Docker Compose Basics

Autor: Alberto Herrera Poza
Github: https://github.com/AlbertoIHP

Introducción

El presente documento tiene la finalidad de explorar otras funcionalidades del núcleo de Docker, como por ejemplo, docker-compose que nos permite iterar sobre un stack completo en una aplicación de software, realizar distintas configuraciones Dockerfile entre sus componentes, y levantar servicios, dependencias, entre otras características.

1. Docker-compose

Cuando es necesario desplegar una aplicación de software, que está **compuesta** por un stack completo de tecnologías independientes, entonces es útil la utilización de Docker-Compose, este, permite el despliegue sistemático y parametrizado del stack en cuestión, con una sintaxis legible y fácil de implementar. La única limitante de docker-compose, es que el despliegue del todo este stack será dentro del mismo host.

En primer lugar, imaginemos que tenemos una aplicación en Spring Boot, que además está conectada con MySQL en el Backend, por otro lado, en el Frontend una aplicación React.

2. *Springboot Dockerfile*

El Dockerfile para el servidor Spring Boot es.

```
#### Stage 1: Imagen tomada desde OPENJDK asociada al alias 'build'
FROM openjdk:8-jdk-alpine as build

# Luego definimos como directorio de trabajo 'app'
WORKDIR /app

# Se copian los ejecutables de maven a la imagen en el directorio de trabajo 'app'
COPY mvnw .
COPY .mvn .mvn

# Copiamos el fichero de dependencias de maven a nuestro directorio de trabajo 'app'
COPY pom.xml .

# Se instalan las dependencias para que queden disponibles aún desconectados
# Cabe destacar que las dependencias están en caché a menos que el fichero de
dependencias cambie (pom.xml)
RUN ./mvnw dependency:go-offline -B

# Copiamos el directorio con el código fuente a nuestro directorio de trabajo
COPY src src

# Package the application
RUN ./mvnw package -DskipTests
RUN mkdir -p target/dependency && (cd target/dependency; jar -xf ../*.jar)

#### Stage 2: A minimal docker image with command to run the app
FROM openjdk:8-jre-alpine

ARG DEPENDENCY=/app/target/dependency

# Copy project dependencies from the build stage
COPY --from=build ${DEPENDENCY}/BOOT-INF/lib /app/lib
COPY --from=build ${DEPENDENCY}/META-INF /app/META-INF
COPY --from=build ${DEPENDENCY}/BOOT-INF/classes /app

ENTRYPOINT ["java","-cp","app:app/lib/*","com.example.polls.PollsApplication"]
```

En primera instancia se compila la aplicación, y luego se embebe en un micro servidor.

3. *React Dockerfile*

Por otro lado, el Dockerfile de la aplicación en React es.

```
#### Stage 1: Build the react application
FROM node:12.4.0-alpine as build

# Configure the main working directory inside the docker image.
# This is the base directory used in any further RUN, COPY, and ENTRYPOINT
# commands.
WORKDIR /app

# Copy the package.json as well as the package-lock.json and install
# the dependencies. This is a separate step so the dependencies
# will be cached unless changes to one of those two files
# are made.
COPY package.json package-lock.json ./
RUN npm install

# Copy the main application
COPY . ./

# Arguments
ARG REACT_APP_API_BASE_URL
ENV REACT_APP_API_BASE_URL=${REACT_APP_API_BASE_URL}

# Build the application
RUN npm run build

#### Stage 2: Serve the React application from Nginx
FROM nginx:1.17.0-alpine

# Copy the react build from Stage 1
COPY --from=build /app/build /var/www

# Copy our custom nginx config
COPY nginx.conf /etc/nginx/nginx.conf

# Expose port 80 to the Docker host, so we can access it
# from the outside.
EXPOSE 80

ENTRYPOINT ["nginx","-g","daemon off;"]
```

Al igual que con el servidor, en primera instancia se compila la aplicación, y luego se embebe dentro de un micro contenedor con nginx.

4. *Fichero de configuración Docker-Compose*

Para realizar el despliegue de la aplicación con stack Spring boot + MySQL en el Backend, y React + Nginx en el Frontend, es necesario crear un archivo docker-compose.yml, el cual tendrá la configuración necesaria para desplegar la aplicación.

A grandes rasgos, es posible identificar 3 servicios en nuestra aplicación de software, el servidor, el cliente y la base de datos.

```
# Docker Compose file Reference (https://docs.docker.com/compose/compose-file/)

version: '3.7'

# Define services
services:
  # App backend service
  app-server:
    # Configuration for building the docker image for the backend service
    build:
      context: polling-app-server # Use an image built from the specified dockerfile in
the `polling-app-server` directory.
      dockerfile: Dockerfile
    ports:
      - "8080:8080" # Forward the exposed port 8080 on the container to port 8080 on the
host machine
    restart: always
    depends_on:
      - db # This service depends on mysql. Start that first.
    environment: # Pass environment variables to the service
      SPRING_DATASOURCE_URL:
jdbc:mysql://db:3306/polls?useSSL=false&serverTimezone=UTC&useLegacyDatetimeCode=false
      SPRING_DATASOURCE_USERNAME: callicoder
      SPRING_DATASOURCE_PASSWORD: callicoder
    networks: # Networks to join (Services on the same network can communicate with each
other using their name)
      - backend
      - frontend
```

```
  # Frontend Service
  app-client:
    build:
      context: polling-app-client # Use an image built from the specified dockerfile in
the `polling-app-client` directory.
      dockerfile: Dockerfile
      args:
        REACT_APP_API_BASE_URL: http://127.0.0.1:8080/api
    ports:
      - "9090:80" # Map the exposed port 80 on the container to port 9090 on the host
machine
    restart: always
    depends_on:
      - app-server
    networks:
      - frontend

  # Database Service (Mysql)
  db:
    image: mysql:5.7
    ports:
      - "3306:3306"
    restart: always
    environment:
      MYSQL_DATABASE: polls
      MYSQL_USER: callicoder
      MYSQL_PASSWORD: callicoder
      MYSQL_ROOT_PASSWORD: root
    volumes:
      - db-data:/var/lib/mysql
    networks:
      - backend

# Volumes
volumes:
  db-data:

# Networks to be created to facilitate communication between containers
networks:
  backend:
  frontend:
```

5. *Corriendo y deteniendo Docker-Compose*

Finalmente, para correr docker-compose.
`docker-compose up`

```
docker-compose up
```

Y para darlo de baja.

```
docker-compose down
```

6. *Docker-Compose en el Stack Ruby-Rails-Puma-PostgreSQL-NginX*

En el caso de este stack, las cosas no cambian mucho, excepto por la configuración de NginX. Para la aplicación en ruby, el fichero Dockerfile es.

```
FROM ruby:2.3.1
RUN apt-get update -qq && apt-get install -y build-essential libpq-dev
nodejs
# Set an environment variable where the Rails app is installed to inside
of Docker image
ENV RAILS_ROOT /var/www/app_name
RUN mkdir -p $RAILS_ROOT
# Set working directory
WORKDIR $RAILS_ROOT
# Setting env up
ENV RAILS_ENV='production'
ENV RACK_ENV='production'
# Adding gems
COPY Gemfile Gemfile
COPY Gemfile.lock Gemfile.lock
RUN bundle install --jobs 20 --retry 5 --without development test
# Adding project files
COPY . .
RUN bundle exec rake assets:precompile
EXPOSE 3000
CMD ["bundle", "exec", "puma", "-C", "config/puma.rb"]
```

Por otro lado, el Dockerfile de NginX es.

```
# Base image
FROM nginx
# Install dependencies
RUN apt-get update -qq && apt-get -y install apache2-utils
# establish where Nginx should look for files
ENV RAILS_ROOT /var/www/app_name
# Set our working directory inside the image
WORKDIR $RAILS_ROOT
# create log directory
RUN mkdir log
# copy over static assets
COPY public public/
# Copy Nginx config template
COPY docker/web/nginx.conf /tmp/docker.nginx
# substitute variable references in the Nginx config template for real
values from the environment
# put the final config in its place
RUN envsubst '$RAILS_ROOT' < /tmp/docker.nginx >
/etc/nginx/conf.d/default.conf
EXPOSE 80
# Use the "exec" form of CMD so Nginx shuts down gracefully on SIGTERM
(i.e. `docker stop`)
CMD [ "nginx", "-g", "daemon off;" ]
```

y su archivo de configuración respectivo.

```
upstream rails_app {
   server app:3000;
}
server {
   # define your domain
   server_name www.example.com;
   # define the public application root
   root   $RAILS_ROOT/public;
   index  index.html;
   # define where Nginx should write its logs
   access_log $RAILS_ROOT/log/nginx.access.log;
   error_log $RAILS_ROOT/log/nginx.error.log;
   # deny requests for files that should never be accessed
   location ~ /\. {
      deny all;
   }
   location ~* ^.+\.(rb|log)$ {
      deny all;
   }
   # serve static (compiled) assets directly if they exist (for rails
production)
   location ~ ^/(assets|images|javascripts|stylesheets|swfs|system)/   {
      try_files $uri @rails;
      access_log off;
      gzip_static on;
      # to serve pre-gzipped version
      expires max;
      add_header Cache-Control public;
      add_header Last-Modified "";
      add_header ETag "";
      break;
   }
   # send non-static file requests to the app server
   location / {
      try_files $uri @rails;
   }
   location @rails {
      proxy_set_header  X-Real-IP  $remote_addr;
      proxy_set_header  X-Forwarded-For $proxy_add_x_forwarded_for;
      proxy_set_header Host $http_host;
      proxy_redirect off;
      proxy_pass http://rails_app;
```

```
    }
}
```

Por último, el fichero Docker-Compose es.

```
version: '3'
volumes:
  postgres_data: {}
services:
  app:
    build:
      context: .
      dockerfile: ./docker/app/DockerFile
    depends_on:
      - db
  db:
    image: postgres
    volumes:
      - postgres_data:/var/lib/postgresql/data
  web:
    build:
      context: .
      dockerfile: ./docker/web/DockerFile
    depends_on:
      - app
    ports:
      - 80:80
```

Para que la base de datos contenida en el contenedor declarado como servicio, funcione y sea consumida por la aplicación de Ruby, es necesario cambiar el fichero en config/database.yml.

```
default: &default
  adapter: postgresql
  encoding: unicode
  username: postgres
  password:
  pool: 5
  host: db
production:
  <<: *default
  database: app_name_production
```

En este caso, es necesario primero compilar los servicios declarados en nuestro Docker-Compose.

```
docker-compose build
```

Para crear la base de datos.

```
docker-compose run app rake db:create RAILS_ENV=production
```

Para poblar la base de datos.

```
docker-compose run app rake db:migrate db:seed RAILS_ENV=production
```

Finalmente para levantar todos los servicios declarados.

```
docker-compose up -d
```