Національний університет біоресурсів і
природокористування України

Технології контейнеризації Python-додатків: використання Docker

Частина 2

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

Вступ до Dockerfile

Що таке Dockerfile?

- Текстовий документ з інструкціями для автоматизованої збірки Docker-образу
- Містить усі команди, необхідні для створення образу
- Дозволяє автоматизувати процес створення контейнерів

Призначення Dockerfile

- Створення відтворюваного середовища
- Автоматизація процесу розгортання
- Версіонування інфраструктури
- Спрощення процесу CI/CD

Переваги використання Dockerfile

- Контроль версій інфраструктури
- Автоматизація процесів
- Стандартизація середовищ

Структура Dockerfile та основні інструкції

FROM

Визначає базовий образ для створення нового образу

```
FROM python:3.9-slim
```

- Обов'язкова інструкція
- Перша інструкція у Dockerfile

COPY

Копіює файли з хост-системи в контейнер

```
COPY . /app
```

- Синтаксис: COPY <src> <dest>
- Підтримує символи підстановки

RUN

Виконує команди під час збірки образу

```
RUN pip install -r requirements.txt
```

- Форми: shell або exec
- Створює новий шар образу

Приклад базового Dockerfile

```
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
CMD ["python", "app.py"]
```

Додаткові інструкції Dockerfile

CMD

Визначає команду за замовчуванням для запуску контейнера

```
CMD ["python", "app.py"]
```

- Форми: exec, shell або параметр ENTRYPOINT
- Може бути перевизначена при запуску

EXPOSE

Інформує Docker про порти, які слухає контейнер

```
EXPOSE 8000
```

- Не відкриває порт автоматично
- Для відкриття порту використовуйте -p

ENV

Встановлює змінні середовища в контейнері

```
ENV APP_ENV=production
```

- Формати: KEY=VALUE або KEY VALUE
- Зберігається у фінальному образі

Приклад Dockerfile з додатковими інструкціями

```
FROM python:3.9-slim
WORKDIR /app
ENV PYTHONUNBUFFERED=1
ENV APP_ENV=production
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
EXPOSE 8000
CMD ["python", "app.py"]
```

Написання Dockerfile для Python-додатків

Кращі практики

- Використовуйте офіційні образи Python з Docker Hub
- Встановлюйте робочу директорію (WORKDIR)
- Використовуйте .dockerignore для виключення зайвих файлів
- Копіюйте залежності окремо для кешування шарів

Поширені помилки

- Запуск від root-користувача (використовуйте USER)
- Ігнорування оптимізації розміру образу
- Використання застарілих версій Python

Приклад Dockerfile для Flask-додатку

```
FROM python:3.9-slim
WORKDIR /app
ENV PYTHONUNBUFFERED=1
# Встановлюємо залежності
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
# Копіюємо код додатку
COPY . .
# Створюємо не-root користувача
RUN useradd --create-home --shell /bin/bash appuser
USER appuser
EXPOSE 5000
CMD ["python", "app.py"]
```

Збірка Docker-образу з Dockerfile

Команда docker build

```
docker build [OPTIONS] PATH | URL | -
```

- PATH - шлях до директорії з Dockerfile
- За замовчуванням шукає файл з назвою "Dockerfile"
- Створює образ згідно інструкцій у Dockerfile

Ключові опції

- `-t, --tag` - ім'я та тег образу
- `-f, --file` - шлях до Dockerfile
- `--no-cache` - без використання кешу
- `--build-arg` - встановлення змінних збірки

Приклади використання

```
# Базова збірка
docker build -t my-python-app .
```

```
# З тегом та іншим файлом
docker build -t my-app:v1.0 -f Dockerfile.prod .
```

```
# З аргументами збірки
docker build --build-arg PYTHON_VERSION=3.9 -t my-app .
```

Процес збірки

1. Docker читає Dockerfile
2. Виконує інструкції послідовно
3. Створює проміжні шари для кожної інструкції
4. Кешує шари для прискорення наступних збірок
5. Створює фінальний образ після виконання всіх інструкцій

Запуск контейнера з власного образу

Команда docker run

```
docker run [OPTIONS] IMAGE[:TAG] [COMMAND] [ARG...]
```

- IMAGE[:TAG] - ім'я та тег образу для запуску
- Створює та запускає новий контейнер
- Якщо образ відсутній локально, завантажує його з реєстру

Ключові опції

- `-d, --detach` - запуск у фоновому режимі
- `-p, --publish` - публікація портів
- `-v, --volume` - монтування томів
- `--name` - присвоєння імені контейнеру
- `-e, --env` - встановлення змінних середовища

Приклади використання

```
# Базовий запуск
docker run my-python-app
```

```
# З портом та у фоновому режимі
docker run -d -p 8000:8000 my-python-app
```

```
# З монтуванням томів та змінними середовища
docker run -d -p 8000:8000 -v ./data:/app/data --name myapp -e
ENV=prod my-python-app
```

Управління контейнерами

Перегляд запущених

```
docker ps
```

Перегляд усіх

```
docker ps -a
```

Зупинка контейнера

```
docker stop [ID]
```

Видалення контейнера

```
docker rm [ID]
```

Робота з Docker Hub: публікація та завантаження образів

Публікація образів

1. Вхід до Docker Hub

```
docker login
```

2. Тегування образу

```
docker tag my-app username/my-app:v1.0
```

3. Завантаження образу

```
docker push username/my-app:v1.0
```

Завантаження образів

1. Пошук образів

```
docker search python
```

2. Завантаження образу

```
docker pull username/my-app:v1.0
```

3. Перегляд локальних образів

```
docker images
```

Важливі моменти

- Ім'я образу має включати ім'я користувача Docker Hub
- Теги допомагають керувати версіями образів
- Публічні образи доступні всім користувачам

Корисні опції

- `--all-tags` - завантажити всі теги образу
- `--platform` - вказати платформу образу
- `docker logout` - вихід з облікового запису

Приклад повного циклу роботи з Docker Hub

```
# Створення та публікація
docker build -t my-python-app .
docker tag my-python-app user/my-python-app:latest
docker login
docker push user/my-python-app:latest
```

```
# Завантаження та використання
docker pull user/my-python-app:latest
docker run -d -p 8000:8000 user/my-python-app:latest
docker ps
```

Типові помилки при контейнеризації Python-додатків

Проблеми із залежностями

```
ModuleNotFoundError: No module named 'requests'
```

- Не встановлені залежності
- Використовуйте requirements.txt та правильний COPY

Проблеми з правами доступу

```
PermissionError: [Errno 13] Permission denied
```

- Запуск від root-користувача
- Створюйте не-root користувача за допомогою USER

Проблеми з кешуванням

```
Using cache → e2b3a4c5d6f7
```

- Застарілий кеш при зміні залежностей
- Використовуйте --no-cache або змінюйте порядок інструкцій

Проблеми з портами

```
Connection refused: localhost:8000
```

- Неправильне налаштування мережі
- Використовуйте EXPOSE та -p для пробросу портів

Рекомендації для уникнення помилок

Структура проекту

- ✓ Використовуйте .dockerignore
- ✓ Розділяйте залежності та код

Оптимізація

- ✓ Використовуйте multi-stage builds
- ✓ Обирайте легкі базові образи

Безпека

- ✓ Не зберігайте секрети в образі
- ✓ Регулярно оновлюйте базові образи

Практична вправа: створення Dockerfile, збірка образу та запуск контейнера

1 Крок 1: Створення Python-додатку

app.py

```python
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello():
    return "Hello from Docker!"

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=5000)
```

requirements.txt

```
Flask==2.0.1
```

2 Крок 2: Створення Dockerfile

```dockerfile
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
EXPOSE 5000
CMD ["python", "app.py"]
```

3 Крок 3: Збірка образу та запуск контейнера

Збірка Docker-образу

```
docker build -t my-python-app .
```

Запуск контейнера

```
docker run -d -p 5000:5000 --name my-app my-python-app
```

Перевірка роботи

Перевірка контейнера

```
docker ps
```

Перевірка логів

```
docker logs my-app
```

Тестування додатку

```
curl http://localhost:5000
```

Висновки

Стандартизація середовищ

Docker забезпечує однакове середовище виконання на всіх етапах розробки та розгортання Python-додатків.

Ізоляція залежностей

Контейнери дозволяють ізолювати Python-додатки та їхні залежності, уникаючи конфліктів між проєктами.

Автоматизація процесів

Dockerfile дозволяє автоматизувати процес створення образів та розгортання додатків.

Переносимість

Docker-контейнери можна легко переносити між різними системами та хмарними платформами без змін у коді.

Ключовий висновок

Використання Docker для контейнеризації Python-додатків значно підвищує ефективність розробки, надійність та переносимість програмного забезпечення.