

COMP 350

Introduction to DevOps

Lecture 8

Docker Compose

Hakan Ayrar

What is Docker Compose?

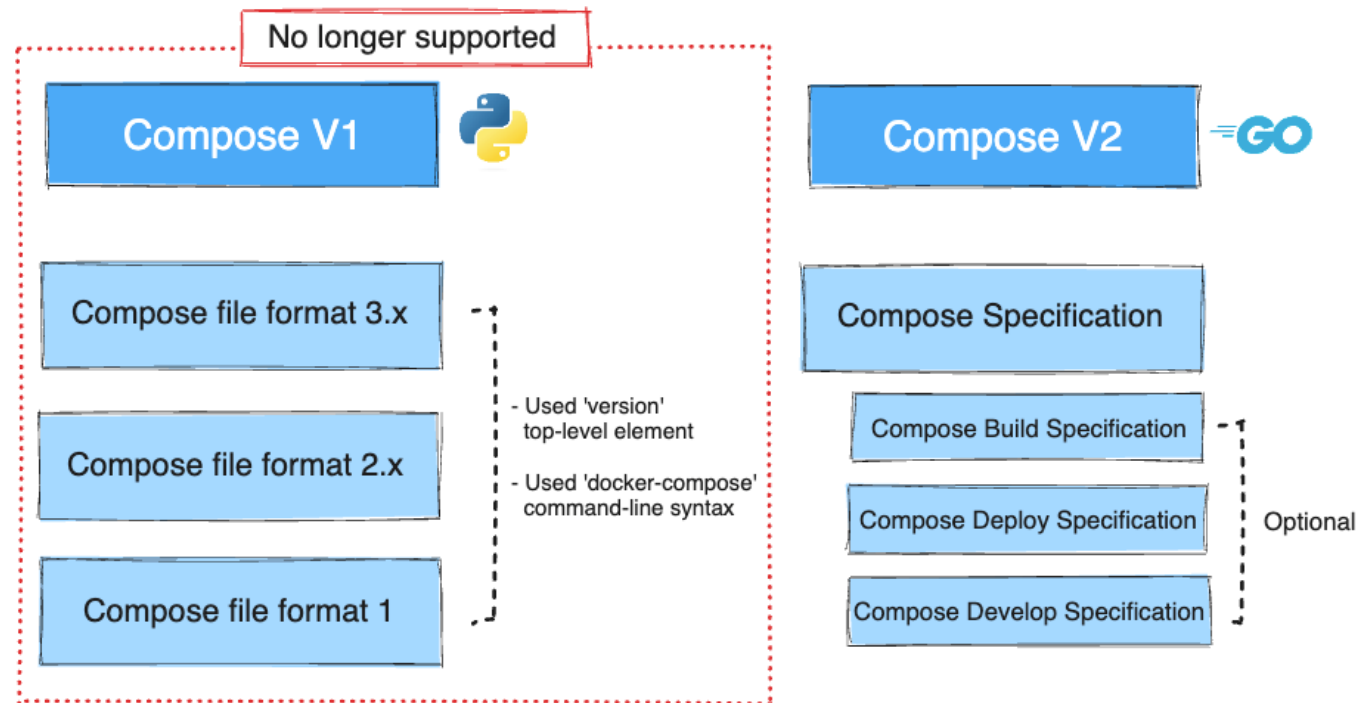
- Docker Compose is a tool for defining and running multi-container applications.
- Compose simplifies the control of your entire application stack, making it easy to manage services, networks, and volumes in a single, comprehensible YAML configuration file.
 - With a single command, you create and start all the services from your configuration file.
- Compose works in all environments; production, staging, development, testing, as well as CI workflows.
- It has commands for managing the whole lifecycle of your application:
 - Start, stop, and rebuild services
 - View the status of running services
 - Stream the log output of running services
 - Run a one-off command on a service

Why use Compose?

- Using Docker Compose offers several benefits that streamline the development, deployment, and management of containerized applications:
 - **Simplified control:** Docker Compose allows you to define and manage multi-container applications in a single YAML file. This simplifies the complex task of orchestrating and coordinating various services, making it easier to manage and replicate your application environment.
 - **Efficient collaboration:** Docker Compose configuration files are easy to share, facilitating collaboration among developers, operations teams, and other stakeholders. This collaborative approach leads to smoother workflows, faster issue resolution, and increased overall efficiency.
 - **Rapid application development:** Compose caches the configuration used to create a container. When you restart a service that has not changed, Compose re-uses the existing containers. Re-using containers means that you can make changes to your environment very quickly.
 - **Portability across environments:** Compose supports variables in the Compose file. You can use these variables to customize your composition for different environments, or different users.

History and development of Docker Compose

- The currently supported version of the Docker Compose CLI is Compose V2 which is defined by the Compose Specification.



Docker Compose CLI versioning

- Version one of the Docker Compose command-line binary was first released in 2014.
 - It was written in Python, and is invoked with **docker-compose**.
 - Typically, Compose V1 projects include a top-level **version** element in the **compose.yml** file, with values ranging from **2.0** to **3.8**, which refer to the specific file formats.
- Version two of the Docker Compose command-line binary was announced in 2020, is written in Go, and is invoked with **docker compose**.
 - Compose V2 ignores the **version** top-level element in the **compose.yml** file.

Compose file format versioning

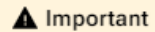
- The Docker Compose CLIs are defined by specific file formats.
- Three major versions of the Compose file format for Compose V1 were released:
 - Compose file format 1 with Compose 1.0.0 in 2014
 - Compose file format 2.x with Compose 1.6.0 in 2016
 - Compose file format 3.x with Compose 1.10.0 in 2017
- Compose file format 1 is substantially different to all the following formats as it lacks a top-level services key.
 - Its usage is historical and files written in this format don't run with Compose V2.
- Compose file format 2.x and 3.x are very similar to each other, but the latter introduced many new options targeted at Swarm deployments.
- To address confusion around Compose CLI versioning, Compose file format versioning, and feature parity depending on whether Swarm mode was in use, file format 2.x and 3.x were merged into the Compose Specification.
- Compose V2 uses the Compose Specification for project definition.
 - Unlike the prior file formats, the Compose Specification is rolling and makes the **version** top-level element optional.
 - Compose V2 also makes use of optional specifications - **Deploy**, **Develop** and **Build**.
- To make migration easier, Compose V2 has backwards compatibility for certain elements that have been deprecated or changed between Compose file format 2.x/3.x and the Compose Specification.

How Compose works (compose application model)

- Docker Compose relies on a YAML configuration file, usually named **compose.yaml**.
- Computing components of an application are defined as **services**.
 - A service is an abstract concept implemented on platforms by running the same container image, and configuration, one or more times.
- Services communicate with each other through **networks**.
 - In the Compose Specification, a network is a platform capability abstraction to establish an IP route between containers within services connected together.
- Services store and share persistent data into **volumes**.
 - The Specification describes such a persistent data as a high-level filesystem mount with global options.
- Some services require configuration data that is dependent on the runtime or platform. For this, the Specification defines a dedicated **configs** concept.
 - From a service container point of view, configs are comparable to volumes, in that they are files mounted into the container. But the actual definition involves distinct platform resources and services, which are abstracted by this type.
- A **secret** is a specific flavor of configuration data for sensitive data that should not be exposed without security considerations.
 - Secrets are made available to services as files mounted into their containers, but the platform-specific resources to provide sensitive data are specific enough to deserve a distinct concept and definition within the Compose specification.

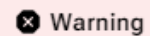
How to install?

- Scenario 1: Install the Compose plugin
 - if you already have Docker Engine and Docker CLI installed, you can install the Compose plugin



This is only available on Linux

- Scenario 2: Install the Compose standalone
 - You can install the Compose standalone on Linux or on Windows Server.



This install scenario is not recommended and is only supported for backward compatibility purposes.

Install using the repository

1. Set up the repository. Find distro-specific instructions in:
[Ubuntu](#) | [CentOS](#) | [Debian](#) | [Raspberry Pi OS](#) | [Fedora](#) | [RHEL](#) | [SLES](#).
2. Update the package index, and install the latest version of Docker Compose:

- For Ubuntu and Debian, run:

```
$ sudo apt-get update
$ sudo apt-get install docker-compose-plugin
```

- For RPM-based distros, run:

```
$ sudo yum update
$ sudo yum install docker-compose-plugin
```

On Linux

i Compose standalone

Note that Compose standalone uses the `-compose` syntax instead of the current standard syntax `compose`.

For example type `docker-compose up` when using Compose standalone, instead of `docker compose up`.

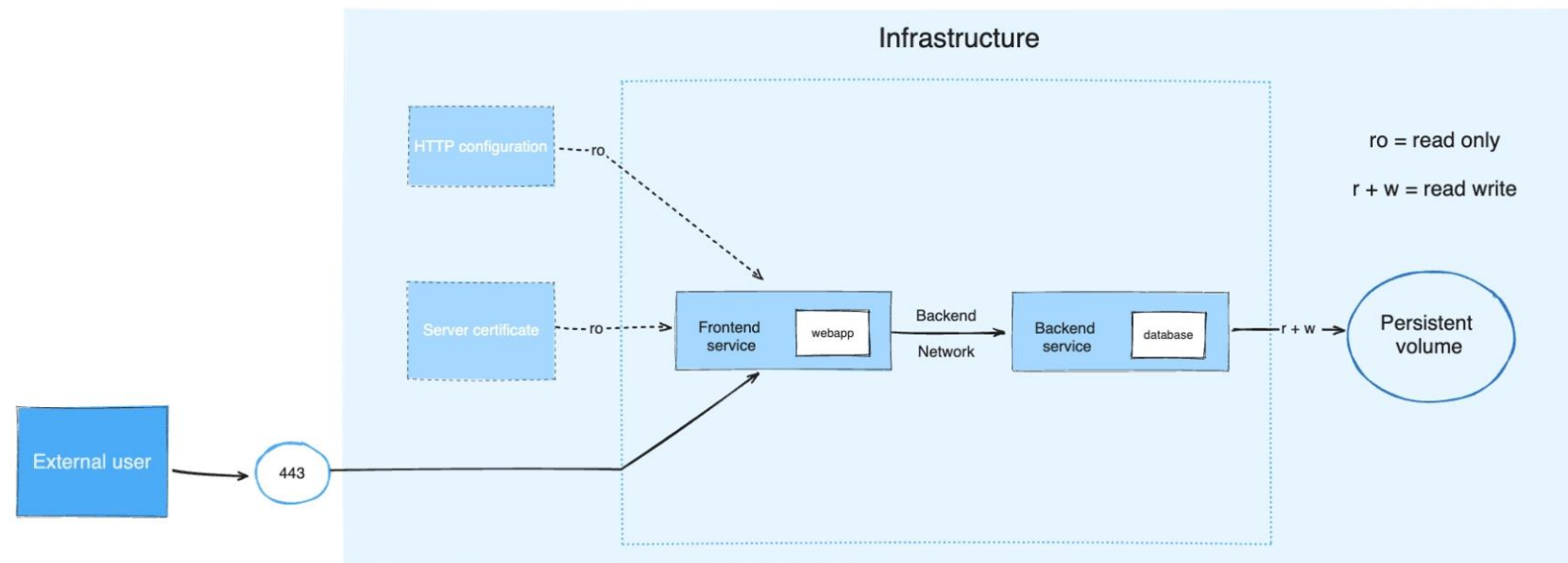
1. To download and install Compose standalone, run:

```
$ curl -SL https://github.com/docker/compose/releases/download/v2.25.0/docker-compose-lin
```

2. Apply executable permissions to the standalone binary in the target path for the installation.

Example 1

- Consider an application split into a frontend web application and a backend service.
 - The **frontend** is configured at runtime with an HTTP configuration file managed by infrastructure, providing an external domain name, and an HTTPS server certificate injected by the platform's secured secret store.
 - The **backend** stores data in a persistent volume.
 - Both services communicate with each other on an isolated back-tier network, while the frontend is also connected to a front-tier network and exposes port 443 for external usage.



Example 1

Docker compose file

- The example application is composed of the following parts:
 - 2 **services**, backed by Docker images: webapp and database
 - 1 **secret** (HTTPS certificate), injected into the frontend
 - 1 **configuration** (HTTP), injected into the frontend
 - 1 persistent **volume**, attached to the backend
 - 2 **networks**

```
services:
  frontend:
    image: example/webapp
    ports:
      - "443:8043"
    networks:
      - front-tier
      - back-tier
    configs:
      - httpd-config
    secrets:
      - server-certificate

  backend:
    image: example/database
    volumes:
      - db-data:/etc/data
    networks:
      - back-tier


volumes:
  db-data:
    driver: flocker
    driver_opts:
      size: "10GiB"

configs:
  httpd-config:
    external: true

secrets:
  server-certificate:
    external: true

networks:
  # The presence of these objects is sufficient to define them
  front-tier: {}
  back-tier: {}
```

Example 2

- Using the **Flask** framework, the application features a hit counter in **Redis**, providing a practical example of how **Docker Compose** can be applied in web development scenarios.
- Step 1: Setup 
 - Step 1.1 – create a directory
 - Step 1.2 – create the application in flask

1. Create a directory for the project:

```
$ mkdir composetest  
$ cd composetest
```

2. Create a file called `app.py` in your project directory and paste the following code in:

```
import time  
  
import redis  
from flask import Flask  
  
app = Flask(__name__)  
cache = redis.Redis(host='redis', port=6379)  
  
def get_hit_count():  
    retries = 5  
    while True:  
        try:  
            return cache.incr('hits')  
        except redis.exceptions.ConnectionError as exc:  
            if retries == 0:  
                raise exc  
            retries -= 1  
            time.sleep(0.5)  
  
@app.route('/')  
def hello():  
    count = get_hit_count()  
    return 'Hello World! I have been seen {} times.\n'.format(count)
```

In this example, `redis` is the hostname of the redis container on the application's network and the default port, `6379` is used.

Example 2

- Step 1.3 – requirements.txt
- Step 1.4 - Dockerfile

3. Create another file called `requirements.txt` in your project directory and paste the following code in:

```
flask  
redis
```

4. Create a `Dockerfile` and paste the following code in:

```
# syntax=docker/dockerfile:1  
FROM python:3.10-alpine  
WORKDIR /code  
ENV FLASK_APP=app.py  
ENV FLASK_RUN_HOST=0.0.0.0  
RUN apk add --no-cache gcc musl-dev linux-headers  
COPY requirements.txt requirements.txt  
RUN pip install -r requirements.txt  
EXPOSE 5000  
COPY . .  
CMD ["flask", "run", "--debug"]
```

Example 2

- Step 2: Define services in a Compose file

Compose simplifies the control of your entire application stack, making it easy to manage services, networks, and volumes in a single, comprehensible YAML configuration file.

Create a file called `compose.yaml` in your project directory and paste the following:

```
services:
  web:
    build: .
    ports:
      - "8000:5000"
  redis:
    image: "redis:alpine"
```

This Compose file defines two services: `web` and `redis`.

The `web` service uses an image that's built from the `Dockerfile` in the current directory. It then binds the container and the host machine to the exposed port, `8000`. This example service uses the default port for the Flask web server, `5000`.

The `redis` service uses a public [Redis](#) image pulled from the Docker Hub registry.

Example 2

- Step 3: Build and run your app with Compose
 - Step 3.1 – start docker compose
 - Step 3.2 – see the result on browser

With a single command, you create and start all the services from your configuration file.

1. From your project directory, start up your application by running `docker compose up`.

```
$ docker compose up

Creating network "composetest_default" with the default driver
Creating composetest_web_1 ...
Creating composetest_redis_1 ...
Creating composetest_web_1
Creating composetest_redis_1 ... done
Attaching to composetest_web_1, composetest_redis_1
web_1      | * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
redis_1    | 1:C 17 Aug 22:11:10.480 # o000o000o000o Redis is starting o000o000o000o
redis_1    | 1:C 17 Aug 22:11:10.480 # Redis version=4.0.1, bits=64, commit=00000000, modif
redis_1    | 1:C 17 Aug 22:11:10.480 # Warning: no config file specified, using the default
web_1      | * Restarting with stat
redis_1    | 1:M 17 Aug 22:11:10.483 * Running mode=standalone, port=6379.
redis_1    | 1:M 17 Aug 22:11:10.483 # WARNING: The TCP backlog setting of 511 cannot be en
web_1      | * Debugger is active!
redis_1    | 1:M 17 Aug 22:11:10.483 # Server initialized
redis_1    | 1:M 17 Aug 22:11:10.483 # WARNING you have Transparent Huge Pages (THP) support
web_1      | * Debugger PIN: 330-787-903
redis_1    | 1:M 17 Aug 22:11:10.483 * Ready to accept connections
```

Compose pulls a Redis image, builds an image for your code, and starts the services you defined. In this case, the code is statically copied into the image at build time.


2. Enter `http://localhost:8000/` in a browser to see the application running.

If this doesn't resolve, you can also try `http://127.0.0.1:8000`.

You should see a message in your browser saying:

```
Hello World! I have been seen 1 times.
```

Example 2

- Step 4.1: Compose Watch
- Edit the **compose.yaml** file in your project directory to use **watch** so you can preview your running Compose services which are automatically updated as you edit and save your code: 
- Whenever a file is changed, Compose syncs the file to the corresponding location under **/code** inside the container. Once copied, the bundler updates the running application without a restart.
- For this example to work, you must add the **--debug** option to the Dockerfile.
 - The **--debug** option in **Flask** enables automatic code reload, making it possible to work on the backend API without the need to restart or rebuild the container.
 - After changing the .py file, subsequent API calls will use the new code, but the browser UI will not automatically refresh in this small example.
 - Most frontend development servers include native live reload support that works with Compose.

```
services:
  web:
    build: .
    ports:
      - "8000:5000"
    develop:
      watch:
        - action: sync
          path: .
          target: /code
  redis:
    image: "redis:alpine"
```

Example 2

- **Compose Watch** is available in Docker Compose version 2.22 and later.
- Use **watch** to automatically update and preview your running Compose services as you edit and save your code.
- For many projects, this allows for a hands-off development workflow once Compose is running, as services automatically update themselves when you save your work.
- **watch** adheres to the following file path rules:
 - All paths are relative to the project directory
 - Directories are watched recursively
 - Glob patterns aren't supported
 - Rules from **.dockerignore** apply
 - Use **ignore** to defined additional paths to be ignored (same syntax)
 - Temporary/backup files for common IDEs (Vim, Emacs, JetBrains, & more) are ignored automatically
 - **.git** directories are ignored automatically
- You don't need to switch on **watch** for all services in a Compose project. In some instances, only part of the project, for example the Javascript frontend, might be suitable for automatic updates.

Example 2

- Step 4.2: Re-build and run the app with Compose
- From your project directory, type **docker compose watch** or **docker compose up --watch** to build and launch the app and start the file watch mode.

```
$ docker compose watch
[+] Running 2/2
 ✓ Container docs-redis-1 Created
 ✓ Container docs-web-1    Recreated
Attaching to redis-1, web-1
    Ⓢ watch enabled
...
```

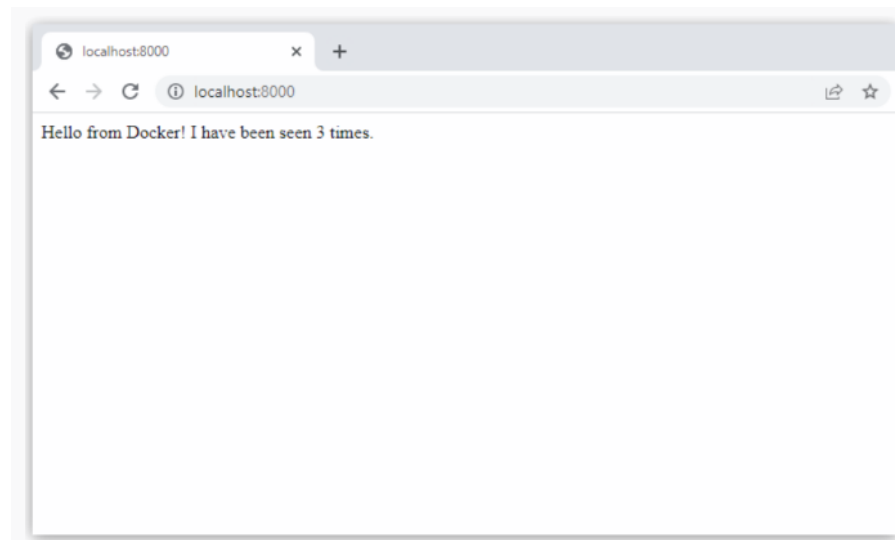
- Check the Hello World message in a web browser again, and refresh to see the count increment.

Example 2

- Step 4.3: Update the application.
- To see Compose Watch in action: Change the greeting in app.py and save it. For example, change the ***Hello World!*** message to ***Hello from Docker!***:

```
return 'Hello from Docker! I have been seen {} times.\n'.format(count)
```

- Refresh the app in your browser. The greeting should be updated, and the counter should still be incrementing.



Environment Variables

- Use environment variables to pass configuration information to containers at runtime.
- Environment variables are key-value pairs that contain data that can be used by processes running inside a Docker container.
 - They are often used to configure application settings and other parameters that may vary between different environments, such as development, testing, and production.
- This section covers:
 - The various ways you can set environment variables with Compose.
 - How environment variable precedence works.
 - The correct syntax for an environment file.
 - Changing pre-defined environment variables.


Ways you can set environment variables with Compose

- With Compose, there are multiple ways you can set environment variables in your containers. You can use either your **Compose file**, or the **CLI**.
 - Be aware that each method is subject to environment variable precedence.

1. Compose file

- Substitute with an .env file
- An **.env file** in Docker Compose is a text file used to define environment variables that should be made available to Docker containers when running docker compose up.
- This file typically contains key-value pairs of environment variables, and it allows you to centralize and manage configuration in one place.
- The .env file is useful if you have multiple environment variables you need to store.

.env file

- The **.env file** is the default method for setting environment variables in your containers.
 - The .env file should be placed at the root of the project directory next to your compose.yaml file.
- This is a simple example: 
- When you run **docker compose up**, the **web** service defined in the Compose file interpolates in the image **webapp:v1.5** which was set in the **.env** file.
- You can verify this with the config command, which prints your resolved application config to the terminal:

```
$ cat .env
```

```
TAG=v1.5
```

```
$ cat compose.yaml
```

```
services:
```

```
  web:
```

```
    image: "webapp:${TAG}"
```

```
$ docker compose config
```


```
services:
```

```
  web:
```

```
    image: 'webapp:v1.5'
```

Environment attribute

- You can set environment variables directly in your Compose file without using an .env file, with the **environment** attribute in your **compose.yml**. It works in the same way as **docker run -e VARIABLE=VALUE ...** command line




```
web:
  environment:
    - DEBUG=1
```

- You can choose not to set a value and pass the environment variables from your shell straight through to your containers.
 - The value of the DEBUG variable in the container is taken from the value for the same variable in the shell in which Compose is run. Note that in this case no warning is issued if the DEBUG variable in the shell environment is not set.



```
web:
  environment:
    - DEBUG
```

- You can also take advantage of interpolation.
 - The result is similar to the one above but Compose gives you a warning if the DEBUG variable is not set in the shell environment.



```
web:
  environment:
    - DEBUG=${DEBUG}
```

Python/Flask with Nginx proxy and MySQL DB

- Docker Compose Examples for Some Common Stacks

```
services:
  db:
    # We use a mariadb image which supports both amd64 & arm64 architecture
    image: mariadb:10-focal
    # If you really want to use MySQL, uncomment the following line
    #image: mysql:8
    command: '--default-authentication-plugin=mysql_native_password'
    restart: always
    healthcheck:
      test: ['CMD-SHELL', 'mysqladmin ping -h 127.0.0.1 --password="$$(cat /run/secrets/db-password)" --silent']
      interval: 3s
      retries: 5
      start_period: 30s
    secrets:
      - db-password
    volumes:
      - db-data:/var/lib/mysql
    networks:
      - backnet
    environment:
      - MYSQL_DATABASE=example
      - MYSQL_ROOT_PASSWORD_FILE=/run/secrets/db-password
    expose:
      - 3306
      - 33060
```

```
backend:
  build:
    context: backend
    target: builder
  restart: always
  secrets:
    - db-password
  ports:
    - 8000:8000
  networks:
    - backnet
    - frontnet
  depends_on:
    db:
      condition: service_healthy
```

```
proxy:
  build: proxy
  restart: always
  ports:
    - 80:80
  depends_on:
    - backend
  networks:
    - frontnet
```

```
volumes:
  db-data:

secrets:
  db-password:
    file: db/password.txt

networks:
  backnet:
  frontnet:
```

Project structure:

```
.
├── compose.yaml
├── flask
│   ├── Dockerfile
│   ├── requirements.txt
│   └── server.py
├── nginx
│   └── nginx.conf
```

Python/Flask with Nginx proxy and MySQL DB

- backend
 - Dockerfile

```
FROM --platform=$BUILDPLATFORM python:3.10-alpine AS builder

WORKDIR /code
COPY requirements.txt /code
RUN --mount=type=cache,target=/root/.cache/pip \
    pip3 install -r requirements.txt

COPY . .

ENV FLASK_APP hello.py
ENV FLASK_ENV development
ENV FLASK_RUN_PORT 8000
ENV FLASK_RUN_HOST 0.0.0.0

EXPOSE 8000

CMD ["flask", "run"]

FROM builder AS dev-envs

RUN <<EOF
apk update
apk add git
EOF

RUN <<EOF
addgroup -S docker
adduser -S --shell /bin/bash --ingroup docker vscode
EOF

# install Docker tools (cli, buildx, compose)
COPY --from=gloursdocker/docker / /

CMD ["flask", "run"]
```

- proxy
 - DockerFile

```
FROM nginx:1.13-alpine
COPY conf /etc/nginx/conf.d/default.conf
```

Expected result

Listing containers should show three containers running and the port mapping as below:

```
$ docker compose ps
```

NAME	COMMAND	SERVICE	STATUS	PORTS
nginx-flask-mysql-backend-1	"flask run"	backend	running	0.0.0.0:8000->8000/tcp
nginx-flask-mysql-db-1	"docker-entrypoint.s..."	db	running (healthy)	3306/tcp, 33060/tcp
nginx-flask-mysql-proxy-1	"nginx -g 'daemon of..."	proxy	running	0.0.0.0:80->80/tcp

