Big Data, organisation and analysis

Summarising the Docker and the database hands-on lectures

Steffen M. Noe, Spring 2023

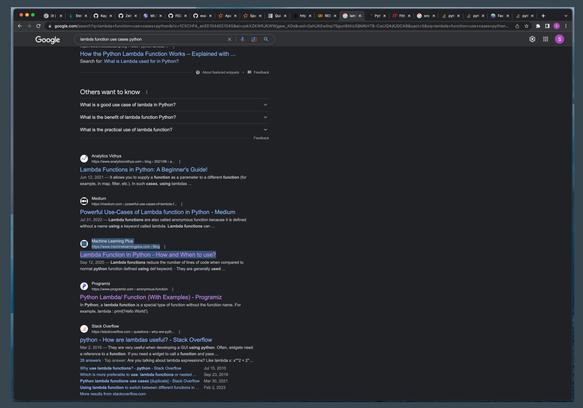
Overall principle (remainder)

Two container data pipeline

Container 1 (frontend)

Container 2 (backend)

Webbrowser



GET

POST

...

{ ... }

REST API

SQL

...

Database

Why did we use Docker?

- A container packs all needed dependencies into one place
- We can use “one” operating system (Linux) even if our development computer has another system (MacOS, Windows)
- Once build, the containers can be deployed to other platforms easily
- Containers can be used as development environment

Containers as development environment

Remember the containers we build!

Example server 1: simple “Hello World”

```dockerfile
FROM python:3.10-slim

# set the working directory
WORKDIR /code

# install dependencies
COPY ./requirements.txt ./
RUN pip install --no-cache-dir --upgrade -r requirements.txt

# copy the src to the folder
COPY ./src ./src

# start the server
CMD ["uvicorn", "src.main:app", "--host", "0.0.0.0", "--port", "80", "--reload"]
```

We use a Python image.
No need to install everything on my computer!

```
fastapi
pydantic
uvicorn
redis
debugpy
```

We define our dependencies that will be valid in the container!

```python
from fastapi import FastAPI


app = FastAPI()


@app.get("/")
def read_root():
    return {"Hello": "World123"}
```

We copy the python code (script) to run our service into the container!

Planning Docker projects

What language to use?

- We need to first thing about the language to write the app
- Python: easy, very popular, massive amount of libraries, has REST API libraries
- Golang (or Go): very fast, compiled language, specific for micro-services, REST API libraries, optimised to scale on large computing infrastructures, built in parallelisation features, backend tools.
  - Examples:

kubernetes

CockroachDB

All need parallel processing (concurrency) and high scalability

Language decision!

- It will depend on the purpose of your project!
- In many cases, a multiple language approach may be chosen.
- For user site (frontend) often Python can be a good choice, e.g. via Jupyter notebooks visualisation of data is rather easy. Pure web apps may also use JavaScript.
- For backend, database connections, or micro-services, Golang is often a good choice.

More planning for a Docker project

How many containers are needed?

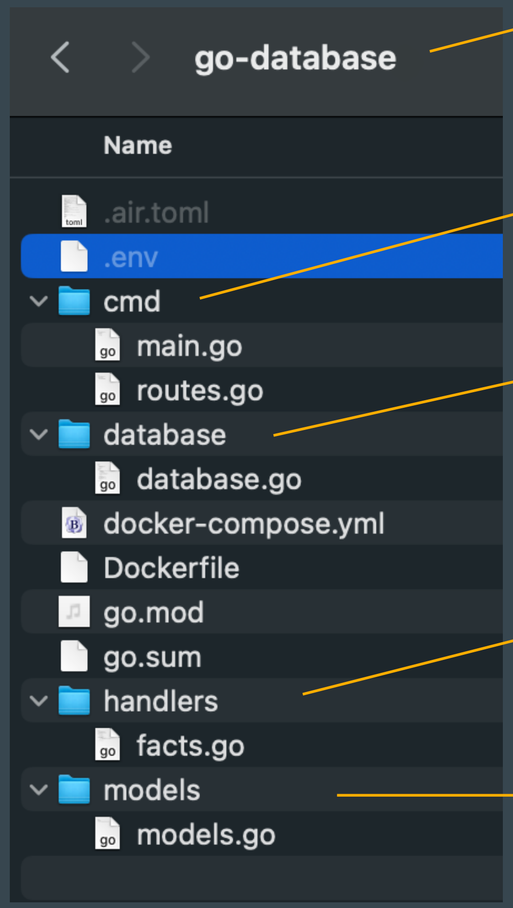
- The number of containers to be used is determined by the number of services that are needed
- In our example with the database two services, web and db were used, and we decided for two containers
- It is a good idea to use one container per service to avoid too much dependencies in the container

Setting the scene

The directory structure of the project

- It is a good idea to think about a systematic directory structure
- Use a base directory were all the Docker relevant files and possible environment variables will go
- Use a /cmd directory for the app entry point
- use different directories for packages
- Try logic splitting (database, models, handlers,...)

The directory structure



/go-database is the base directory of the project, here are the Docker related files (Dockerfile, docker-compose.yml) and the Golang created module dependency files

/cmd keeps the main.go and routes of the REST API

/database keeps the logic to write and read from the postgresql database via the gorm library

/handlers keep the functions that fill the data structures given by the models and are invoked by the routes (REST endpoints) This is the middleware to facilitate communication between the database and the frontend

/models are a description of the database schema's, i.e., the structures our datasets we operate on have.

Stepwise building the application

- We initialise the project with `go mod init <name>`
- We use VSCode as development tool
- The web service is build from the Dockerfile
- It reads the .env file to set the “secrets” in environment variables
- It runs the web service via the air command that allows dynamic loading on Unix and MacOSX filesystems. Unfortunately, Windows security does not allow this behaviour without changes

```yaml
version: '3.8'

services:
  web:
    build: .
    env_file:
      - .env
    ports:
      - 3002:3002
    volumes:
      - .:/usr/src/app
    command: air cmd/main.go - b 0.0.0.0
  db:
    image: postgres:alpine
    environment:
      - POSTGRES_USER=${DB_USER}
      - POSTGRES_PASSWORD=${DB_PASSWORD}
      - POSTGRES_NAME=${DB_NAME}
    ports:
      - 5432:5432
    volumes:
      - postgres-db:/var/lib/postgresql/data

volumes:
  postgres-db:
```

Stepwise building the application II

- The db service is based on the Postgres database running on Alpine Linux.
- It fetches the environment variables to set a database user, password and database name
- for persistency, a volume is created on the local hard drive. In my case it is in the docker directory path /var/lib/docker/volumes/go-database_postgres-db/_data
-

```yaml
version: '3.8'

services:
  web:
    build: .
    env_file:
      - .env
    ports:
      - 3002:3002
    volumes:
      - .:/usr/src/app
    command: air cmd/main.go - b 0.0.0.0
  db:
    image: postgres:alpine
    environment:
      - POSTGRES_USER=${DB_USER}
      - POSTGRES_PASSWORD=${DB_PASSWORD}
      - POSTGRES_NAME=${DB_NAME}
    ports:
      - 5432:5432
    volumes:
      - postgres-db:/var/lib/postgresql/data

volumes:
  postgres-db:
```

The Docker application is organised in Layers