

Containerizing HELIPORT

Development and Deployment with Docker

Marc Hanisch, <marc.hanisch@gfz-potsdam.de>
Helmholtz Zentrum Potsdam Deutsches GeoForschungsZentrum GFZ

HELIPORT Workshop, HZDR, 12.-14.06.2023

Containerizing HELIPOINT – What to expect

- Motivation
- Short Docker Introduction
- HELIPOINT Development Setup
- Hands On
- HELIPOINT Deployment Setup
- Benefits
- Drawbacks

Containerizing HELIPOINT – Motivation

- Fast and easy way to start HELIPOINT development
- No software requirements (except Docker)
- Secure and automated deployment for production



Image by [Jarosław Bialik](#)

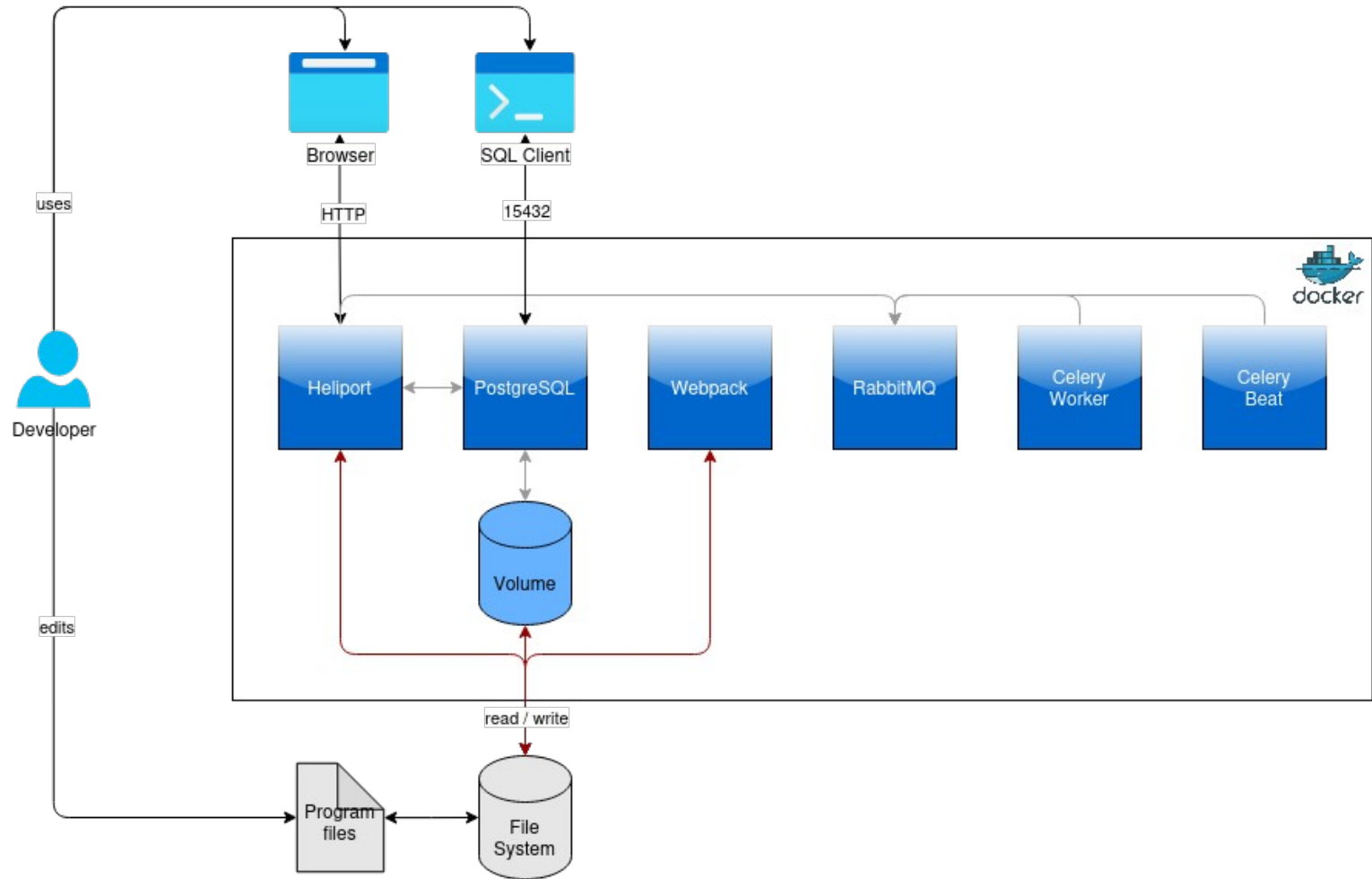
A short Docker Introduction

- OS-level virtualization
- Docker uses Linux kernel to provide sandboxes / containers
- Processes are isolated from the host machine
- Fine grained control of assigning resources, eg:
 - File system access
 - Network ports
 - CPU usage
- Image: read only template to build containers
- Container: running instance of an image, encapsulated environment
- Volume: mechanism for persisting data
- Getting started with Docker...

HELIPORT Development Setup

- Each service used by HELIPORT has its own container:
 - HELIPORT
 - PostgreSQL
 - RabbitMQ
 - Celery Worker
 - Celery Beat
 - Webpack
- Required network services are mapped to the host
 - HTTP via port 8000
 - PostgreSQL via port 15432
- Direct file system access for individual containers

HELIPORT Development Setup



Hands on – Starting & Stopping

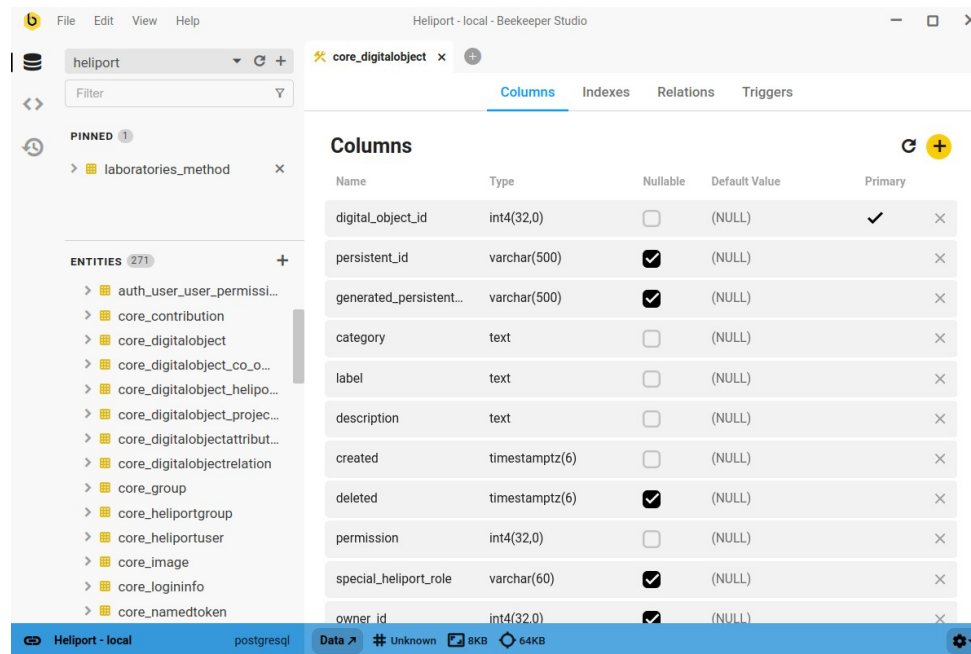
- Create `.env` file from `docker/development/.env.dist`
- Build and start services:
 - `./docker-compose.sh up [-d]`
- Open browser with <http://localhost:8000>
- `docker-compose.sh` script is a simple wrapper around `docker-compose`
- Inspect running containers:
 - `./docker-compose.sh ps`
- Stop and destroy services:
 - `./docker-compose.sh down [--volumes]`

Hands on – Running Commands

- Running:
 - heliport-cli commands:
 - `./docker-compose.sh exec heliport poetry run heliport-cli --help`
 - linters:
 - `./docker-compose.sh exec heliport poetry run black --check .`
 - `./docker-compose.sh exec webpack yarn run lint`
 - tests:
 - `./docker-compose.sh exec heliport poetry run heliport-cli test`
- Getting into the Python shell
 - `./docker-compose.sh exec heliport bash`
`$ poetry shell # interactive Poetry shell`
`(heliport-py3.11) $ heliport-cli shell # interactive Django shell`
`>>> from heliport.core.utils.colors import pick_color`

Hands on – Working with the Database

- Connect to the database:
 - `./docker-compose.sh exec postgres psql -U dev heliport`
- Use any PostgreSQL compatible client via port 15432



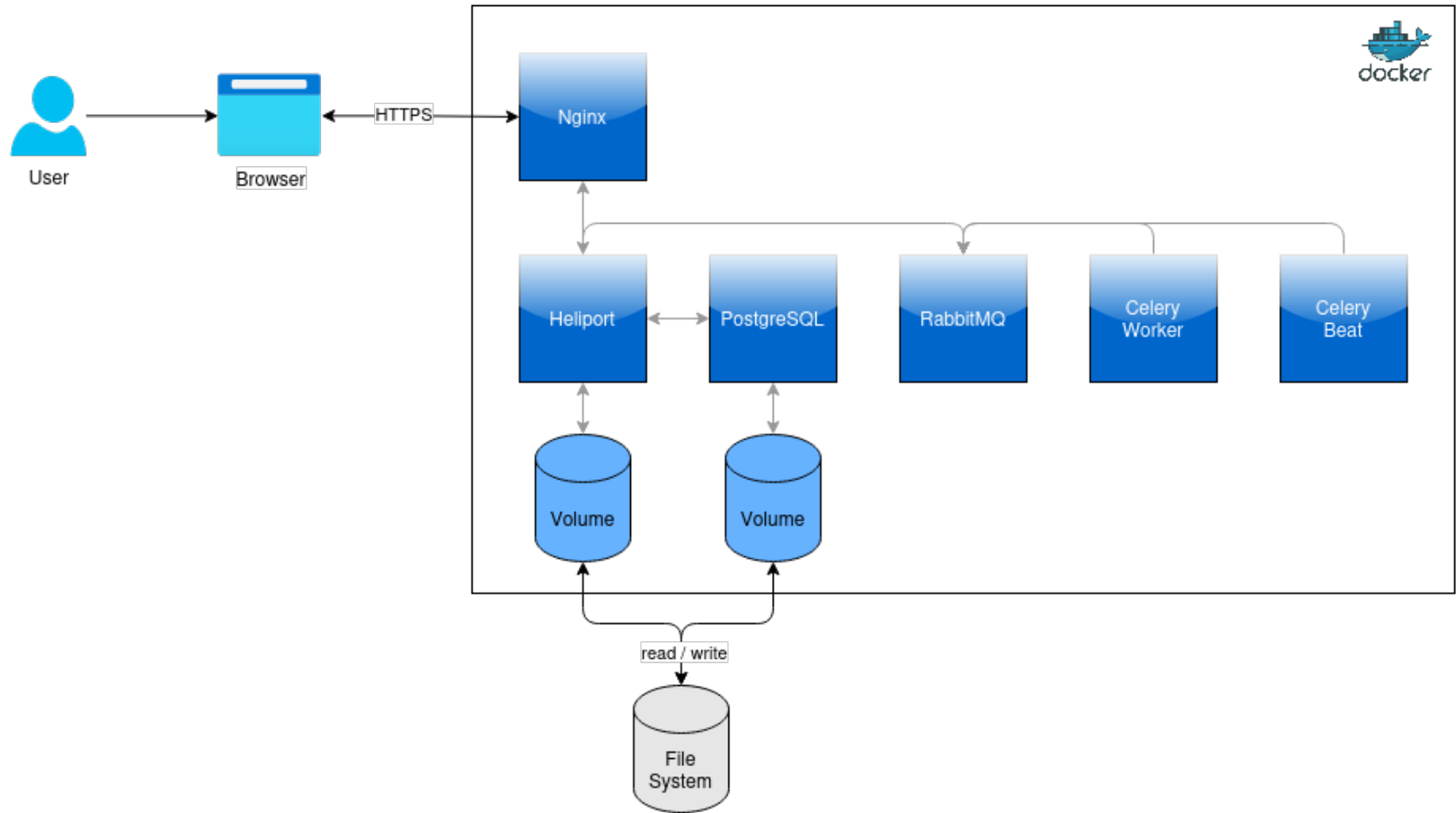
Hands on – Getting information

- Viewing logs:
 - `./docker-compose.sh logs [-f] heliport`
- Inspecting processes:
 - `./docker-compose.sh top heliport`

HELIPORT Deployment Setup

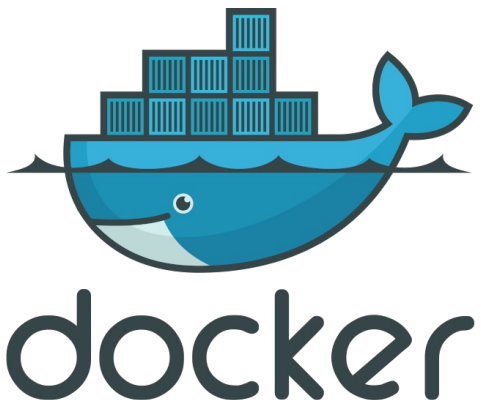
- Again, each service in a separate container
- Webpack runs at HELIPORT image build process
- Nginx container in front of HELIPORT, handles SSL, GZIP etc.
- Only ports 80 and 443 are shared with host
- Persistent Docker volumes for database and application files
- At GFZ, automated deployment via GitLab CI:
 - GitLab runner on codebase.helmholtz.cloud builds HELIPORT image when branch is tagged
 - Pushes HELIPORT image to GitLab registry
 - GitLab runner on production host pulls all images
 - Runs containers on production host

HELIPORT Deployment Setup



Benefits

- No dependencies to install (Python, poetry, PostgreSQL, RabbitMQ)
- No version conflicts with services on host machine
- Fine grained control of resources (file system access, network ports)
- Reliable & deterministic execution on systems supporting Docker
- Deployment setup can be tested on developer's machine



Drawbacks

- Adds another level of complexity
- Docker setup must be maintained for updates of
 - HELIPOINT itself
 - New base image versions
 - New Docker versions

Thank you very much.

(๐ 5 ๐)