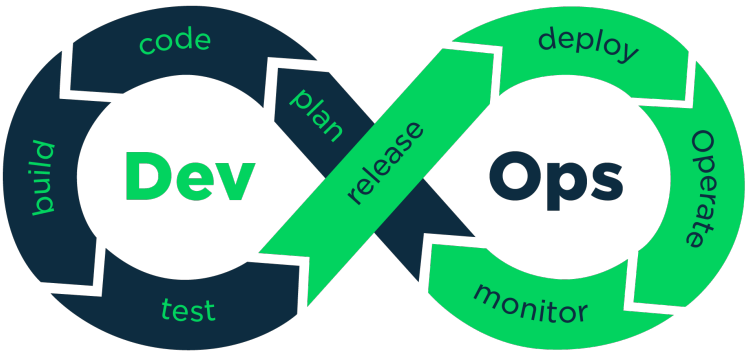
B2 - Introduction to DevOps

B-DOP-200

Popeye

Set sail for the amazing world of containers

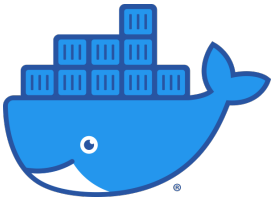


1.0

Popeye

- The totality of your source files, except all useless files (binary, temp files, obj files,...), must be included in your delivery.

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



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

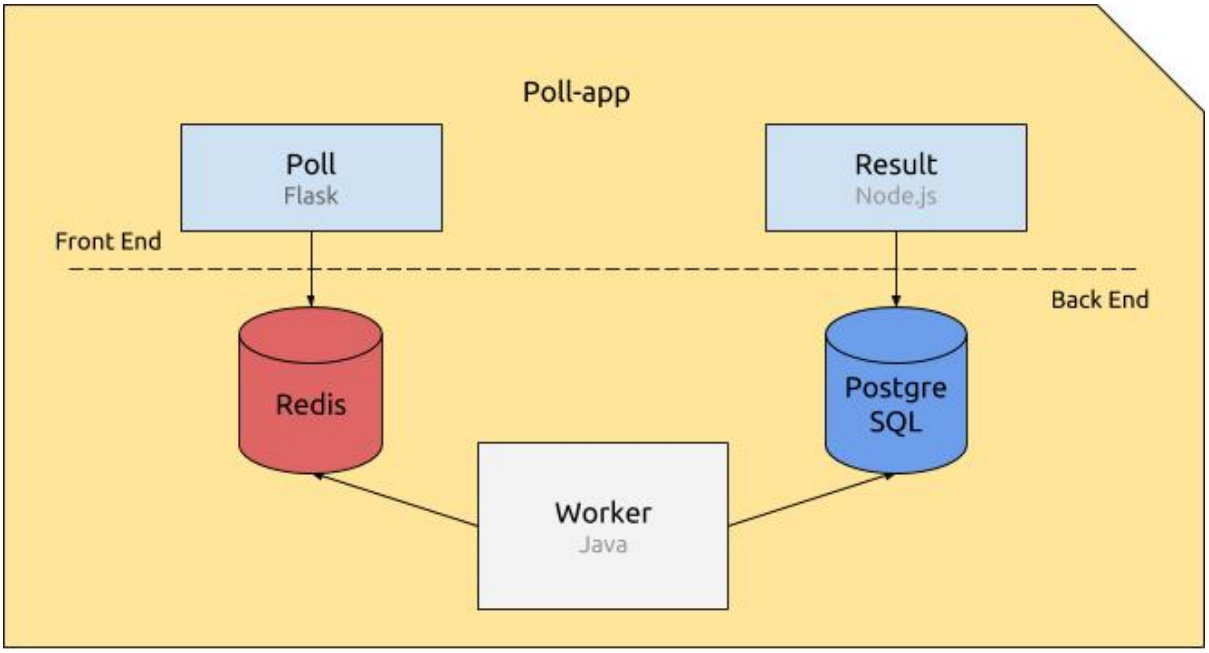
Like the brave sailor that Popeye is, containers can also confidently sail across the vast ocean of operating systems and configurations, being sure of working wherever they might end. As such, containers can be used on any host OS where Docker is installed.

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.

There are five elements constituting the application:
- ***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. ;)



The code of these applications 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.

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 a Node.js official 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` folder 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:3.5-jdk-8-alpine` and be named `builder`.
It must be used to build (natuurlijk) and package the Worker application using the following commands:
- `mvn dependency:resolve`, from within the folder containing `pom.xml`;
- then, `mvn package`, from within the folder containing the `src` folder.
It generates a file in the `target` folder named `worker-jar-with-dependencies.jar` (relative to your `WORKDIR`).

Second stage - run

The second stage must be based on `openjdk:8-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.

Databases addresses and credentials are hard-coded into applications. However, in order to make them ready for true deployment, you must use environment variables. Update the applications' code if necessary.

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. You must use version 3 of Docker Compose's syntax.

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.
- `redis`:
  - uses an existing official image of Redis;
  - opens port 6379.
- `worker`:
  - builds your `worker` image.
- `db`:
  - represents the database that will be used by the apps;
  - uses an existing official image of PostgreSQL;
  - has its database schema created during container first start.

- `result`:
  - builds your `result` image;
  - redirects port 5001 of the host to the port 80 of the container.

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`.

1 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.

Once your `docker-compose.yml` is complete, you should be able to run all the services and observe the votes you submitted to the `poll`'s webpage on `result`'s webpage.

Your containers must restart automatically when they stop unexpectedly.

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 applications, given to you on the intranet.