Rapport d'AL

Réalisé par Olivier Peurichard & Dylan Béhêtre
durant l'année universitaire 2016-2017

Sommaire

Contexte

Dans le cadre du module d'Architecture logicielle, de notre formation d'ingénieur à l'Ecole Supérieur d'Ingénieur de Rennes, il nous a été demandé de réaliser le déploiement de l'application RSS Reader, via l'utilisation de briques architecturales fourni par Netflix OSS.

Ce rapport a pour objectif d'expliquer l'architecture que nous avons mis en place pour le déploiement de cette application.

L'architecture de notre projet

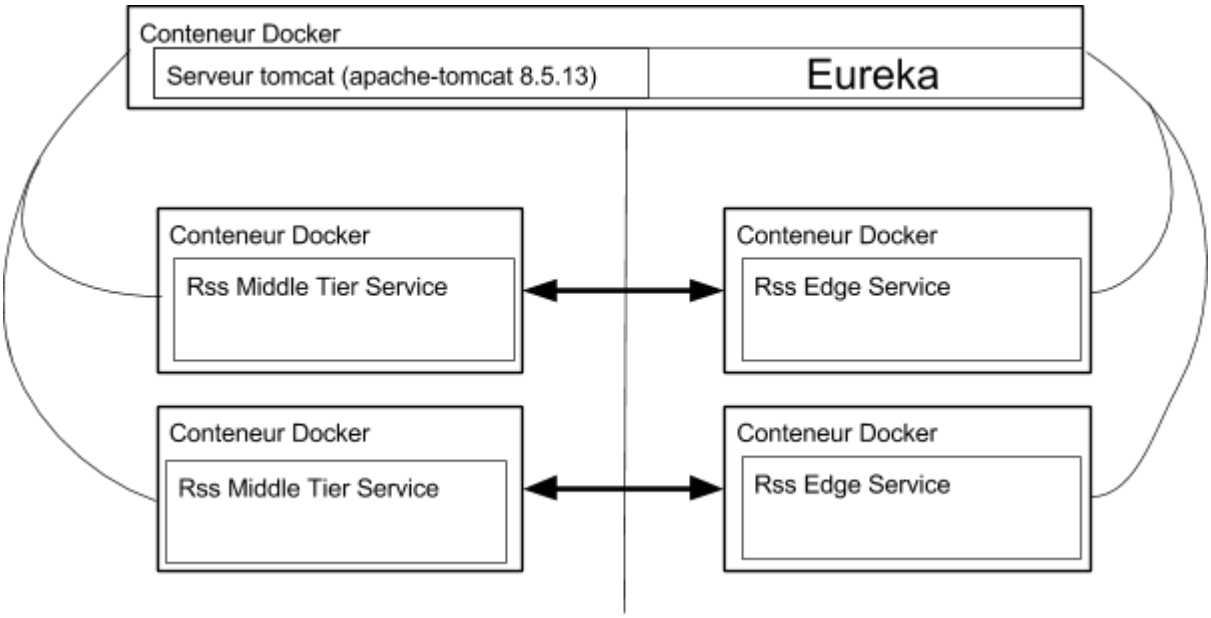

Figure 1 : Schéma de notre architecture

Comme vous pouvez le voir sur la figure 1, notre architecture se décompose en trois type de conteneur docker :

- ➔ Un conteneur docker "Eureka" permettant de faire tourner l'application Eureka via un serveur tomcat.
- ➔ Un conteneur docker "Rss_Middle_Tier" permettant de faire tourner le service "Rss Middle Tier Service".
- ➔ Un conteneur docker "Rss_Edge" permettant de faire tourner le service "Rss Edge Service".

```
ENV EUREKA_VERSION 1.1.158

RUN apt-get update && apt-get install -y wget git
WORKDIR /
RUN wget http://repo1.maven.org/maven2/com/netflix/eureka/eureka-server/${EUREKA_VERSION}/eureka-server-${EUREKA_VERSION}.war
RUN cp /eureka-server-${EUREKA_VERSION}.war /usr/local/tomcat/webapps/eureka.war

```

Figure 2 : Dockerfile du conteneur docker "Eureka"

Comme vous pouvez le voir dans la figure 2, nous avons utilisé comme comme base de notre image, l'image de tomcat 7 via la commande "From tomcat:7".

Cela nous a permis d'obtenir la structure de base d'un serveur tomcat et donc d'obtenir une version fonctionnelle sans trop de problème.

Nous avons ensuite mis à jour tous ce qui était installé sur le serveur via la commande "RUN apt-get update" puis nous avons installé wget via la suite de la commande précédente "&& apt-get install -y wget".

Nous mettons alors à jour le workspace de travail via la commande "WORKDIR /" puis nous récupérons l'application Eureka pour ensuite la déplacer au bonne endroit sur notre serveur tomcat via les commande "RUN wget (...).war" et "RUN cp (...).war".

```
FROM java:7

RUN apt-get update && apt-get install -y git libgradle-core-java

RUN git clone https://github.com/Netflix/recipes-rss.git

COPY recipes-rss/rss-middletier/src/main/resources/middletier.properties /recipes-rss/rss-middletier/src/main/resources/middletier.properties

WORKDIR /recipes-rss
ENV APP_ENV=dev
RUN ./gradlew clean build

CMD ["java", "-jar", "rss-middletier/build/libs/rss-middletier-0.1.0-SNAPSHOT.jar"]
```

*Figure 3 : Dockerfile du conteneur docker "*Rss_Middle_Tier*"*

Pour le Dockerfile du conteneur docker "Rss_Middle_Tier" (figure 3), nous récupérons comme base de notre image, l'image de java 7. Cela nous permet d'avoir toute la structure java nécessaire pour faire tourner notre service.

Nous mettons tout à jour puis nous installons git et gradle pour java.

Nous récupérons ensuite via un git clone le service "Rss Middle Tier Service". Puis nous remplaçons le fichier de configuration "middletier.properties" par le notre afin de corriger un certains nombre d'erreurs et pour configurer le service de manière à correspondre à notre architecture (notamment l'URL de notre container EUREKA).

Nous mettons ensuite l'emplacement de notre espace de travail à jour via la commande "WORKDIR". Puis nous spécifions la variable d'environnement "APP_ENV" via la commande

“ENV APP_ENV=dev”, avant d'exécuter un clean build gradle via la commande “RUN ./gradlew clean build”.

Nous finissons ensuite par lancer le .jar de notre service via la commande “CMD ["java", "-jar", "rss-middletier/build/libs/rss-middletier-0.1.0-SNAPSHOT.jar"]”.

```
FROM java:7

RUN apt-get update && apt-get install -y git wget libgradle-core-java

RUN git clone https://github.com/Netflix/recipes-rss.git
COPY recipes-rss/rss-edge/src/main/resources/edge.properties /recipes-rss/rss-edge/src/main/resources/edge.properties

WORKDIR /recipes-rss
ENV APP_ENV=dev
RUN ./gradlew clean build

WORKDIR /recipes-rss
CMD ["java", "-jar", "rss-edge/build/libs/rss-edge-0.1.0-SNAPSHOT.jar"]
```

*Figure 4 : Dockerfile du conteneur docker “*Rss_Edge*”*

Pour notre dernier Dockerfile (figure 4), destiné à créer une image pour le service “Rss Edge Service”, nous avons, comme pour le Dockerfile précédant, pris pour base l’image “java 7”.

Nous mettons également, comme pour le précédent Dockerfile, tout à jour et nous installons git et gradle.

Nous récupérons ensuite via un git clone le service “Rss Edge Service” et nous écrasons le fichier de configuration “edge.properties” par le notre afin d’adapter le service à notre architecture (notamment l’URL de notre container EUREKA).

Nous mettons, comme précédemment, l’espace de travail à jour puis nous spécifions la variable d’environnement “APP_ENV” à “dev” avant de lancer un clean build gradle.

Nous lançons finalement le .jar du service via la commande “CMD ["java", "-jar", "rss-edge/build/libs/rss-edge-0.1.0-SNAPSHOT.jar"]”.

```
version: "3"
services:

  eureka:
    image: myeureka
    hostname: eureka
    container_name: eureka
    ports: ['8080:8080']
    privileged: true
    network_mode: "bridge"

  middletier:
    image: mymiddletier
    depends_on: ['eureka']
    container_name: middletier
    hostname: middletier
    ports: ['9190:9191']
    external_links: ['eureka:eureka']
    network_mode: "bridge"

  middletier1:
    image: mymiddletier
    depends_on: ['eureka']
    container_name: middletier1
    hostname: middletier1
    ports: ['9191:9191']
    external_links: ['eureka:eureka']
    network_mode: "bridge"

  edge:
    image: myedge
    depends_on: ['eureka','middletier']
    ports: ['9091:9090']
    external_links: ['eureka:eureka','middletier:middletier']
    network_mode: "bridge"

  edge1:
    image: myedge
    depends_on: ['eureka','middletier1']
    ports: ['9090:9090']
    external_links: ['eureka:eureka','middletier1:middletier1']
    network_mode: "bridge"
```

Figure 5 : le docker-compose de notre architecture

Notre architecture consiste à lier, via un docker-compose (figure 5), un conteneur docker “Eureka” avec deux conteneur “Rss_Middle_Tier” et deux conteneur “Rss_Edge”.

Pour cela, nous avons créé un fichier “.yml” qui recense les différents services que nous souhaitons lancer. On crée donc un service par container voulu, en précisant si besoin une dépendance vers un autre container (depends_on).

Si cela n’est pas fait, les containers Edge vont tenter de communiquer avec un container Eureka qui peut ne pas encore être fonctionnel, et une erreur sera renvoyée, interrompant tout le processus.

Ensuite, et ce pour chacun des services, on précise diverses informations nécessaires au bon fonctionnement tels que les redirections de ports host→ container, ou encore les liens entre les différents containers afin que ceux-ci puissent communiquer entre eux.

Enfin, il faut préciser, pour chaque service, le réseau sur lequel il va se connecter. En effet, Docker met à disposition un système de réseaux (un peu comme dans la vraie vie) qui permet de limiter les communications d’un container au réseau auquel il est rattaché.
Docker-compose, par défaut, n’ajoute pas les services au réseau par défaut “bridge”. Il a donc fallu le préciser.

Les forces de notre architecture

Notre architecture se compose de trois forces.

Notre première force réside dans la dockerisation de celle-ci.

Nous pouvons ainsi créer des instance de chacun de nos services rapidement, en mettant simplement à jour notre docker-compose.

Cela amène à notre deuxième force, notre facilité à pouvoir passer à l'échelle. En effet, nous pouvons adapter le nombre de service lancé aux besoins de nos utilisateurs.

Notre troisième force réside dans notre résistance relative aux pannes. En effet, nous disposons de plusieurs instances de chacun de nos services. Dans ce contexte, si une de nos instances tombe en panne, une autre pourra tout à fait prendre le relais.

Les faiblesses de notre architecture

Notre architectures dispose de trois faiblesses.

La première de ces faiblesses est la résistance à la panne sur notre instance d’Eureka. En effet si notre unique serveur tomcat hébergeant Eureka tombe en panne, notre architecture ne fonctionne plus.

La deuxième faiblesse de notre application est la résistance à la panne de toutes nos instances de services. Ainsi, si toutes nos instances existantes arrêtent de fonctionner, aucun système ne permettra d’en relancer de nouvelles. Notre application ne fonctionnera plus.

La troisième faiblesse de notre application réside dans notre incapacité actuelle de répliquer les données d’un middle tier à un autre. Ainsi, si un middle tier tombe, les informations qu’il contenait disparaissent.