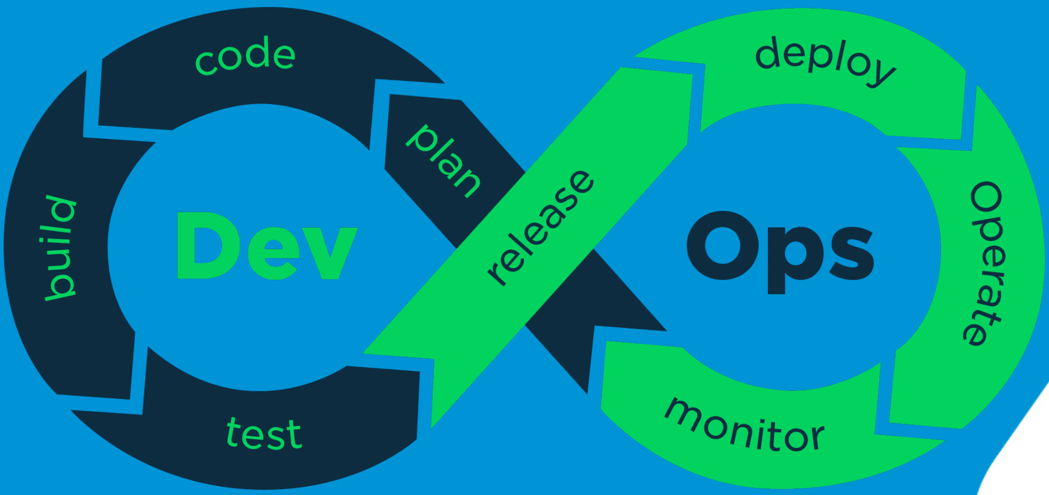
{EPITECH}

POPEYE

SET SAIL FOR THE AMAZING WORLD OF CONTAIN-
ERS



POPEYE

Docker is one of today's most popular containerization software.

It allows the packaging of applications and the runtime environments they need (down to the operating systems), which in return allows them to work wherever Docker is installed.

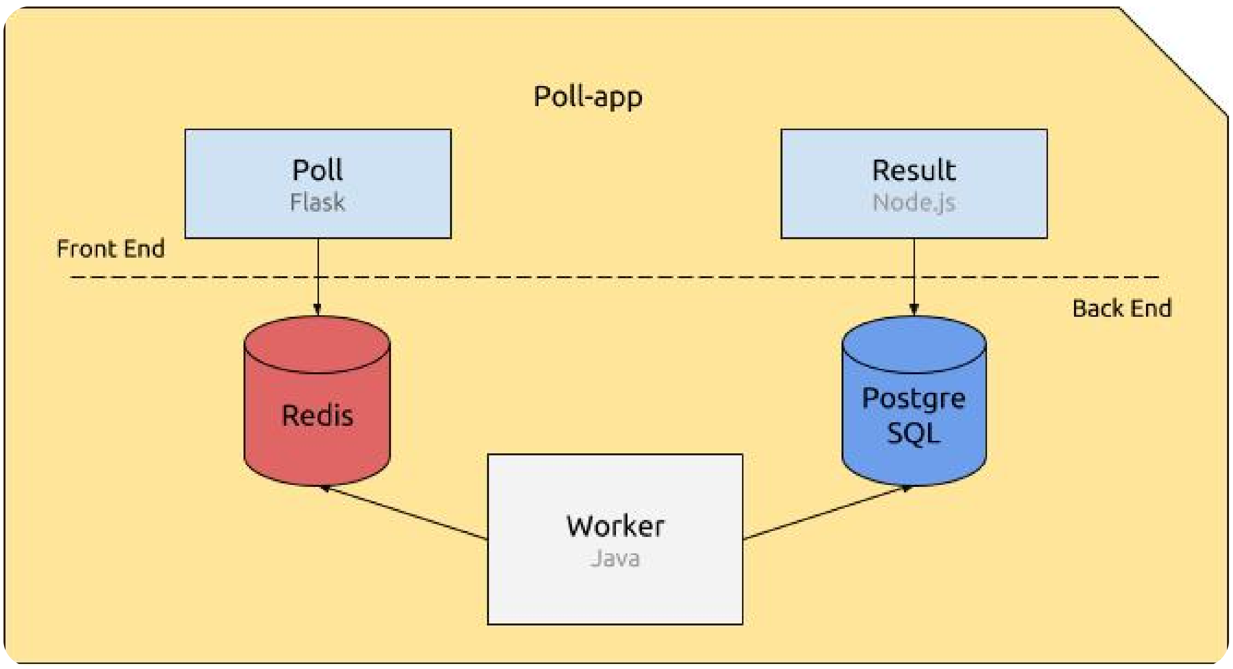
Like the brave sailor that Popeye is, containers can also confidently sail across the vast ocean of operating systems and configurations, being sure of working wherever they might end. As such, containers can be used on any host OS where Docker is installed.

During this project, you are going to master the basics of containerizing applications and describing multi-containers infrastructures with Docker and Docker Compose.

General description

You will have to containerize and define the deployment of a simple web poll application.

The application is composed of 5 elements:
- ***Poll***, a Flask Python web application that gathers votes and push them into a Redis queue.
- A **Redis queue**, which holds the votes sent by the Poll application, awaiting for them to be consumed by the Worker.
- The ***Worker***, a Java application which consumes the votes being in the Redis queue, and stores them into a PostgreSQL database.
- A **PostgreSQL database**, which (persistently) stores the votes stored by the Worker.
- ***Result***, a Node.js web application that fetches the votes from the database and displays the... well, result. ;)



Do not worry, you do not have to code them!
Popeye is generous: the entire application's code is given to you on the intranet's project page.

In DevOps, it is especially important that you take the time to research and understand the technologies you are asked to work with, as you will need to understand how and by which way you can configure them as needed.

Application configuration

The Poll, Worker, and Result elements are all designed to be configured using specific environment variables, described below.

Poll

- ✓ `REDIS_HOST`: the hostname of the Redis service to connect to.

Result

- ✓ `POSTGRES_HOST`: the hostname of the database service to connect to.
- ✓ `POSTGRES_PORT`: the port the database service is listening on.
- ✓ `POSTGRES_DB`: the name of the PostgreSQL database to connect to.
- ✓ `POSTGRES_USER`: the user that will be used to connect to the database.
- ✓ `POSTGRES_PASSWORD`: the password of the user that will be used to connect to the database.

Worker

The Worker uses the same environment variables as both Poll and Result.

You will have to set these environment variables with **Docker Compose**.
Do **not** set them directly in the Dockerfiles.

Docker images

You have to create 3 images.

The specifications for each image are as described below.

Poll

The image must be based on a Python official image.

The dependencies of the application can be installed using the following command:

```
pip3 install -r requirements.txt
```

The application must expose and run on the port 80, and can be started with:

```
flask run --host=0.0.0.0 --port=80
```

Result

The image must be based on an official Node.js version 20 Alpine image.

The application must expose and run on the port 80.

The dependencies of the application can be installed using the following command:

```
npm install
```

Be careful about the location where this command has to be run.

The `node_modules` directory must be excluded from the build context.

Worker

The image will be built using a multi-stage build.

First stage - compilation

The first stage must be based on `maven:3.9.6-eclipse-temurin-21-alpine` and be named `builder`.

It must be used to build (of course) and package the Worker application using the following commands:
- `mvn dependency:resolve`, from within the directory containing `pom.xml`;
- then, `mvn package`, from within the directory containing the `src` directory.

It generates a file in the `target` directory named `worker-jar-with-dependencies.jar` (relative to your `WORKDIR`).

Second stage - run

The second stage must be based on `eclipse-temurin:21-jre-alpine`.

This is the one really running the worker using the command:

```
java -jar worker-jar-with-dependencies.jar
```

Your Docker images must be simple, lightweight and not bring too much things.

Popeye does not like the `ENTRYPOINT` instruction, so you must not use it in this project.

Docker Compose

You now have 3 Dockerfiles that create 3 isolated images.
It is now time to make them all work together using Docker Compose!

Create a `docker-compose.yml` file that will be responsible for running and linking your containers.

Your Docker Compose file should contain:

5 services:

- ✓ `poll`:
  - builds your `poll` image;
  - redirects port 5000 of the host to the port 80 of the container.
  - correctly sets the necessary environment variable.
- ✓ `redis`:
  - uses an existing official image of Redis 7;
  - opens port 6379.
- ✓ `worker`:
  - builds your `worker` image;
  - correctly sets the same environment variables as both the `poll` and `result` services.
- ✓ `db`:
  - represents the database that will be used by the apps;
  - uses an existing official image of PostgreSQL 16;
  - has its database schema created during container first start;
  - correctly sets the appropriate environment variables.
- ✓ `result`:
  - builds your `result` image;
  - redirects port 5001 of the host to the port 80 of the container;
  - correctly sets the necessary environment variables.

Databases must be launched before the services that use them, because these services are *depending on* them.

3 networks:

- ✓ `poll-tier` to allow `poll` to communicate with `redis`.
- ✓ `result-tier` to allow `result` to communicate with `db`.
- ✓ `back-tier` to allow `worker` to communicate with `redis` and `db`.

You **must** use networks. Using the `links` property is **forbidden**.

1 named volume:

- ✓ `db-data` which allows the database's data to be persistent, if the container dies.

The path of data in the official PostgreSQL image is documented on Docker Hub.

Do not add unnecessary data, such as extra volumes, extra networks, unnecessary inter-container dependencies, or unnecessary container commands/entrypoints.

Once your `docker-compose.yml` is complete, you should be able to run all the services and access the different web pages:

- ✓ Poll at `http://localhost:5000`;
- ✓ and Result at `http://localhost:5001`.

You should be able to vote on Poll's page and see its effect on Result's page.

Your containers must restart automatically when they stop unexpectedly.

The environment variables or environment files (depending on what you choose to use) **have** to be defined in the `docker-compose.yml` file.

The environment variables defined by `ENV` instructions in Dockerfiles, or the environment files not specified in the `docker-compose.yml` file will **not** be taken into account.

Technical formalities

Your project will be entirely evaluated with Automated Tests, by analyzing your configuration files (the different Dockerfiles and `docker-compose.yml`).

In order to be correctly evaluated, your repository must at least contain the following files:

```
.
|-- docker-compose.yml
|-- schema.sql
|-- poll
|    |-- Dockerfile
|-- result
|    |-- Dockerfile
\-- worker
     \-- Dockerfile
```

`poll`, `result` and `worker` are the directories containing the services, given to you on the intranet.
`schema.sql` is also given to you on the intranet.

Going further

That's an amazing idea! Here are some bonuses suggestions:

- ✓ Add a password to Redis (you will need to update Poll's and Worker's code accordingly).
- ✓ Add a `healthcheck` to your services.

v 2.1

{EPITECH}