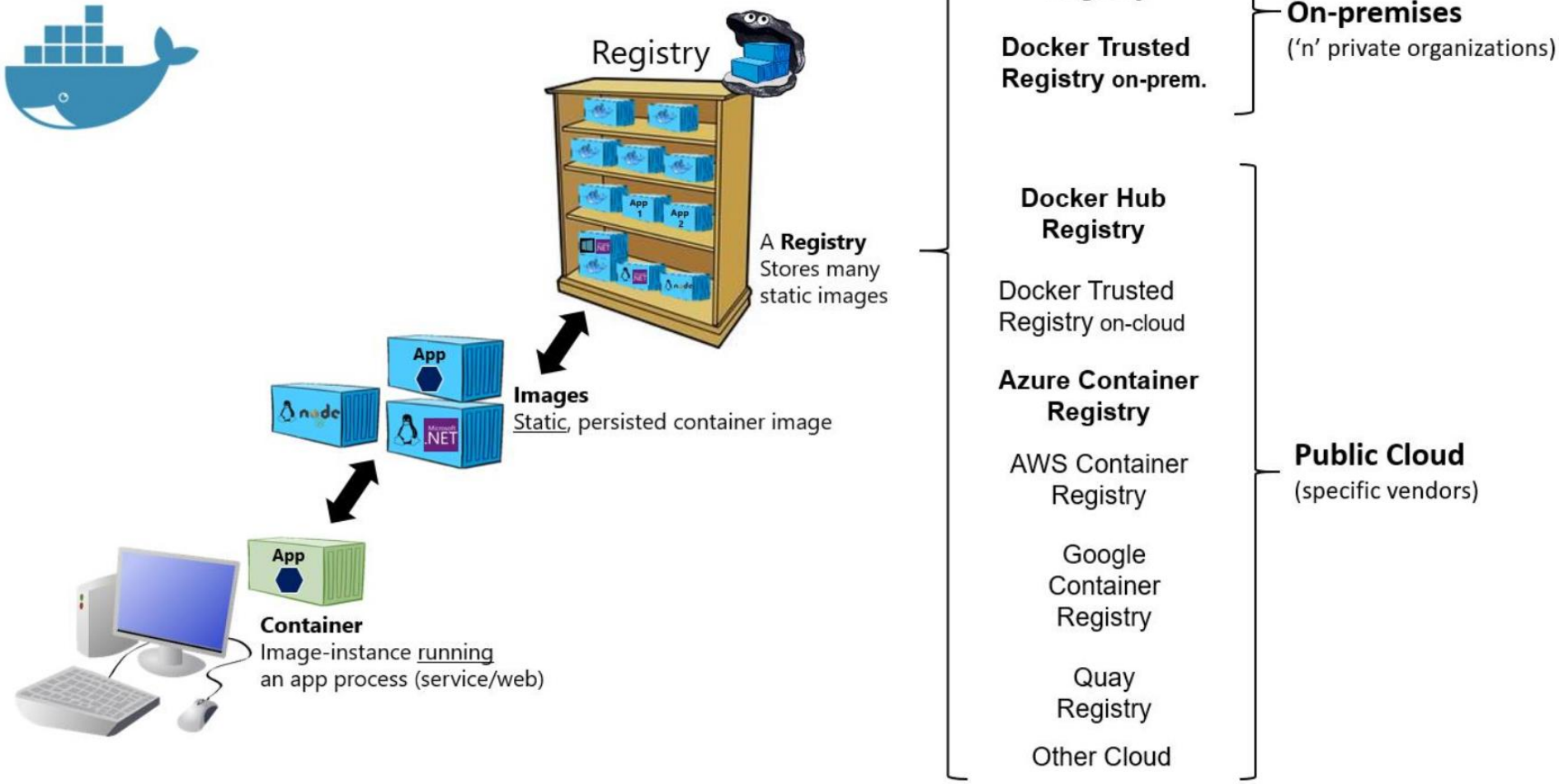


Рис. 2-4. Таксономія термінів та понять Докера

Вибір між .NET Core та .NET Framework для контейнерів Docker

- Існує два підтримуваних фреймворки для побудови серверних контейнерних програм Docker з .NET: .NET Framework та .NET Core. Вони мають багато компонентів платформи .NET, і ви можете ділитися кодом між ними.
- Однак між ними є принципові відмінності, і те, яку структуру ви використовуєте, буде залежати від того, що ви хочете досягти. Цей розділ містить вказівки щодо того, коли слід вибирати кожну структуру.

Загальні вказівки

У цьому розділі подано короткий опис того, коли вибрати .NET Core або .NET Framework.

Вам слід використовувати .NET Core з контейнерами Linux або Windows для вашої контейнерної серверної програми Docker, коли:

- Ви маєте потреби у різних платформах. Наприклад, ви хочете використовувати контейнери Linux і Windows.
- Архітектура вашого додатку базується на мікросервісах.
- Вам потрібно швидко запустити контейнери, і ви хочете мати невеликий розмір на контейнер, щоб досягти кращої щільності або більше контейнерів на апаратну одиницю, щоб зменшити ваші витрати.

- Коротше кажучи, коли ви створюєте нові контейнерні .NET-програми, вам слід розглянути .NET Core як вибір за замовчуванням. Він має багато переваг і найкраще відповідає філософії та стилю роботи контейнерів.
- Додатковою перевагою використання .NET Core є те, що ви можете запускати паралельні версії .NET для програм на одній машині. Ця перевага важливіша для серверів або віртуальних машин, які не використовують контейнери, оскільки контейнери ізолюють версії .NET, які потрібні додатку. (Поки вони сумісні з базовою ОС.)

Вам слід використовувати .NET Framework для вашої контейнерної серверної програми Docker, коли:

- Ваша програма в даний час використовує .NET Framework і сильно залежить від Windows.
- Вам потрібно використовувати API Windows, які не підтримуються .NET Core.
- Вам потрібно використовувати сторонні бібліотеки .NET або пакети NuGet, недоступні для .NET Core.

Використання .NET Framework на Docker може покращити ваш досвід розгортання, мінімізуючи проблеми із розгортанням. Цей сценарій "піднімання та зміщення" (“lift and shift”) є важливим для контейнеризації застарілих програм, які спочатку розроблялися за допомогою традиційних .NET Framework, таких як ASP.NET WebForms, веб-програми MVC або служби WCF (Windows Communication Foundation).

На яку ОС націлити контейнери .NET

- Враховуючи різноманітність операційних систем, що підтримуються Docker, та відмінності між .NET Framework та .NET Core, вам слід орієнтуватися на конкретну ОС та конкретні версії залежно від використовуваної платформи.
- Для Windows ви можете використовувати **Windows Server Core** або **Windows Nano Server**. Ці версії Windows забезпечують різні характеристики (IIS у Windows Server Core порівняно із самодостатнім веб-сервером, таким як Kestrel у Nano Server), які можуть знадобитися .NET Framework або .NET Core, відповідно.
- Для Linux доступні та підтримуються декілька дистрибутивів в офіційних образах .NET Docker (наприклад, Debian).

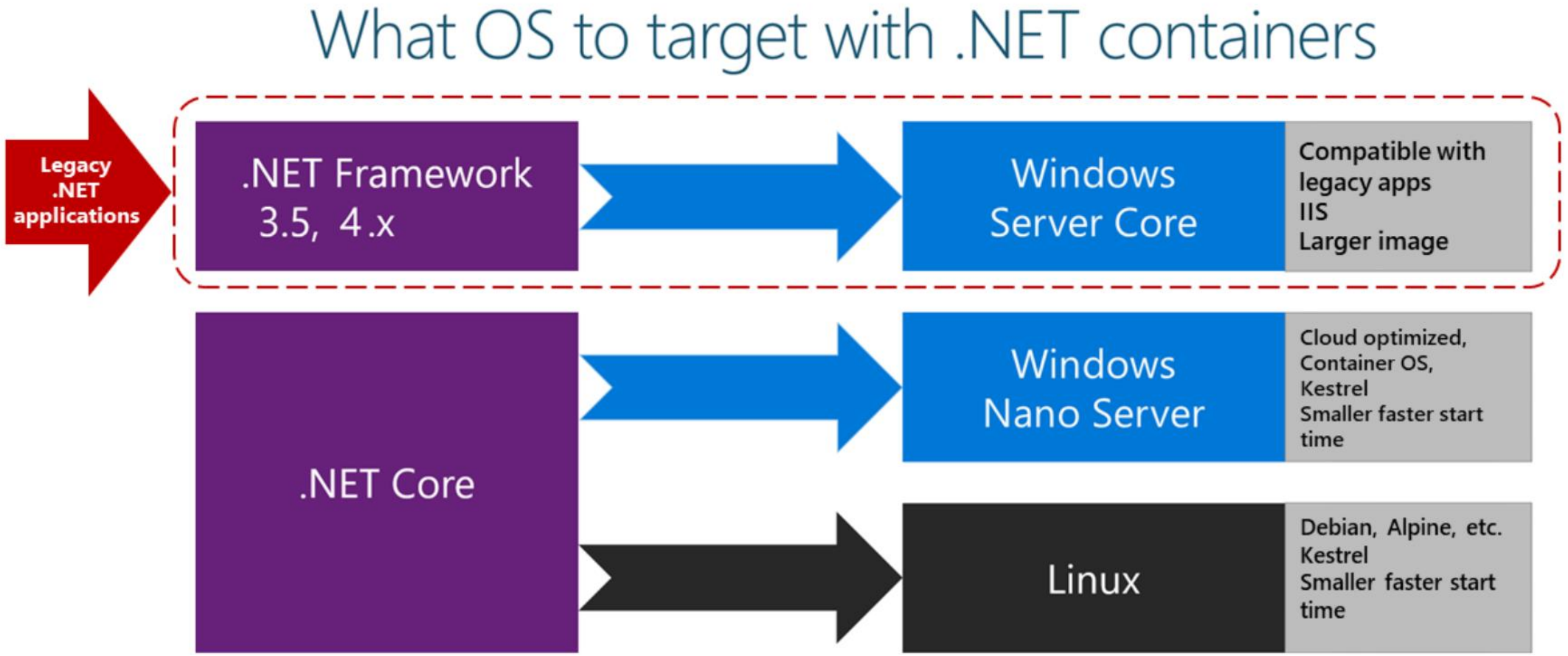


Рис. 3-1. Цільові операційні системи залежно від версій платформи .NET

Офіційні Docker образи .NET

- Офіційні Docker образи .NET - це образи Docker, створені та оптимізовані корпорацією Майкрософт.
- Вони є загальнодоступними у сховищах Microsoft на Docker Hub.
- Кожне сховище може містити декілька образів, залежно від версій .NET та залежно від ОС та версій (Linux Debian, Linux Alpine, Windows Nano Server, Windows Server Core тощо).
- Починаючи з .NET Core 2.1, усі образи .NET Core, включаючи ASP.NET Core, доступні в **Docker Hub** у сховищі зображень .NET Core: https://hub.docker.com/_/microsoft-dotnet-core/ .

Процес розробки додатків на основі Docker

Для розробки контейнеризованих .NET-програм можна використовувати:

- **Інтегроване середовище розробки** (IDE), орієнтоване на Visual Studio та Visual Studio tools for Docker
- **або CLI / Editor** (Docker CLI та Visual Studio Code)

Середовище розробки для Docker програм

Вибір інструменту розробки: IDE або редактор

- Незалежно від того, чи віддаєте ви перевагу повноцінному і потужному IDE або легкому та гнучкому редактору, Microsoft має інструменти, які ви можете використовувати для розробки програм Docker.
- **Visual Studio (for Windows).** Для розробки додатків .NET Core 3.1 на основі Docker за допомогою Visual Studio потрібна Visual Studio 2019 версії 16.4 або пізнішої. Visual Studio 2019 постачається з інструментами для вже вбудованого Docker. Інструменти для Docker дозволяють розробляти, запускати та перевіряти свої програми безпосередньо в цільовому середовищі Docker.

Ви можете натиснути клавішу F5, щоб запустити та налагодити програму (один контейнер або кілька контейнерів) безпосередньо на хості Docker, або натиснути CTRL + F5, щоб відредагувати та оновити програму без необхідності перебудовувати контейнер. Це найпотужніший варіант розробки програм на базі Docker.

- **Visual Studio Code and Docker CLI**. Якщо ви віддаєте перевагу легкому та кроссплатформенному редактору, який підтримує будь-яку мову розробки, ви можете використовувати Visual Studio Code та Docker CLI. Це крос-платформенний підхід до розробки для macOS, Linux та Windows. Крім того, Visual Studio Code підтримує розширення для Docker, такі як IntelliSense для Dockerfiles та гарячі клавіші для запуску команд Docker з редактора.

- Встановивши Docker Desktop Community Edition (CE), ви можете використовувати один Docker CLI для створення додатків як для Windows, так і для Linux.

Додаткові ресурси

- **Visual Studio**. Official site. https://visualstudio.microsoft.com/vs/
- **Visual Studio Code**. Official site. https://code.visualstudio.com/download
- **Docker Desktop for Windows Community Edition** (CE)

https://hub.docker.com/editions/community/docker-ce-desktop-windows

- **Docker Desktop for Mac Community Edition** (CE)

https://hub.docker.com/editions/community/docker-ce-desktop-mac

.NET мови та фреймворки для Docker контейнерів

- Як згадувалося в попередніх розділах цього посібника, ви можете використовувати .NET Framework, .NET Core або проект OpenSource Mono під час розробки Docker в контейнерах .NET.
- Ви можете розробляти на C#, F# або Visual Basic при націлюванні на контейнери Linux або Windows, залежно від того, яка платформа .NET використовується.

Процес розробки додатків Docker

Development workflow for Docker apps

- Життєвий цикл розробки додатків починається на вашому комп’ютері розробника, де ви програмуєте застосунок вибраною мовою та тестуєте його локально.
- За допомогою цього робочого процесу, незалежно від того, яку мову, фреймворк та платформу ви вибрали, ви завжди розробляєте та тестуєте контейнери Docker, але робите це локально.

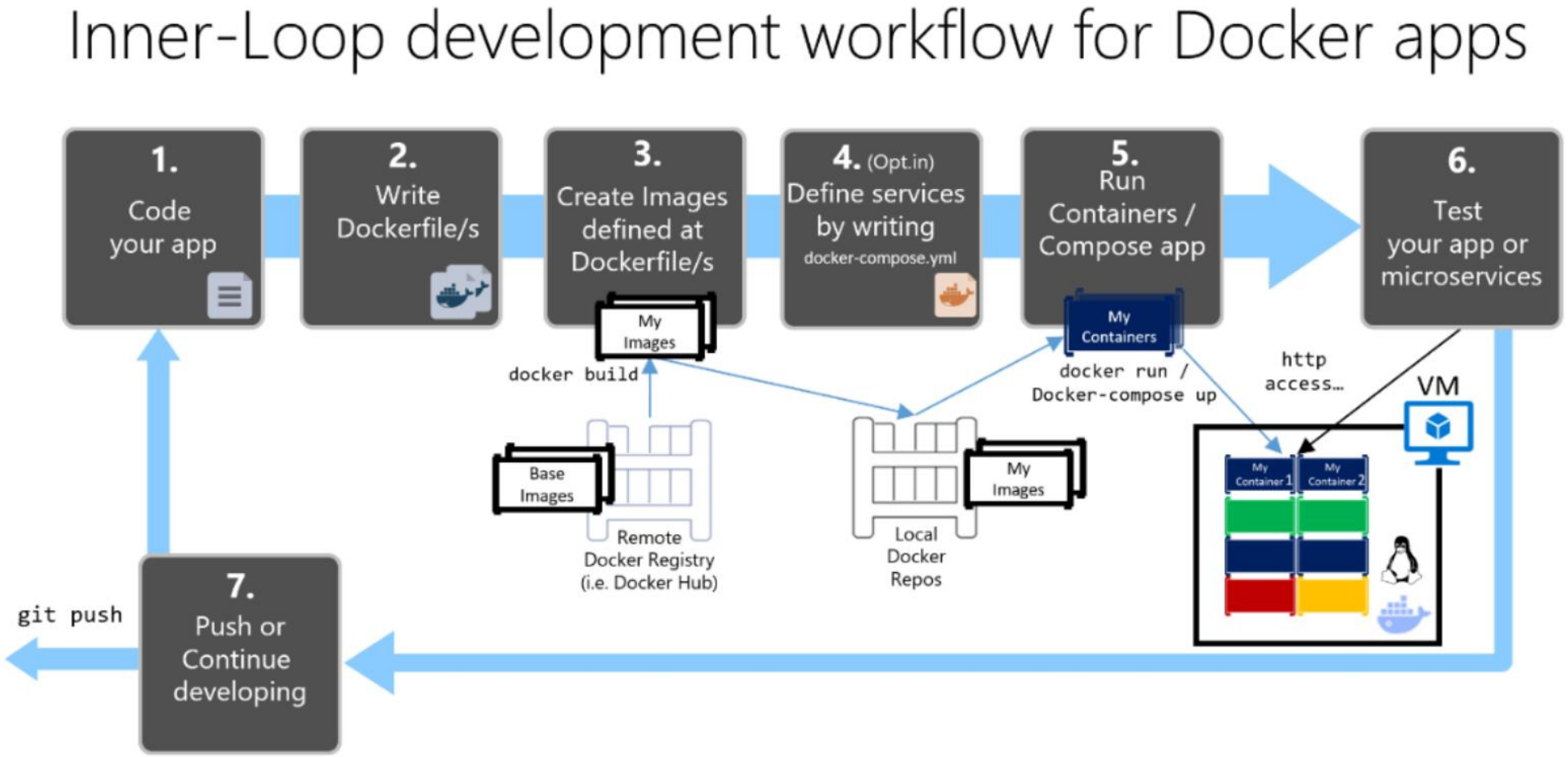
Кожен контейнер (екземпляр образу Docker) містить такі компоненти:

- Вибрану операційну систему, наприклад, дистрибутив Linux, Windows Nano Server або Windows Server Core.
- Файли, додані під час розробки, наприклад, вихідний код та двійкові файли додатків.
- Інформацію про конфігурацію, таку як налаштування середовища та залежності.

Робочий процес для розробки контейнерних програм Docker

- Цей розділ описує робочий процес розробки внутрішнього циклу для контейнерних програм Docker.
- Процес внутрішнього циклу означає, що він не враховує ширший робочий процес DevOps, який може включати розгортання production, а просто зосереджується на роботі над розробкою, виконаній на комп'ютері розробника.
- Початкові кроки з налаштування середовища не включені, оскільки ці кроки виконуються лише один раз.

- Додаток складається з ваших власних сервісів плюс додаткові бібліотеки (залежності). Нижче наведено основні кроки, які ви зазвичай робите під час створення програми Docker, як показано на Рис. 5-1.



- У цьому розділі весь цей процес детально описаний, і кожен основний крок пояснюється зосередженням уваги на середовищі Visual Studio.
- Коли ви використовуєте підхід до розробки **editor/CLI** (наприклад, Visual Studio Code плюс Docker CLI на macOS або Windows), вам потрібно знати кожен крок, як правило, більш детально, ніж якщо ви використовуєте Visual Studio.
- Для отримання додаткової інформації про роботу в середовищі CLI див. Електронну книгу Containerized Docker Application lifecycle with Microsoft Platforms and Tools.

- Коли ви використовуєте Visual Studio 2019, багато з цих кроків обробляються для вас, що суттєво покращує вашу продуктивність.
- Це особливо актуально, коли ви використовуєте Visual Studio 2019 і націлюєтеся на багатоконтейнерні програми. Наприклад, лише одним клацанням миші Visual Studio додає файл Dockerfile та docker-compose.yml до ваших проектів із конфігурацією для вашої програми.
- Коли ви запускаєте програму у Visual Studio, вона створює образ Docker і запускає багатоконтейнерну програму безпосередньо в Docker; це навіть дозволяє налагодити кілька контейнерів одночасно. Ці функції збільшать вашу швидкість розробки.
- Однак те, що Visual Studio робить ці кроки автоматичними, не означає, що вам не потрібно знати, що відбувається з Docker. Тому наступні вказівки деталізують кожен крок.

Крок 1. Почніть кодування та створіть початкову програму або базис сервісу

- Розробка програми Docker схожа на те, як ви розробляєте програму без Docker.
- Різниця полягає в тому, що під час розробки для Docker ви розгортаєте та тестуєте свою програму чи служби у контейнерах Docker, у вашому локальному середовищі (або у встановленій Linux VM Docker, або безпосередньо Windows, якщо ви використовуєте контейнери Windows).

Set up your local environment with Visual Studio

- Для початку переконайтеся, що у вас встановлено Docker Community Edition (CE) для Windows, як це пояснюється в наступних інструкціях: Get started with Docker CE for Windows

- Окрім того, вам потрібна Visual Studio 2019 версії 16.4 або пізнішої, із встановленим .NET Core cross-platform development, як показано на Рис. 5-2.
- Додаткові ресурси
  - Get started with Docker CE for Windows

  https://docs.docker.com/docker-for-windows/

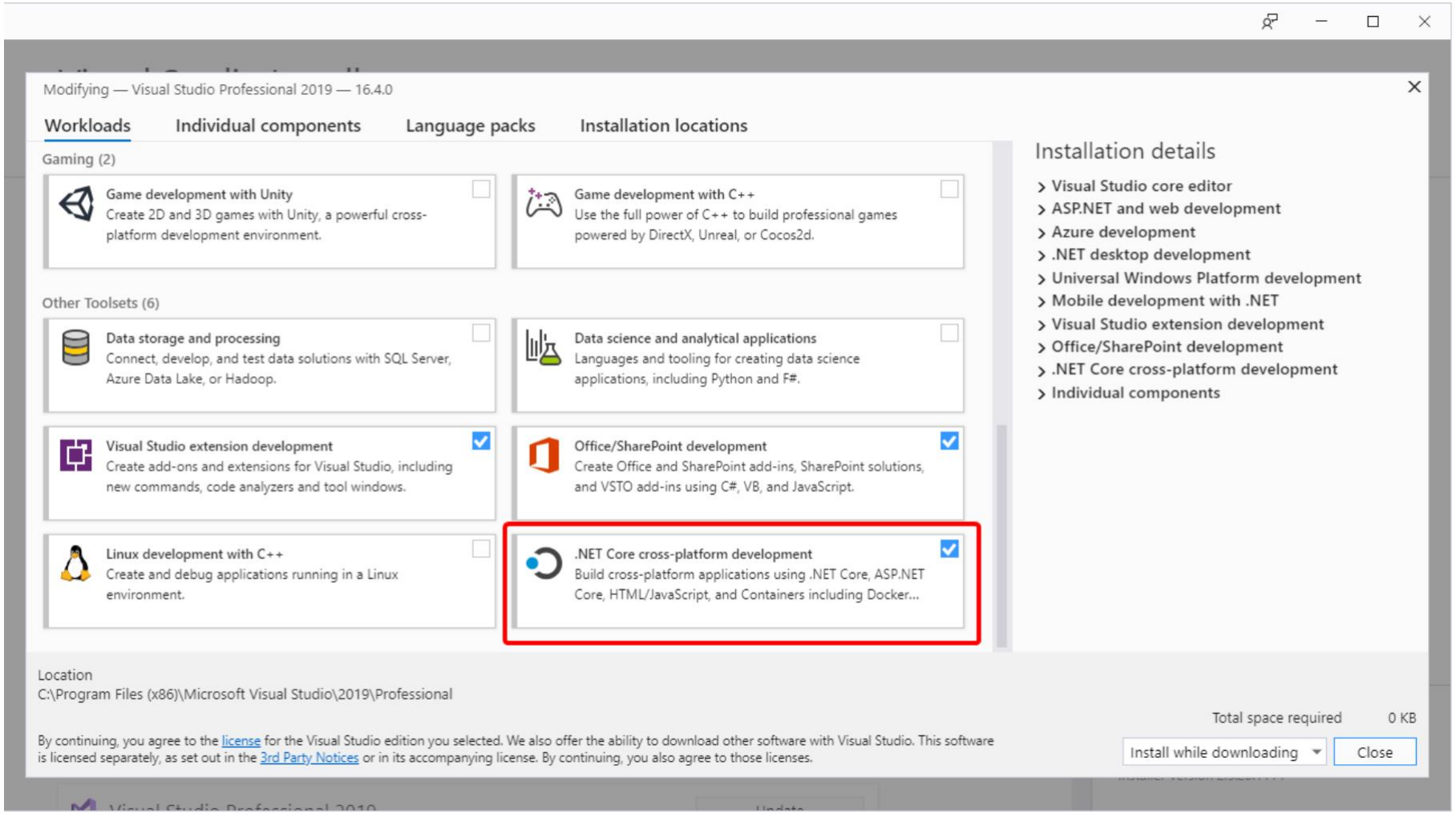


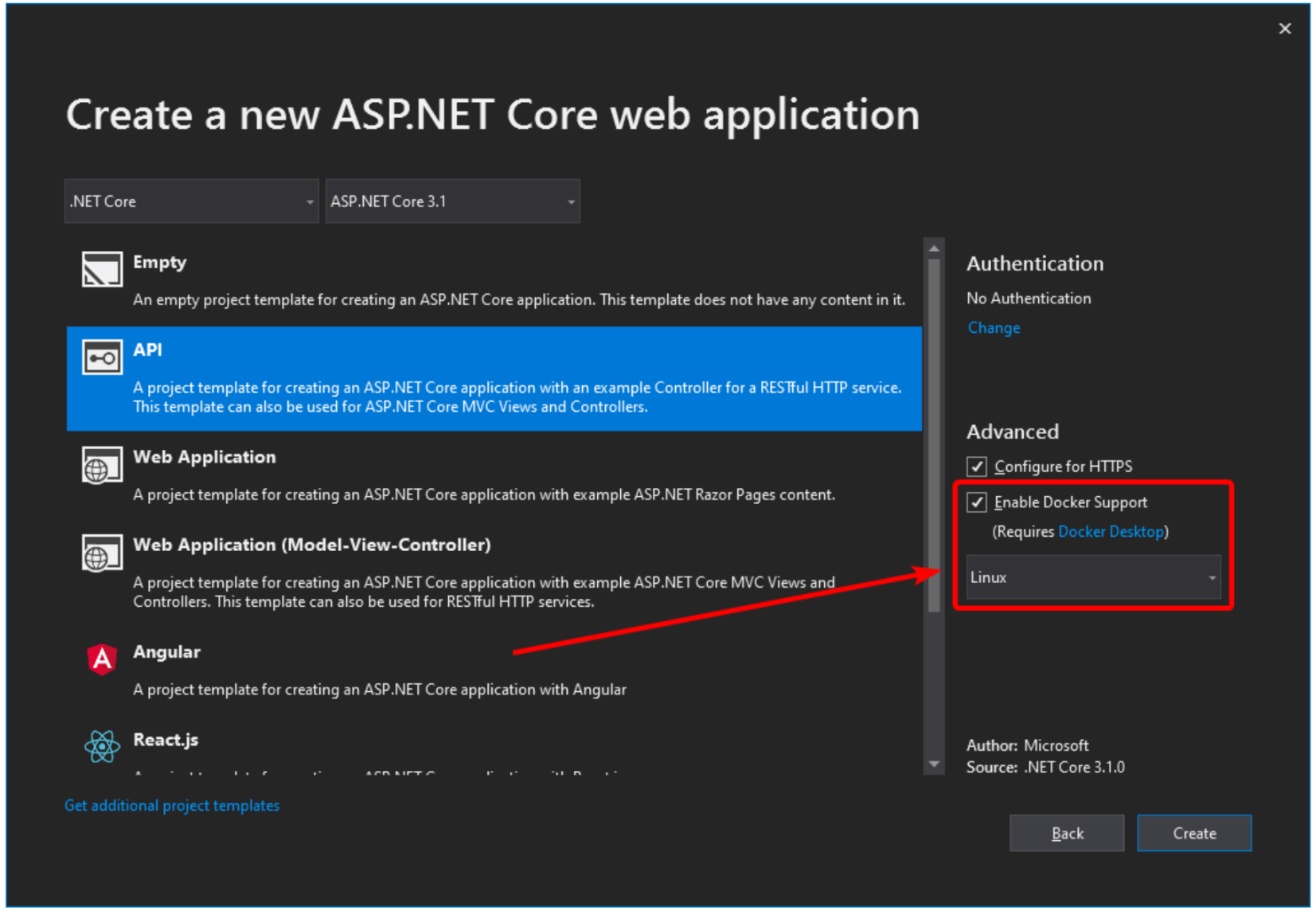
Рис 5-2. Selecting the .NET Core cross-platform development workload during Visual Studio 2019 setup

- Ви можете розпочати кодування своєї програми в простому .NET (зазвичай у .NET Core, якщо ви плануєте використовувати контейнери), навіть до того, як увімкнути Docker у вашому додатку та розгорнути та протестувати в Docker.
- Однак рекомендується починати працювати з Docker якомога раніше, оскільки це буде реальне середовище, і будь-які проблеми можна буде виявити якомога швидше.
- Це заохочується, оскільки Visual Studio дозволяє настільки легко працювати з Docker, що він майже відчуває себе прозорим - найкращий приклад - при налагодженні багатоконтейнерних програм з Visual Studio.

Крок 2. Створіть Dockerfile, пов’язаний із існуючим базовим образом .NET

- Вам потрібен Dockerfile для кожного власного образу, який ви хочете створити; вам також потрібен Dockerfile для кожного контейнера для розгортання, незалежно від того, чи ви розгортаєтеся автоматично з Visual Studio або вручну, використовуючи Docker CLI (команди запуску докера та docker-compose).
- Якщо ваша програма містить один користувацький сервіс, вам потрібен один Dockerfile. Якщо ваша програма містить кілька сервісів (як в архітектурі мікрсервісів), вам потрібен один Dockerfile для кожного сервісу.

- Dockerfile розміщується в кореневій папці вашої програми або служби. Він містить команди, які повідомляють Docker, як налаштувати та запустити програму чи службу в контейнері.
- Ви можете вручну створити Dockerfile у коді та додати його до свого проекту разом із залежностями .NET.
- З Visual Studio та його інструментами для Docker це завдання вимагає лише декількох клацань мишею. Коли ви створюєте новий проект у Visual Studio 2019, є опція Enable Docker Support, як показано на Рис. 5-3.



Enabling Docker Support when creating a new ASP.NET Core project in Visual Studio 2019

- Ви також можете ввімкнути підтримку Docker у існуючому проекті ASP.NET Core web app, клацнувши правою кнопкою миші проект у Solution Explorer та вибравши Add > Docker Support ...
- Ця дія додає Dockerfile до проекту з необхідною конфігурацією та доступна лише для проектів ASP.NET Core.
- Подібним чином Visual Studio може також додати файл docker-compose.yml для цілого рішення опцією Add > Container Orchestrator Support …. На кроці 4 ми детальніше розглянемо цей варіант.

Використання існуючого офіційного .NET Docker image

- Зазвичай ви створюєте власний образ для вашого контейнера **поверх базового образу**, який ви отримуєте з офіційного сховища, такого як реєстр Docker Hub.
- Саме це відбувається «за лаштунками», коли ви вмикаєте підтримку Docker у Visual Studio. Ваш Dockerfile використовуватиме наявний образ dotnet/core/aspnet.
- Раніше ми пояснювали, якими образами та репозиторіями Docker ви можете скористатися залежно від обраної вами структури та ОС. Наприклад, якщо ви хочете використовувати ASP.NET Core (Linux або Windows), використовуватиметься образ **mcr.microsoft.com/dotnet/core/aspnet:3.1**. Тому вам просто потрібно вказати, який базовий образ Docker ви будете використовувати для свого контейнера. Ви робите це, додаючи **FROM mcr.microsoft.com/dotnet/core/aspnet:3.1** у свій Dockerfile. Це буде автоматично виконуватися Visual Studio, але якщо ви хочете оновити версію, ви оновите це значення.

- Використання офіційного сховища образів .NET від Docker Hub з номером версії гарантує, що однакові мовні функції доступні на всіх машинах (включаючи розробку, тестування та виробництво).
- У наступному прикладі показано зразок Dockerfile для контейнера ASP.NET Core.

```
FROM mcr.microsoft.com/dotnet/core/aspnet:3.1
ARG source
WORKDIR /app
EXPOSE 80
COPY ${source:-obj/Docker/publish} .
ENTRYPOINT ["dotnet", " MySingleContainerWebApp.dll "]
```

Крок 3. Створіть власні образи Docker та вбудуйте в них свою програму чи службу

- Для кожної служби у вашій програмі вам потрібно створити відповідний образ. Якщо ваш додаток складається з однієї служби або веб-програми, вам потрібно лише один образ.
- Зверніть увагу, що образи Docker автоматично створюються для вас у Visual Studio. Наступні кроки потрібні лише для робочого процесу editor/CLI та пояснюються для ясності щодо того, що відбувається нижче.
- Щоб створити власний образ у вашому локальному середовищі за допомогою Docker CLI та вашого Dockerfile, ви можете скористатися командою **docker build**, як на Рис. 5-5.

```
PS C:\dev\netcore-webapi-microservice-docker> docker build -t cesardl/netcore-webapi-microservice-docker:first .
Sending build context to Docker daemon 1.148 MB
Step 1 : FROM microsoft/dotnet:latest
latest: Pulling from microsoft/dotnet
5c90d4a2d1a8: Downloading [================>                                  ] 18.34 MB/51.35 MB
ab30c63719b1: Downloading [==================================================> ] 18.48 MB/18.55 MB
c6072700a242: Downloading [====================>                              ] 18.34 MB/42.53 MB
121d7eef6c20: Waiting
eb57cf4f29ee: Waiting
b2c5ae2d325b: Waiting
```

- Це створить образ Docker з іменем **cesardl/netcore-webapi-microservicedocker: first**. У цьому випадку: first - це тег, що представляє конкретну версію. Ви можете повторити цей крок для кожного власного образу, який потрібно створити для складеної програми Docker.
- Коли додаток складається з декількох контейнерів (тобто це багатоконтейнерний додаток), ви також можете використовувати команду **docker-compose up --build**, щоб побудувати всі пов'язані образи за допомогою однієї команди, використовуючи метадані, пов'язані з файлами docker-compose.yml.

Створення образів Docker за допомогою Visual Studio

- Коли ви використовуєте Visual Studio для створення проекту з підтримкою Docker, ви явно не створюєте образи. Натомість образи створюється для вас, коли ви натискаєте клавішу F5 (або Ctrl-F5), щоб запустити докернізовану програму чи службу. Цей крок є автоматичним у Visual Studio, і ви цього не побачите, але важливо знати, що відбувається внизу.

Крок 4. Визначте свої служби у docker-compose.yml під час створення багатоконтейнерної програми Docker

- Файл docker-compose.yml дозволяє визначити набір пов'язаних служб, які будуть розгорнуті як складена програма з командами розгортання. Він також налаштовує свої відносини залежностей та конфігурацію часу виконання.
- Щоб використовувати файл docker-compose.yml, вам потрібно створити файл у вашій основній або кореневій папці рішення із вмістом, подібним до наведеного в наступному прикладі:

```yaml
version: '3.4'

services:

  webmvc:
    image: eshop/web
    environment:
      - CatalogUrl=http://catalog-api
      - OrderingUrl=http://ordering-api
    ports:
      - "80:80"
    depends_on:
      - catalog-api
      - ordering-api

  catalog-api:
    image: eshop/catalog-api
    environment:
      - ConnectionString=Server=sqldata;Port=1433;Database=CatalogDB;…
    ports:
      - "81:80"
    depends_on:
      - sqldata
```

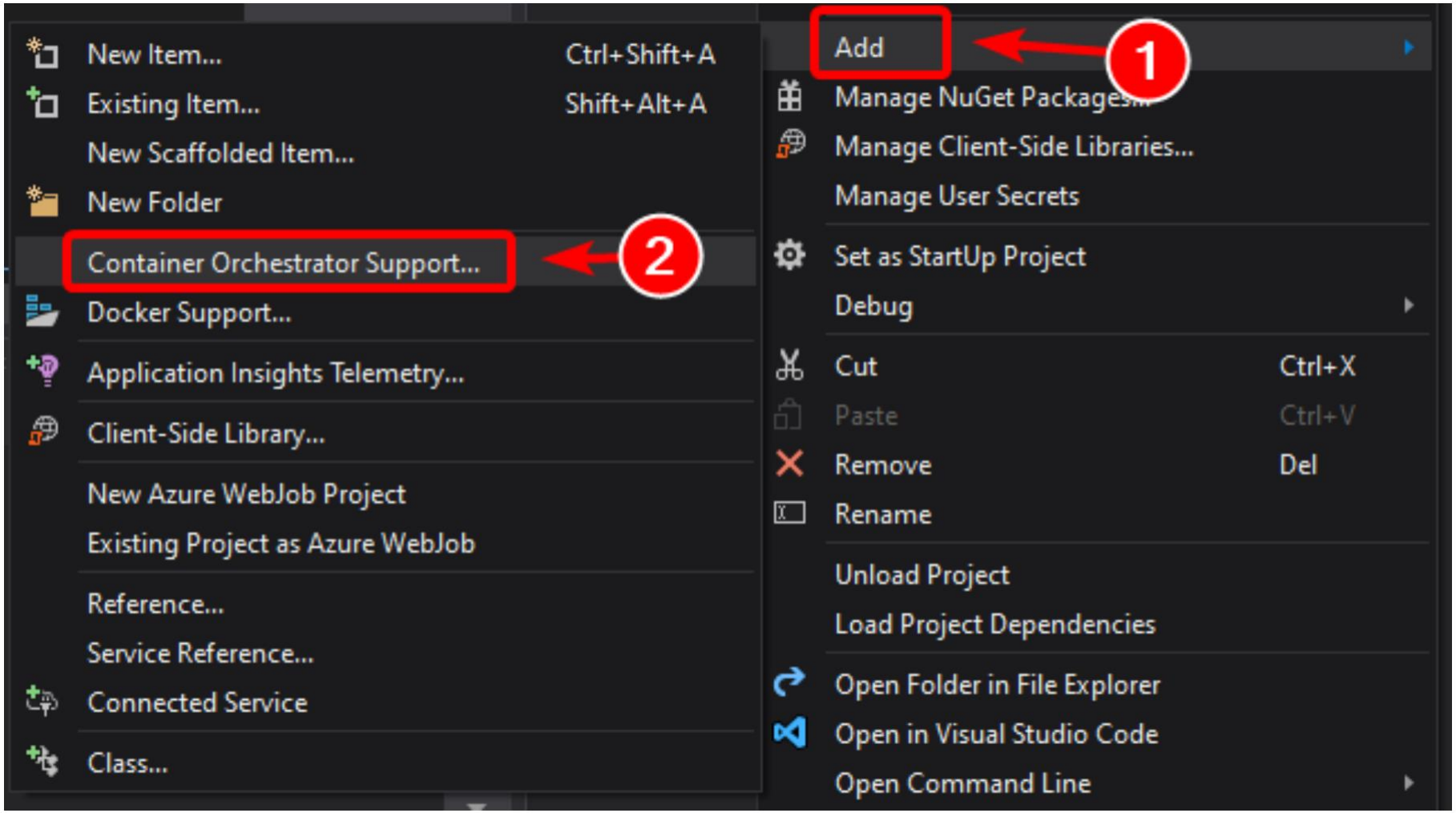
```
  ordering-api:
    image: eshop/ordering-api
    environment:
      - ConnectionString=Server=sqldata;Database=OrderingDb;…
    ports:
      - "82:80"
    extra_hosts:
      - "CESARDLBOOKVHD:10.0.75.1"
    depends_on:
      - sqldata

  sqldata:
    image: mssql-server-linux:latest
    environment:
      - SA_PASSWORD=Pass@word
      - ACCEPT_EULA=Y
    ports:
      - "5433:1433"
```

- Цей файл docker-compose.yml є спрощеною та об’єднаною версією. Він містить статичні дані конфігурації для кожного контейнера (наприклад, ім'я власного зображення), які завжди потрібні, та інформацію про конфігурацію, яка може залежати від середовища розгортання, наприклад рядка з'єднання.

- Приклад файлу docker-compose.yml визначає чотири служби: службу webmvc (*веб-додаток*), два *мікросервіси* (ordering-api та basket-api) та один контейнер *джерела даних*, sqldata, на основі SQL Server для Linux, що працює як контейнер. Кожна служба буде розгорнута як контейнер, тому для кожної потрібен образ Docker.
- Файл docker-compose.yml визначає не тільки те, які контейнери використовуються, але і те, як вони налаштовуються індивідуально. Наприклад, визначення контейнера webmvc у файлі .yml:
- Використовує заздалегідь побудований образ eshop/web:latest.
- Ініціалізує дві змінні середовища (CatalogUrl та OrderingUrl).
- Перенаправляє відкритий порт 80 на контейнері до зовнішнього порту 80 на хост-машині.
- Пов’язує веб-додаток із каталогом та службою замовлення з налаштуванням depend_on. Це змушує службу чекати, поки ці служби не будуть запущені.

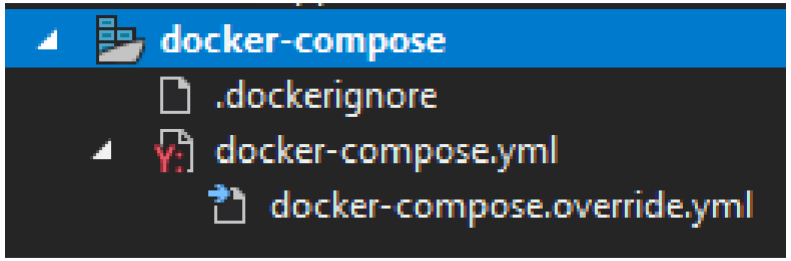
Робота з docker-compose.yml у Visual Studio 2019

- Окрім додавання Dockerfile до проекту, як ми вже згадували раніше, Visual Studio 2017 (починаючи з версії 15.8) може додавати рішення до підтримки оркестратора для Docker Compose.
- Коли ви вперше додаєте підтримку оркестратора контейнерів, як показано на Рис. 5-7, Visual Studio створює Dockerfile для проекту та створює новий проект (розділ service) у вашому рішенні з декількома глобальними файлами docker-compose * .yml , а потім додає проект до цих файлів. Потім можна відкрити файли docker-compose.yml та оновити їх додатковими функціями.
- На даний момент Visual Studio підтримує Docker Compose та оркестратори Kubernetes / Helm.



Adding Docker support in Visual Studio 2019 by right-clicking an ASP.NET Core project

- Після того, як ви додасте підтримку оркестратора до свого рішення у Visual Studio, ви також побачите новий вузол (у файлі проекту docker-compose.dcproj) у Провіднику рішень, який містить додані файли dockercompose.yml, як показано на Рис. 5-8.



Крок 5. Побудуйте та запустіть програму Docker

- Якщо у вашій програмі є лише один контейнер, ви можете запустити його, розгорнувши на своєму хості Docker (віртуальна машина або фізичний сервер).
- Однак, якщо ваша програма містить кілька служб, ви можете розгорнути її як складену програму, або за допомогою однієї команди CLI (docker-compose up), або за допомогою Visual Studio, яка використовуватиме цю команду за лаштунками.
- Давайте розглянемо різні варіанти.

Option A: Running a single-container application

Using Docker CLI

- Ви можете запустити контейнер Docker за допомогою команди docker run, як показано на Рис. 5-9:

```
docker run -t -d -p 80:5000 cesardl/netcore-webapi-microservice-docker:first
```

Вищенаведена команда створюватиме новий екземпляр контейнера із зазначеного образу при кожному його запуску. Ви можете використовувати параметр --name, щоб вказати ім'я контейнеру, а потім використати docker start {name} (або використовувати ідентифікатор контейнера або автоматичне ім'я) для запуску існуючого екземпляра контейнера.

```
PS C:\dev\netcore-webapi-microservice-docker> docker run -t -d -p 80:5000 cesardl/netcore-webapi-microservice-docker:first
d96975a683b0a9411595816f63be6c135801878b8a85181a4d86dc848ea4ca6f
```

- У цьому випадку команда прив'язує внутрішній порт 5000 контейнера до порту 80 хост-машини. Це означає, що хост прослуховує порт 80 і переадресовує порт 5000 на контейнер.
- Показаний хеш - це ідентифікатор контейнера, а також йому присвоюється випадкове для читання ім'я, якщо параметр --name не використовується.

Using Visual Studio

- Якщо ви не додали підтримку оркестратора контейнерів, ви також можете запустити одну програму-контейнер у Visual Studio, натиснувши **Ctrl-F5**, а також можна використовувати **F5** для налагодження програми в контейнері. Контейнер запускається локально, використовуючи docker run.

Option B: Running a multi-container application

- У більшості корпоративних сценаріїв програма Docker буде складатися з декількох служб, що означає, що вам потрібно запустити багатоконтейнерну програму, як показано на Рис. 5-10.

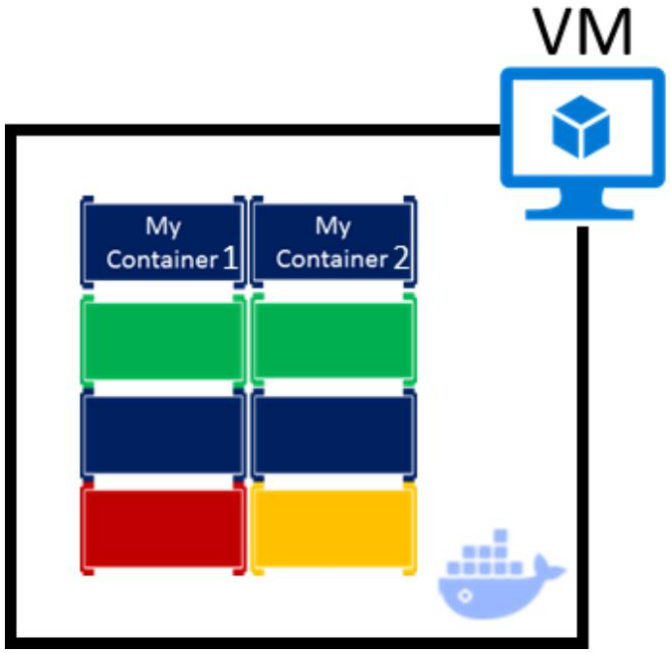


Figure 5-10. VM with Docker containers deployed

Using Docker CLI

- Щоб запустити багатоконтейнерну програму з Docker CLI, ви використовуєте команду *docker-compose up*. Ця команда використовує файл **docker-compose.yml**, який ви маєте на рівні рішення, для розгортання багатоконтейнерної програми. Рис. 5-11 показує результати під час запуску команди з вашого основного каталогу рішення, який містить файл docker-compose.yml.

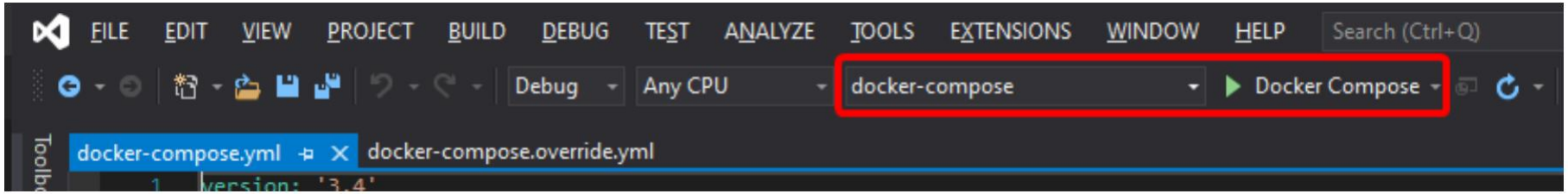
```
PS C:\Dev\WebApplication> docker-compose up
Recreating webapplication_webapplication_1
Attaching to webapplication_webapplication_1
webapplication_1   | Hosting environment: Production
webapplication_1   | Content root path: /app
webapplication_1   | Now listening on: http://*:80
webapplication_1   | Application started. Press Ctrl+C to shut down.
```

Figure 5-11. Example results when running the docker-compose up command

Після запуску команди docker-compose up програма та пов'язані з нею контейнери розгортаються у вашому хості Docker, як показано на Рис. 5-10.

Using Visual Studio

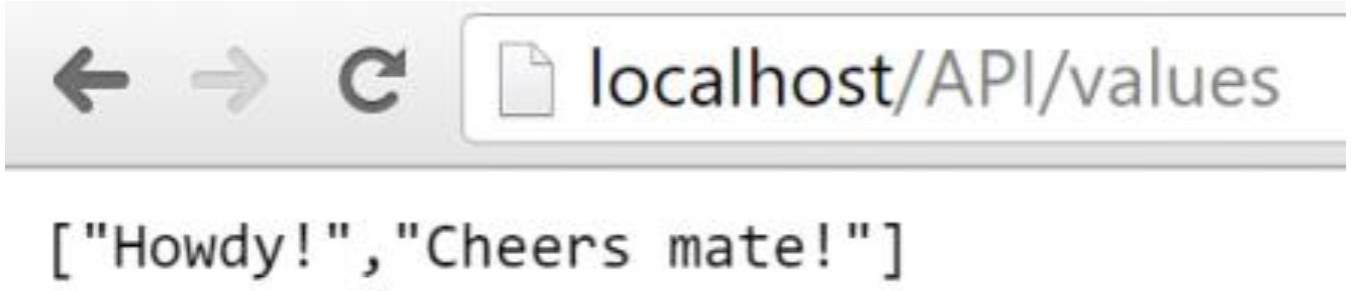
- Запуск програми з декількома контейнерами за допомогою Visual Studio 2019 не може бути простішим. Ви просто натискаєте Ctrl-F5 для запуску або F5 для налагодження.
- Visual Studio обробляє всі необхідні налаштування, тому ви можете створювати точки зупинки і налагоджувати, як зазвичай.
- Важливим моментом тут є те, що, як показано на малюнку 5-12, у Visual Studio 2019 існує додаткова команда Docker для дії клавіші F5.



- Цей параметр дозволяє запускати або налагоджувати багатоконтейнерну програму, запускаючи всі контейнери, визначені у файлах docker-compose.yml на рівні рішення.
- Можливість налагоджувати рішення з декількома контейнерами означає, що ви можете встановити кілька точок зупинки, кожна точка зупинки в іншому проекті (контейнері), і під час налагодження з Visual Studio ви зупинитесь на точках зупинки, визначених у різних проектах і запущених на різних контейнерах.

Крок 6. Тестуйте свою програму Docker, використовуючи локальний хост Docker

- Цей крок буде залежати від того, чим займається ваша програма. У простому веб-застосунку .NET Core, який розгортається як єдиний контейнер або служба, ви можете отримати доступ до служби, відкривши браузер на хості Docker і перейшовши на цей сайт, як показано на Рис. 5-13. (Якщо конфігурація в файлі Docker відображає контейнер до порту на хості, який відрізняється від 80, включіть порт хосту в URL-адресу.)



- Якщо localhost не вказує на IP-адресу хоста Docker (за замовчуванням, коли використовується Docker CE, це потрібно), для переходу до вашої служби використовуйте IP-адресу мережевої карти вашого ПК.
- Зверніть увагу, що ця URL-адреса у браузері використовує порт 80 для конкретного прикладу контейнера. Однак внутрішньо запити перенаправляються на порт 5000, оскільки саме так він був розгорнутий за допомогою команди docker run, як пояснювалося на попередньому кроці.
- Ви також можете протестувати додаток, використовуючи **curl** з терміналу, як показано на Рис. 5-14. В установці Docker в Windows, IP-адреса Docker Host за замовчуванням завжди є 10.0.75.1 на додаток до фактичної IP-адреси вашого комп'ютера.

```
PS C:\dev\netcore-webapi-microservice-docker> curl http://10.0.75.1/API/values


StatusCode        : 200
StatusDescription : OK
Content           : ["Howdy!","Cheers mate!"]
RawContent        : HTTP/1.1 200 OK
                    Transfer-Encoding: chunked
                    Content-Type: application/json; charset=utf-8
                    Date: Thu, 14 Jul 2016 19:48:18 GMT
                    Server: Kestrel

                    ["Howdy!","Cheers mate!"]
Forms             : {}
Headers           : {[Transfer-Encoding, chunked], [Content-Type, application/json;
                    charset=utf-8], [Date, Thu, 14 Jul 2016 19:48:18 GMT], [Server, Kestrel]}
Images            : {}
InputFields       : {}
Links             : {}
ParsedHtml        : mshtml.HTMLDocumentClass
RawContentLength  : 25
```

- *Figure 5-14. Example of testing your Docker application locally using curl*