

Lokalne testy z docker-compose a praca programisty

Michał "Butla" Bultrowicz

Primary Software Wizard

<https://witchsoft.com>

<https://bultrowicz.com>

Slajdy + notatki: <https://github.com/butla/presentations>

Co to kontener?

- proces lub grupa procesów oddzielony od systemu hosta
- ma system plików niezależny od hosta
- trochę jak VM, ale nie do końca

Docker i Docker Compose

- Docker: implementacja kontenerów.
- Jest więcej implementacji, np. Podman.
- Docker Compose: pozwala opisać i zarządzać grupą powiązanych kontenerów.

Przykładowa aplikacja - repozytorium notatek

Python REST API + PostgreSQL

Endpointy API:

- `POST /notes/` - stwórz notkę
- `GET /notes/{id}/` - pobierz notkę po ID
- `GET /notes/` - pobierz wszystkie notki

Uruchamianie aplikacji

```
# docker-compose.yml
---
version: '3'
services:
  api:
    build:
      context: .
    image: sample_backend
    ports:
      - "8080:8080"
    links:
      - database
    environment:
      - POSTGRES_HOST=database
  database:
    image: postgres:15.2
    ports:
      - "5432:5432"
    environment:
      - POSTGRES_PASSWORD=postgres
    volumes:
      - db-data:/var/lib/postgresql/data
volumes:
  db-data:
```

```
# Dockerfile
FROM python:3.10-alpine

EXPOSE 8080

WORKDIR /app

COPY requirements.txt /app/
RUN pip install -r requirements.txt

COPY sample_backend /app/sample_backend

CMD ["uvicorn", "--host", "0.0.0.0",
      "--port", "8080",
      "sample_backend.main:app"]
```

Uruchamianie aplikacji

```
# Makefile
.EXPORT_ALL_VARIABLES:
SHELL:=/bin/bash

setup_development:
    poetry install

run: _start_compose _db_migration

_start_compose:
    docker-compose up -d

_db_migration:
    poetry run alembic upgrade head
```

- modyfikacja migracji, aby czekały na pojawienie się bazy
- ``git clone <repo> && cd <repo> && make setup_development``
`run``

Co już mamy?

- Żywa aplikacja, lokalnie.
- Możliwość eksperymentowania.
- Bardzo prosta instrukcja uruchamiania.

Pora na testy!



Testy zintegrowane (integrated)

- używanie wewnętrznych interfejsów kodu (jak testy jednostkowe)
- wykorzystanie zewnętrznych systemów (np. PostgreSQL z kontenera)
- nie potrzeba mocków

```
def test_create_a_note():  
    # arrange  
    note_contents = f"I'm a note, wee! {uuid.uuid4()}" # some randomness  
    notes_repo = NotesRepository(...) # object that connects to the DB  
  
    # act  
    id = notes_repo.create(note_contents) # calls out to Postgres at localhost:5432  
  
    # assert  
    with db_session() as session:  
        query = select(Note).where(Note.id == id)  
        saved_object = session.execute(query).scalar()  
    assert saved_object.contents == note_contents
```

Testy zewnętrzne (external; aka. functional/e2e)

- używanie tylko zewnętrznych interfejsów (np. HTTP)
- konfiguracja maksymalnie zbliżona do produkcyjnej
- trudniejsze w debugu, trzeba zaglądać do logów kontenerów

```
import uuid, httpx

def test_store_and_retrieve_note(app_url): # a more elaborate scenario
    note_contents = f"a note {uuid.uuid4()}" # some randomness

    create_result = httpx.post(f"{app_url}/notes/", json={"contents": note_contents}) # calling the app in Docker
    assert create_result.json()["contents"] == note_contents
    note_id = create_result.json()["id"]

    get_by_id_result = httpx.get(f"{app_url}/notes/{note_id}/")
    assert get_by_id_result.json()["contents"] == note_contents

    get_all_result = httpx.get(f"{app_url}/notes/")
    # finding the new note among all notes
    assert next(note for note in get_all_result.json() if note["id"] == note_id)
```

Brakujący kod z poprzedniego slajdu

```
import uuid, httpx, pytest, tenacity

# Session scope ensures we wait only once per test suite run.
@pytest.fixture(scope="session")
def app_url():
    app_address = "http://localhost:8080"
    _wait_for_http_url(app_address)
    return app_address

# Call the app until it returns correctly or times out.
@tenacity.retry(stop=tenacity.stop_after_delay(10), wait=tenacity.wait_fixed(0.2), reraise=True)
def _wait_for_http_url(url: str):
    result = httpx.get(url)
    if result.status_code != 200:
        raise ValueError("App returned the wrong status code")

def test_store_and_retrieve_note(app_url):
    ...
```

Uruchamianie testów

```
SOURCES:=sample_backend tests

check: static_checks test

# SUBCOMMANDS =====
test:
    @echo === Running tests... ===
    @poetry run pytest tests

static_checks: _check_isort _check_format _check_linter _check_types


_check_isort:
    @echo === Checking import sorting... ===
    @poetry run isort -c $(SOURCES)

_check_format:
    @echo === Checking code formatting... ===
    @poetry run black --check $(SOURCES)

...
```

```
$ make check
```

Testy zintegrowane i zewnętrzne - co dostajemy?

- dowód, że aplikacja się uruchamia
- większa pewność, że działa
- mniej pracy niż ustawianie mocków
- swoboda w korzystaniu z pełnej mocy narzędzi
- wolniejsze niż jednostkowe, nadal szybkie
-  brak pełnej izolacji między testami

Brak izolacji - odrobina chaosu

- reset danych między każdym testem może być niepraktyczny
 - za wolny dla SQL
 - dla Redisa znośny (ale uniemożliwia równoległe testy)
- niektóre testy (np. daj wszystkie notki) muszą brać na to poprawkę
 - kolekcje "wszystkich elementów" będą zmienne
 - dobrze tworzyć odizolowane grupy danych
- losowe problemy będą irytować

Orobina chaosu - większy realizm

- wersja produkcyjna nie czyści co chwilę bazy
- wyłapywanie błędów przed produkcją:
 - baza rośnie
 - "flaky" testy wskazują wyścigi
- naprawianie losowych problemów zwiększy jakość
- ewentualnie można czyścić lokalną bazę: `docker-compose down -v``

Organizacja testów

```
project_root/
├── ...
├── tests
│   ├── external
│   │   └── ...
│   ├── integrated
│   │   └── ...
│   └── unit
│       └── ...
```

- jawny podział
- ilość wysokopoziomowych testów trzeba kontrolować
- łatwość puszczania szybszych podgrup

Działa dla złożonych aplikacji

- sprawdzone w bojach (trzy różne firmy)
- integracja innych systemów (Kafka, Redis, Rabbit, itp.)
- AWS lokalnie - Localstack
- udawanie innych REST API - Mountebank

Continuous Integration / Delivery

Organizacja CI

- CI usuwa `docker-compose.override.yml` - wykluczenie źle zbudowanego obrazu
- CI używa tego samego Makefile'a
- komendy składowe `make check` rozdzielone między równoległe zadania
- po testach oznacz zbudowany obraz Dockera, wypchnij do repo, używaj w deploymentach

CI self-hosted runners: wolne party

```
# docker-compose.yml
---
version: '3'
services:
  api:
    ports:
      - "${API_PORT:-8080}:8080"
    ...
  database:
    ports:
      - "${POSTGRES_PORT:-5432}:5432"
    ...
```

```
# get_free_port.py
# https://unix.stackexchange.com/a/132524/128610

#!/usr/bin/env python3
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(('', 0))
addr = s.getsockname()
print(addr[1])
s.close()
```

```
$ export \
  API_PORT=$(./get_free_port.py) \
  POSTGRES_PORT=$(./get_free_port.py)
$ make run check
```

Więcej sztuczek ✨

Przetładowywanie kodu w kontenerze

```
# docker-compose.override.yml
---
version: '3'

services:

  api:
    volumes:
      # local folder mounted into the container
      - ./sample_backend:/app/sample_backend/
```

```
# Makefile
run_reloading: run
  fd --exclude .git --no-ignore '\.py$$' sample_backend \
  | entr -c make _start_compose

test_reloading:
  fd --exclude .git --no-ignore '\.py$$' \
  | entr -c make test
```

- nie trzeba przebudowywać Dockera
- aplikacja w Dockerze odświeża się przy zmianie pliku
- entr
- fd

Obiecany, ale pominięty materiał

- debug kodu w kontenerze
 - CLI
 - IDE, e.g. Intellij/Pycharm
- zmiany kodu produkcyjnego dla ułatwienia testowania
 - każdy sleep w apce konfigurowalny, małe wartości dla testów
 - to w porządku dodawać funkcjonalność dla ułatwienia testów
 - ...inne...

Fin

