Національний університет біоресурсів і природокористування України

Технології контейнеризації Python-додатків: використання Docker

Частина 1

Лектор: к.ф.-м.н., доцент Кириченко Віктор Вікторович

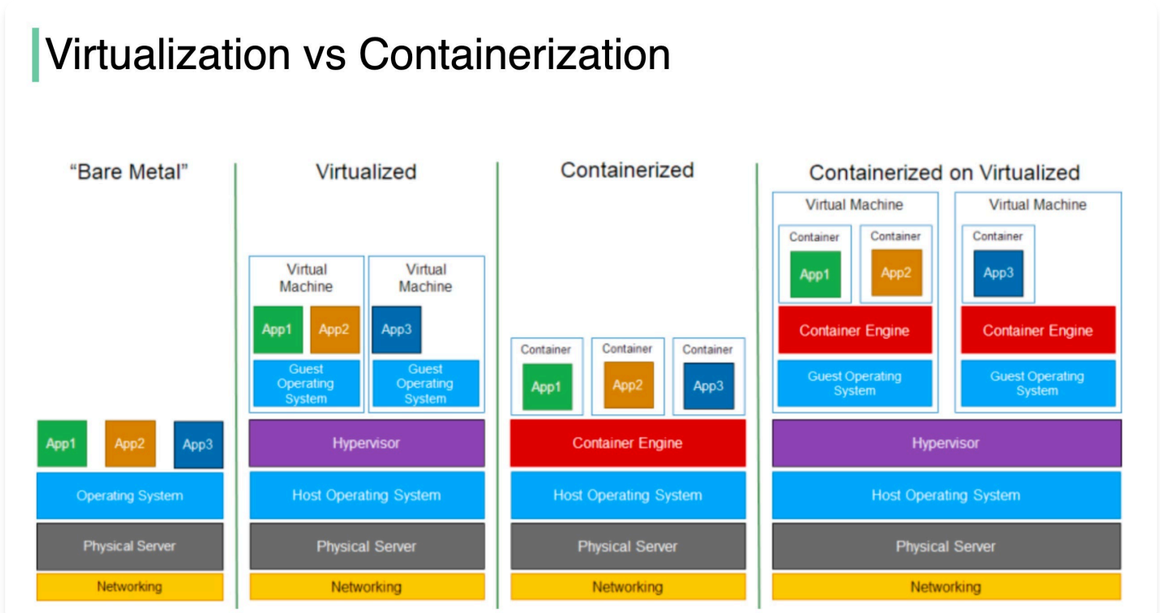
Вступ до контейнеризації та віртуалізації

Віртуалізація

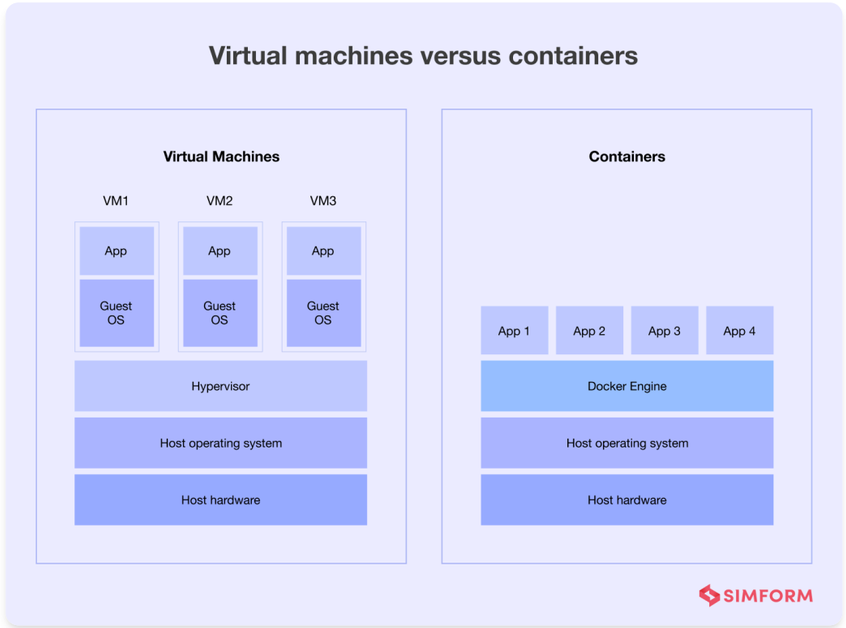
- Технологія створення віртуальних машин (ВМ)
- Кожна ВМ має власну ОС, ядро та бібліотеки
- Високий рівень ізоляції
- Великі ресурсні витрати

Контейнеризація

- Технологія ізоляції додатків на рівні ОС
- Контейнери використовують ядро хост-системи
- Ефективне використання ресурсів
- Швидке розгортання та запуск



Відмінності між віртуальними машинами та контейнерами



Віртуальні машини

- Повна емуляція апаратного забезпечення
- Кожна ВМ має власну ОС
- Повна ізоляція процесів
- Високі вимоги до ресурсів
- Повільний запуск (хвилини)

Контейнери

- Ізоляція на рівні процесів ОС
- Спільне використання ядра хост-системи
- Легка ізоляція процесів
- Ефективне використання ресурсів
- Швидкий запуск (секунди)

Переваги використання Docker у розробці Python-додатків

Вирішення проблеми несумісності

- Єдине середовище для розробки та продакшену
- Уникнення конфліктів залежностей

Відтворюваність

- Ідентичні умови на різних машинах
- Стабільність результатів тестування

Спрощення спільної роботи

- Швидке налаштування середовища для нових розробників
- Стандартизація процесів CI/CD

Ефективність

- Економія ресурсів порівняно з ВМ
- Швидке масштабування та оновлення

Основні компоненти Docker: образи

Що таке Docker-образ?

Незмінний шаблон, що містить всі необхідні компоненти для запуску додатку: код, бібліотеки, залежності, конфігураційні файли.

Структура образу

- Базовий образ (наприклад, Ubuntu, Python)
- Шари змін (read-only layers)
- Метадані (інструкції, порти, змінні середовища)

Створення образів

- Файл Dockerfile з інструкціями
- Команда `docker build`
- Тегування для версіонування

Отримання образів

- Завантаження з Docker Hub
- Команда `docker pull`
- Офіційні та неофіційні репозиторії

Керування образами

- `docker images` - перегляд
- `docker rmi` - видалення
- `docker save/load` - експорт/імпорт

Основні компоненти Docker: контейнери

Що таке Docker-контейнер?

Запущений екземпляр Docker-образу, що виконується в ізольованому середовищі з власними процесами, файловою системою та мережевими інтерфейсами.

▶ Життєвий цикл контейнера

- Створення (create)
- Запуск (start)
- Зупинка (stop)
- Видалення (rm)

Керування контейнерами

- `docker run` - запуск
- `docker ps` - перегляд активних
- `docker stop/start` - зупинка/запуск

Ізоляція та безпека

- Простори імен (namespaces)
- Групи контролю (cgroups)
- Обмеження ресурсів (CPU, RAM, I/O)

Взаємодія з контейнером

- `docker exec` - виконання команди
- `docker logs` - перегляд логів
- `docker attach` - підключення до контейнера

Основні компоненти Docker: Docker Hub

Що таке Docker Hub?

Хмарний реєстр сервіс, що дозволяє знаходити та обмінюватися Docker-образами з командою або спільнотою розробників.

Основні функції

- Зберігання та розповсюдження образів
- Автоматичні збірки з GitHub/Bitbucket
- Веб-хуки для інтеграції з CI/CD
- Організації та команди для керування доступом

Офіційні образи

- Перевірені та підтримувані Docker
- Python, Node.js, Nginx, MySQL та інші
- Безпечні та регулярно оновлювані
- Маркуються синім значком "DOCKER OFFICIAL IMAGE"

Пошук та використання

- `docker search` - пошук образів
- `docker pull [image]` - завантаження
- `docker push [image]` - публікація
- Веб-інтерфейс для перегляду та керування

Облікові записи

- Безкоштовний акаунт для публічних репозиторіїв
- Платні плани для приватних репозиторіїв
- Аутентифікація через `docker login`
- Можливість створення власних образів

Типові сценарії використання Docker у розробці та деплої Python-додатків

Локальна розробка

- Створення однакового середовища для всіх розробників
- Швидке налаштування залежностей (Python, бібліотеки)
- Ізоляція проєктів та їхніх залежностей

Тестування

- Створення ізольованого середовища для тестів
- Автоматизація тестування в CI/CD
- Паралельне тестування на різних версіях Python

Розгортання веб-сервісів

- Портативність між різними середовищами
- Просте масштабування з використанням Docker Swarm або Kubernetes
- Швидке відновлення після збоїв

Пакетна обробка даних

- Запуск аналітичних скриптів у ізольованому середовищі
- Використання специфічних версій бібліотек (NumPy, Pandas)
- Планування завдань через cron або інші засоби

Мікросервісна архітектура

- Розбиття великого додатку на невеликі сервіси
- Кожен сервіс у власному контейнері з власними залежностями
- Комунікація між сервісами через API
- Незалежне масштабування та оновлення окремих сервісів

Встановлення Docker та перевірка робочого середовища

Встановлення Docker

- Завантаження з офіційного сайту docker.com
- Вибір відповідної версії для ОС
- Docker Desktop для Windows/Mac
- Docker Engine для Linux

Перевірка встановлення

- `docker --version` - перевірка версії
- `docker info` - інформація про систему
- `docker run hello-world` - тестовий запуск

Налаштування середовища

- Запуск Docker Desktop/сервісу
- Налаштування ресурсів (CPU, RAM)
- Перевірка доступу до Docker Hub

Робота з командним рядком

- `docker ps` - перегляд активних контейнерів
- `docker images` - перегляд образів
- `docker system prune` - очищення

Поширені проблеми та їх вирішення

Проблема: Docker не запускається
Перевірка віртуалізації в BIOS, оновлення Docker

Проблема: Помилка доступу
Додавання користувача до групи docker

Проблема: Контейнер не запускається
Перевірка логів через `docker logs`

Проблема: Немає доступу до мережі
Перевірка мережевих налаштувань Docker

Створення першого контейнера з офіційного Python-образу

Завантаження Python-образу

- `docker pull python` - остання версія
- `docker pull python:3.10` - конкретна версія
- `docker pull python:slim` - мінімальний образ

Запуск контейнера

- `docker run python` - базовий запуск
- `docker run -it python` - інтерактивний режим
- `docker run --name my-python python` - з іменем

Основні параметри запуску

- `-d` - запуск у фоновому режимі
- `-p 8000:8000` - проброс портів
- `-v /host/path:/container/path` - монтування томів

Перевірка контейнера

- `docker ps` - активні контейнери
- `docker ps -a` - всі контейнери
- `docker inspect [container_id]` - детальна інформація

Приклад запуску Python-контейнера

```
# Завантаження останньої версії Python-образу
docker pull python

# Запуск контейнера в інтерактивному режимі
docker run -it --name my-python-container python

# Перевірка версії Python всередині контейнера
docker exec my-python-container python --version
```

Запуск простого Python-скрипту в контейнері

Створення Python-скрипту

```python
# hello.py
def main():
  print("Привіт, Docker-контейнер!")
  print("Це простий Python-скрипт")
if __name__ == "__main__":
  main()
```

- Збережіть скрипт у файл hello.py
- Перевірте його роботу локально

Структура проєкту

```
python-docker-demo/
  ├── hello.py
  └── Dockerfile
```

- Створіть папку для проєкту
- Розмістіть у ній скрипт hello.py
- Створіть Dockerfile у тій же папці

Створення Dockerfile

```dockerfile
FROM python:3.10
WORKDIR /app
COPY hello.py .
CMD ["python", "hello.py"]
```

- Вказуємо базовий образ Python
- Копіюємо скрипт в контейнер
- Вказуємо команду для запуску

Запуск контейнера

- `docker build -t python-demo .` - збірка образу
- `docker run python-demo` - запуск контейнера
- `docker run --name my-demo python-demo` - з іменем

Результат виконання

```
$ docker run python-demo
Привіт, Docker-контейнер!
Це простий Python-скрипт
```

Практична вправа: запуск Python-контейнера та взаємодія з терміналом

▶ Запуск контейнера в інтерактивному режимі

```
# Запуск контейнера з доступом до терміналу
docker run -it --name python-interactive python bash

# Перевірка версії Python всередині контейнера
python --version

# Запуск інтерактивного інтерпретатора Python
python
```

- Параметр `-it` забезпечує інтерактивний доступ
- Команда `bash` запускає оболонку всередині контейнера

<> Виконання Python-коду в контейнері

```
# Після запуску інтерактивного інтерпретатора
print("Привіт з Docker-контейнера!")
Привіт з Docker-контейнера!
# Простий розрахунок
result = 2 ** 10
print(result)
1024
```

- Використовуйте стандартні Python-команди
- Для виходу з інтерпретатора: `exit()` або `Ctrl+D`

Робота з файловою системою

- `ls -la` - перегляд вмісту директорії
- `mkdir test_dir` - створення директорії
- `cd test_dir` - перехід до директорії
- `pwd` - поточна директорія

```
# Створення та виконання Python-файлу
echo 'print("Файл створено в контейнері")' > test.py
python test.py
Файл створено в контейнері
```

Керування контейнером

- `Ctrl+P, Ctrl+Q` - від'єднання без зупинки
- `docker attach python-interactive` - підключення знову
- `docker stop python-interactive` - зупинка контейнера
- `docker start python-interactive` - запуск знову

Важливі моменти

Збереження змін

Всі зміни в контейнері будуть втрачені після його видалення, якщо не створити новий образ

Вихід з контейнера

Для виходу з контейнера без зупинки використовуйте Ctrl+P, Ctrl+Q

Перевірка статусу

Використовуйте `docker ps` для перегляду активних контейнерів

Видалення контейнера

Для видалення зупиненого контейнера: `docker rm python-interactive`

Практична вправа: запуск Python-контейнера та взаємодія з терміналом (частина 2)

Монтування томів (Volumes)

```
# Створення директорії на хості
mkdir ~/python-app

# Створення файлу в цій директорії
echo 'print("Цей файл знаходиться на хості")' > ~/python-app/app.py

# Запуск контейнера з монтуванням томів
docker run -it -v ~/python-app:/app python bash
```

- Параметр `-v` монтує директорію хоста в контейнер
- Формат: `-v [шлях_на_хості]:[шлях_в_контейнері]`

Робота зі спільними файлами

```
# Перехід до директорії /app в контейнері
cd /app

# Перегляд вмісту директорії
ls -la

# Запуск Python-файлу
python app.py
Цей файл знаходиться на хості
```

- Файли, створені на хості, доступні в контейнері
- Зміни, зроблені в контейнері, зберігаються на хості

Редагування коду в реальному часі

- Вийдіть з контейнера (Ctrl+P, Ctrl+Q)
- Відредагуйте файл app.py на хості
- Підключіться до контейнера знову
- Запустіть оновлений файл в контейнері

```
# Редагування файлу на хості
echo 'print("Файл оновлено!")' > ~/python-app/app.py

# Підключення до контейнера
docker attach [container_id]

# Запуск оновленого файлу
python /app/app.py
Файл оновлено!
```

Збереження змін у новий образ

- Вийдіть з контейнера
- Зупиніть контейнер: `docker stop [container_id]`
- Створіть новий образ: `docker commit [container_id] my-python-app`
- Перевірте новий образ: `docker images`

Підсумок вправи

Вивчені навички

- Запуск інтерактивного Python-контейнера
- Виконання коду Python в контейнері
- Робота з файловою системою контейнера
- Монтування томів для обміну файлами

Наступні кроки

- Створення складніших Dockerfile
- Використання Docker Compose
- Розгортання веб-додатків
- Інтеграція з CI/CD процесами

Висновки

Ключові переваги Docker для Python-додатків

- Контейнеризація забезпечує однакове середовище виконання на всіх етапах розробки
- Вирішує проблему "в мене працює" через ізоляцію залежностей
- Забезпечує портативність додатків між різними системами

Архітектурні переваги

- Ефективне використання ресурсів порівняно з ВМ
- Швидке масштабування та оновлення сервісів
- Можливість створення мікросервісної архітектури

Практичне застосування

- Легке розгортання веб-додатків
- Спрощення процесів CI/CD
- Швидка інтеграція з хмарними платформами

Командна робота

- Стандартизація середовищ для всіх розробників
- Спрощення онбордингу нових членів команди
- Покращення спільної роботи над проєктами

Перспективи розвитку

- Інтеграція з оркестраторами (Kubernetes, Docker Swarm)
- Розвиток серверних serverless-архітектур на основі контейнерів
- Поглиблення інтеграції з хмарними сервісами

Docker для Python-додатків — це потужний інструмент, що забезпечує ізоляцію, портативність та ефективність розробки та розгортання програмного забезпечення.