# Developer workflow with local tests using Docker Compose

Michał "Butla" Bultrowicz

Primary Software Wizard

https://witchsoft.com

https://bultrowicz.com

Slides + notes: https://github.com/butla/presentations

# What’s a container?

- a process or a group of processes separated from the host’s system
- has a file system independent from the host’s one
- a bit like a Virtual Machine, but not really

# Docker and Docker Compose

- Docker: an implementation of containers.
- There are more implementations, e.g. Podman.
- Docker Compose: describes and manages a group of related containers.

# Sample application - a notes repository

Python REST API + PostgreSQL

Endpoints API:

- `` `POST /notes/` `` - create a note
- `` `GET /notes/{id}/` `` - get the note by ID
- `` `GET /notes/` `` - get all notes

# Docker Compose setup

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

```dockerfile
# 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"]
```

# Running the application

```
# Makefile - central repository of dev commands
.EXPORT_ALL_VARIABLES:  # useful if Makefile gets more elaborate
SHELL:=/bin/bash  # explicit shell declaration

setup_development:
  poetry install

run: _start_compose _db_migration

_start_compose:  # leading underscore disables tab-completion
  docker-compose up -d

_db_migration:
  poetry run alembic upgrade head  # needs to be tweaked to await DB
```

- need to modify the migrations to wait for the DB to get up

```
$ git clone <repo>
$ cd <repo>
$ make setup_development run  # voilà! the local app is running
```

# What we have already

- Live app running locally.
- The ability to experiment with the code and database.
    - Huge time-saver
    - quality improvement
- Very simple "getting started" instructions.

# Time for the tests! 🦾

# Integrated tests

- use internal interfaces (like unit tests)
- use external systems (e.g. PostgreSQL in the container)

```python
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:  # test code also calls out to Postgres
        query = select(Note).where(Note.id == id)
        saved_object = session.execute(query).scalar()
    assert saved_object.contents == note_contents
```

- no need for mocks
- …you should have more tests than that

# External tests (aka. functional/e2e)

- using **only** external interfaces (e.g. HTTP, data in DB)
- configuration as close to production as possible
- harder to debug, gotta look at the container logs

```
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)
```

# Missing code from the previous slide

```python
import uuid, httpx, pytest, tenacity

def test_store_and_retrieve_note(app_url):
    ...

# 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.
# Same technique can be used on DB migrations.
@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")
```

# Running the tests

```
SOURCES:=sample_backend tests  # source code directories for some commands

check: static_checks test  # one make target to validate the code

# 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
```

# Integrated and external tests - what do we get?

- proof that the app turns on
- higher confidence it’s working - app layers seem to work together
- less work than mock setups
- freedom to use full power of the tools
- slower than unit tests, still fast (if the app is fast)
- ⚠️ no full isolation between tests

# No isolation - a bit of chaos

- data reset between tests might be impractical
  - for Redis it’d be OK (but prevent test parallelization)
  - too slow in SQL
- some tests (e.g. get all notes) have to take that into account
  - collections can have unpredictable elements
  - need to build isolation into the data
- random app issues will bug you

# A bit of chaos - more realism

- production app doesn’t wipe the data all the time
- catching bugs before production:
  - local DB keeps growing
  - "flaky" tests point out race conditions
- fixing the "random app issues" increases quality
- if you can’t take it at the time: `` `docker-compose down -v` ``

# Organizing the tests

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

- explicit separation
  - numbers of high-level tests need to be controlled
- running faster test subgroups is easy
- more info about the 3 kinds of tests

# This works for complex applications

- battle-tested at 3 companies
- can integrate many systems (Kafka, Redis, RabbitMQ, etc.)
  - just need Docker images
- AWS locally - Localstack
  - weaker tools for GCP and Azure
- faking other REST APIs - Mountebank
  - check out mountepy

# Reloading the app code in the container

```
# 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
```

- no need to rebuild Docker image
- app in Docker restarts on any code change
- entr
- fd

# Continuos Integration / Delivery

# Organizing CI

- CI removes `docker-compose.override.yml` - prevent bad images
- CI uses the same Makefile
- subcommands of `make check` made into parallel tasks
- after checks succeed:
  - tag the built app image
  - push it out to a repo
  - use in deployments

# CI self-hosted runners: free ports problem

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

```python
# 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
```

# Promised, but skimmed over material

- debug code in the container
  - CLI
  - IDE, e.g. Intellij/Pycharm
- changes to production code for improved testability
  - every sleep in the app is configurable, now values for tests
  - it’s OK to add app features to increase testability
    - testability is an useful feature of the product
  - …others…

# Fin

🫠