Containerization Technologies – TD 6

Step 0: GitLab

The connection is ok!

Step 1: Required architecture

My GitLab repository has the same architecture, except for the 'TD 6.pdf' file at the root directory of the repository (it's the report as asked for some groups).

Step 2: Create the Dockerfiles

/consumer

Containerization Technologies > TD 6 > consumer > Dockerfile > ...

```
1   ## step 1: build the application
2   # use the golang alpine image
3   FROM golang:alpine as builder
4
5   # set the working directory in the container
6   WORKDIR /app
7
8   # copy the current directory contents into the container at /app
9   COPY . .
10
11  # download the dependencies
12  RUN go mod download && go mod verify
13
14  # build the application
15  RUN go build -v -o /consumer
16
17  ## step 2: run the application
18  # empty container
19  FROM scratch
20
21  # copy the binary from the builder stage to the alpine image
22  COPY --from=builder /consumer .
23
24  # expose port 8080
25  EXPOSE 8080
26
27  # run the application
28  CMD ["./consumer"]
```

Here, step by step (and it's not going to be short):

- The first step is to build the application (only build, warning), so we use the official golang image with with the alpine tag as a builder;
- Then we set up the working directory and copy the files;
- We do a first RUN used to download the dependencies specified in the file 'go.sum';
- The second RUN:
    - Is setting up some variables (language, os);
    - 'go build' is the basic command to build a go application;
    - '-a' re-compile all the dependencies;
    - '-installsuffix cgo' desactivate the use of cgo;
    - And '-o /consumer' specify the name of the application and its path (here, our application file is named 'consumer' at the root of the container'.
- Now that we have our application file ONLY (and not all the golang image), we use an alpine image (something light) for our container;
- We set up the working directory (root of the container), copy our application file, expose the right port (8080) and we run the file.

/producer

Containerization Technologies > TD 6 > producer > Dockerfile > ...

```
1   ## step 1: build the application
2   # use the golang alpine image
3   FROM golang:alpine as builder
4
5   # set the working directory in the container
6   WORKDIR /app
7
8   # copy the current directory contents into the container at /app
9   COPY . .
10
11  # download the dependencies
12  RUN go mod download && go mod verify
13
14  # build the application
15  RUN go build -v -o /producer
16
17  ## step 2: run the application
18  # empty container
19  FROM scratch
20
21  # copy the binary from the builder stage to the alpine image
22  COPY --from=builder /producer .
23
24  # expose port 8080
25  EXPOSE 8080
26
27  # run the application
28  CMD ["./producer"]
```

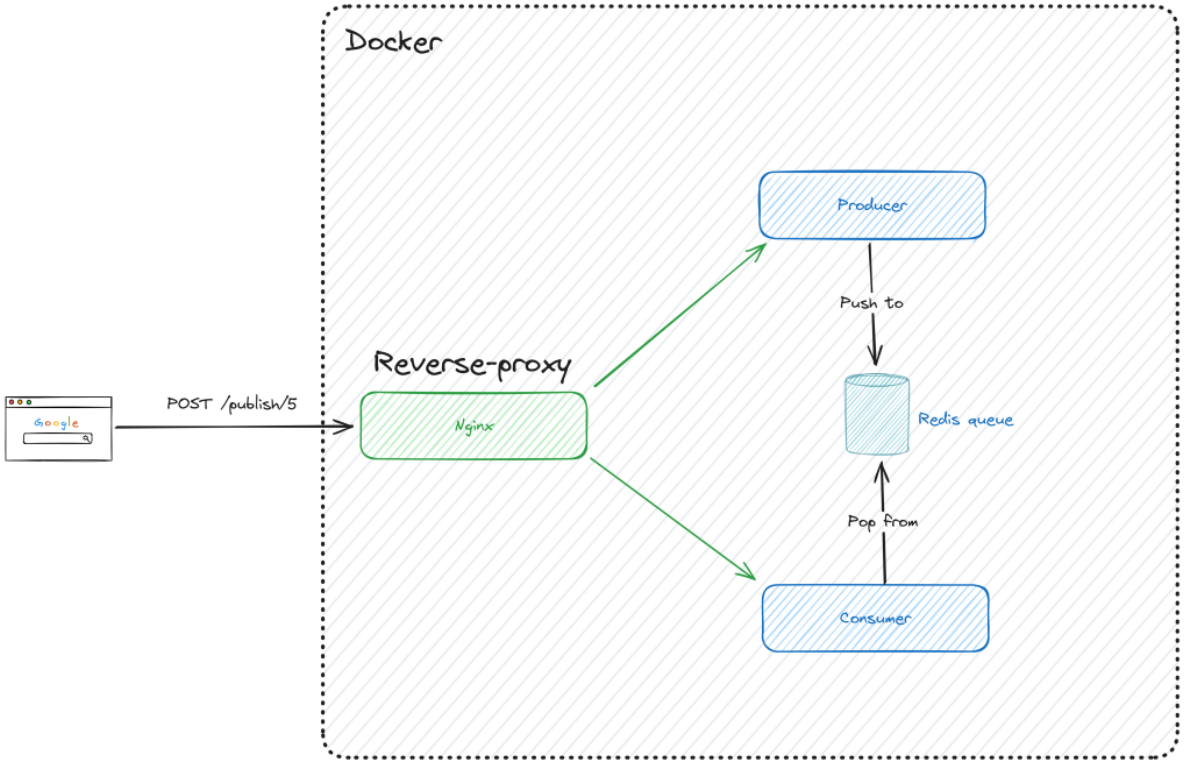
This is the same as seen before with the consumer Dockerfile.

/nginx

```
Containerization Technologies > TD 6 > nginx > Dockerfile > ...
1   # use the nginx image
2   FROM nginx:1.24-alpine
3
4   # copy the nginx configuration file
5   COPY nginx.conf /etc/nginx/nginx.conf
6
7   # expose port 8080
8   EXPOSE 8080
```

Here, I simply use an official nginx image with the alpine tag, copy the configuration file into the container and expose the right port. There's nothing more to do.

Step 3: Understand the target architecture



This architecture is helping us to understand which other container each container depends on. Here, we see that:

- The nginx container depends on the producer and consumer ones;
- The producer container depends on the redis one;
- The consumer container depends on the redis one.

Step 4: docker-compose

Here a quick view of my 'docker-compose.yaml' file:

Containerization Technologies > TD 6 > docker-compose.yaml

```yaml
version: 1.28.2 # specify docker-compose version

services:
  nginx:
    container_name: nginx
    build:
      context: ./nginx # specify the directory of the Dockerfile
    ports:
      - 8080:80
    depends_on: # depends on other services
      - consumer
      - producer

  redis:
    container_name: redis
    image: redis:alpine # specify image to use

  consumer:
    container_name: consumer
    build:
      context: ./consumer # specify the directory of the Dockerfile
    depends_on:
      - redis # depends on redis

  producer:
    container_name: producer
    build:
      context: ./producer # specify the directory of the Dockerfile
    depends_on:
      - redis # depends on redis
```

What is important to notice:

- The first line precises the version of docker-compose we're going to use;
- I added the name for each of my containers for better visbility;
- The context is used to indicate where is each Dockerfile;
- There're also some dependencies, following the target architecture seen before;
- And we create a volume for the redis container (the queue).

Step 5: Try it

Before trying it, I need to build and run with this command:

‘**docker-compose up -d**’

(-d is for detached).

Here’s the response I have when I try it:

```
C:\Users\Loeva>curl -I -H "Host: producer" -X POST http://localhost:8080/publish
HTTP/1.1 404 Not Found
Server: nginx/1.24.0
Date: Sun, 03 Mar 2024 19:35:17 GMT
Content-Type: text/html
Content-Length: 153
Connection: keep-alive
Strict-Transport-Security: max-age=31536000; includeSubDomains
```

Step 6: Fix the 404

404 means basically not found. There seems to be an issue with the server, and I have noticed that the server names where not the good ones (‘titi’ and ‘toto’), so I replaced these with the right ones (‘producer’ and ‘consumer’) in the ‘nginx.conf’ file:

```
# Producer
server {
    server_name producer; # FIXED

    set $upstream producer:8080; # FIXED

    location / {
        proxy_pass http://$upstream;
    }
}

# Consumer
server {
    server_name consumer; # FIXED

    set $upstream consumer:8080; # FIXED

    location / {
        proxy_pass http://$upstream;
    }
}
```

Here’s the response I have when I re-try it :

```
C:\Users\Loeva>curl -I -H "Host: producer" -X POST http://localhost:8080/publish
HTTP/1.1 500 Internal Server Error
Server: nginx/1.24.0
Date: Mon, 04 Mar 2024 18:35:05 GMT
Content-Type: text/plain; charset=utf-8
Content-Length: 80
Connection: keep-alive
X-Content-Type-Options: nosniff
Strict-Transport-Security: max-age=31536000; includeSubDomains
```

Step 7: Fix the 500

Here are the logs of the producer container:

```
2024-03-04 19:35:05 172.18.0.5 - - [04/Mar/2024:18:34:25 +0000] "POST /publish HTTP/1.0" 500 80
2024-03-04 19:31:09 2024/03/04 18:31:09 Listening on :8080...
2024-03-04 19:34:45 2024/03/04 18:34:45 Unable to connect to redis instance: ping: dial tcp: lookup redis-master: i/o timeout...
```

This line (‘tcp: lookup redis-master’) is indicating that it tries to connect to a server named ‘redis-master’, but mine is named ‘redis’ (and it must be). When looking the CLI flags of the producer, I noticed that the ‘redisAddr’ is using the name ‘redis-master’, so I replaced it by ‘redis’:

```
20  // ============================================================================
21  // CLI flags
22  // ============================================================================
23
24  var (
25      httpAddr   = flag.String("http", ":8080", "Address to listen for requests on")
26      redisAddr  = flag.String("redis-server", "redis:6379", "Redis server to publish messages to")
27      redisQueue = flag.String("redis-queue", "esilv", "Redis queue to publish messages to")
28  )
```

Here’s the response I have when I re-try it:

```
C:\Users\Loeva>curl -I -H "Host: producer" -X POST http://localhost:8080/publish
HTTP/1.1 200 OK
Server: nginx/1.24.0
Date: Mon, 04 Mar 2024 19:16:03 GMT
Content-Type: text/plain; charset=utf-8
Content-Length: 23
Connection: keep-alive
Strict-Transport-Security: max-age=31536000; includeSubDomains
```

Step 8: Fix messages not being consumed

Now, we have a good response with the curl command. Let’s see our queue:

```
blxucreep@Blxucreep:/mnt/c/Users/Loeva$ docker exec -it redis redis-cli KEYS '*'
1) "esilv"
```

We check the length of our key (‘esilv’):

```
blxucreep@Blxucreep:/mnt/c/Users/Loeva$ docker exec -it redis redis-cli LLEN esilv
(integer) 1
```

I published one time, so it’s logical to have one here. But the problem is that the consumer is not consuming our messages. Let’s check why it is writen ‘beluga’ here, it seems to be a good start to investigate:

```
2024-03-04 23:29:35 2024/03/04 22:29:35 Starting to comsume messages in beluga...
2024-03-04 23:29:35 2024/03/04 22:29:35 Listening on :8080...
```

When I check the ‘main.go’ file of the consumer, I see that there’s a mistake in the variables, our redis-queue was previously named ‘esilv’ in the producer ‘main.go’ file, but here it’s named ‘beluga’, so let’s replace it by ‘esilv’:

```
// ================================================================================
// CLI flags
// ================================================================================

var (
    cpuBurn    = flag.Bool("cpu-burn", false, "Whether to use CPU while processing messages")
    httpAddr   = flag.String("http", ":8080", "Address to listen for requests on")
    redisAddr  = flag.String("redis-server", "redis:6379", "Redis server to consume messages from")
    redisQueue = flag.String("redis-queue", "esilv", "Redis queue to consume messages from")
    logToFile  = flag.Bool("log-to-file", false, "Whether to log everything in file instead of stdout")
    timePerMsg = flag.Duration("per-msg", time.Second, "The amount of time the consumer spends on each message")
)
```

When re-trying the publish route, we can see that in our consumer container logs the message has been successfully consumed:

```
2024-03-04 23:36:52 2024/03/04 22:36:52 Starting to comsume messages in esilv...
2024-03-04 23:36:52 2024/03/04 22:36:52 Listening on :8080...
2024-03-04 23:37:05 2024/03/04 22:37:05 Received a message: "Hello :) ! 👋
2024-03-04 23:37:05 2024/03/04 22:37:05 Processing message...
2024-03-04 23:37:06 2024/03/04 22:37:06 Message processed.
```

Step 9: Fix the random crash

It appears that there’s a crash after processing ~10 messages only:

```
2024/03/05 02:03:15 Received a message: "Hello :) ! 👋".
2024/03/05 02:03:15 Processing message...
2024/03/05 02:03:16 Message processed.
2024/03/05 02:03:16 Received a message: "Hello :) ! 👋".
2024/03/05 02:03:16 Processing message...
2024/03/05 02:03:16 💥
blxucreep@Blxucreep:/mnt/c/Users/Loeva$
```

Let’s have a look at the ‘main.go’ file. More precisely the functiun to consume messages. We don’t want to play some russian roulette :/, let’s delete this condition:

```
// Let's play some russian roulette!
if rand.Intn(19) == 0 {
    log.Println("💥")
    os.Exit(1)
}
```

Now, we can consume all the messages we want!

Step 10: Git submission

Everything is submitted on GitLab. My commit is also signed with a GPG key.

The link: https://gitlab.sre.paris/td6/td6-loevan-le-quernec.