

Joceran FICHOU--MEUNIER
Allan BONHOMME
Amir ROMDHANE

Application RSS Reader Architecture Logicielle

Compte-Rendu

Introduction

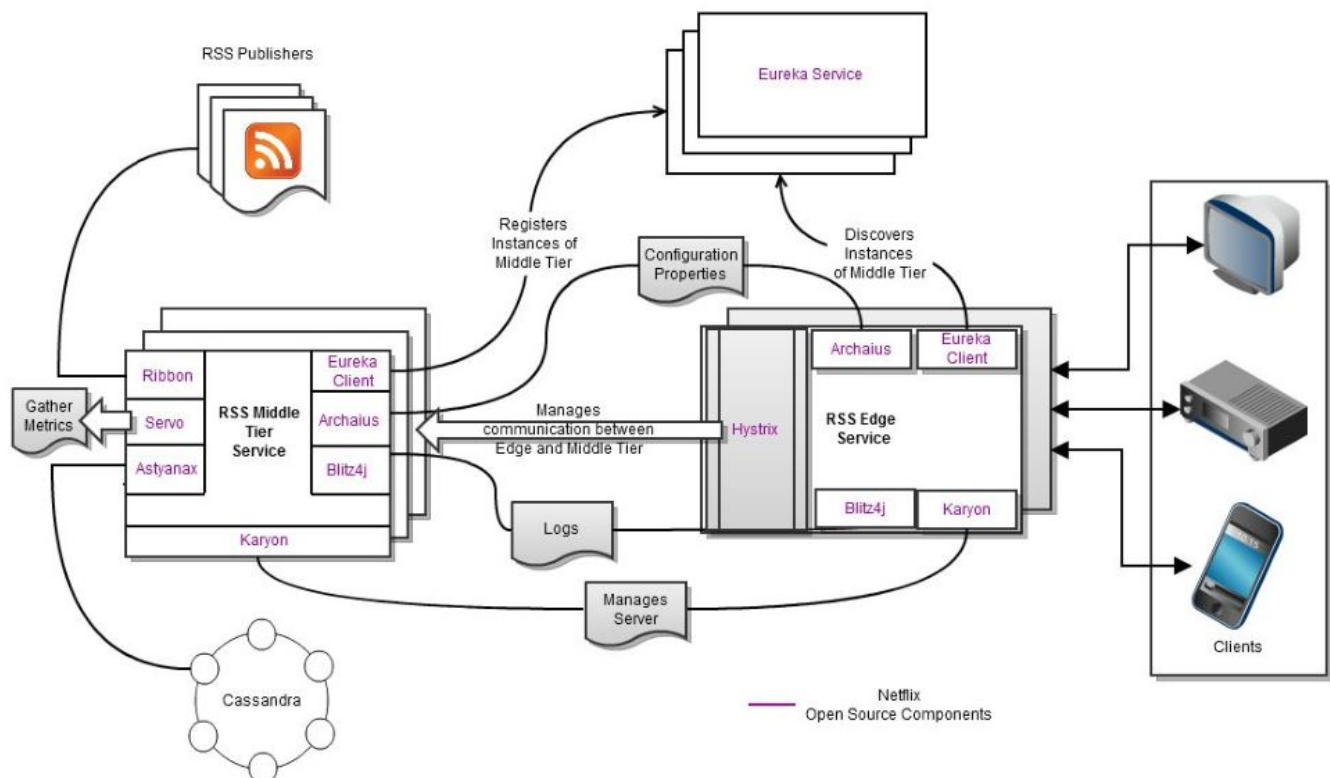
Le résultat de ce TP est disponible sur : <https://github.com/Joceran/RSSReader>

Le but de cette application est de créer un agrégateur de news depuis différents flux dit "RSS" de diverses sources.

L'application possède une architecture logicielle composée de 3 micro-services, tous les trois développés par l'entreprise Netflix.

- **Eureka** : Ce micro-service permet de faire de la découverte de service. Il permettra ici à RSS-Edge de trouver RSS-Middleware.
- **RSS-Middleware** : Ce micro-service est celui qui agrège les différents flux RSS.
- **RSS-Edge** : Ce micro-service agit comme le front-end de l'application, il met en forme les flux de news et fait l'interface auprès des différents clients connectés à l'application

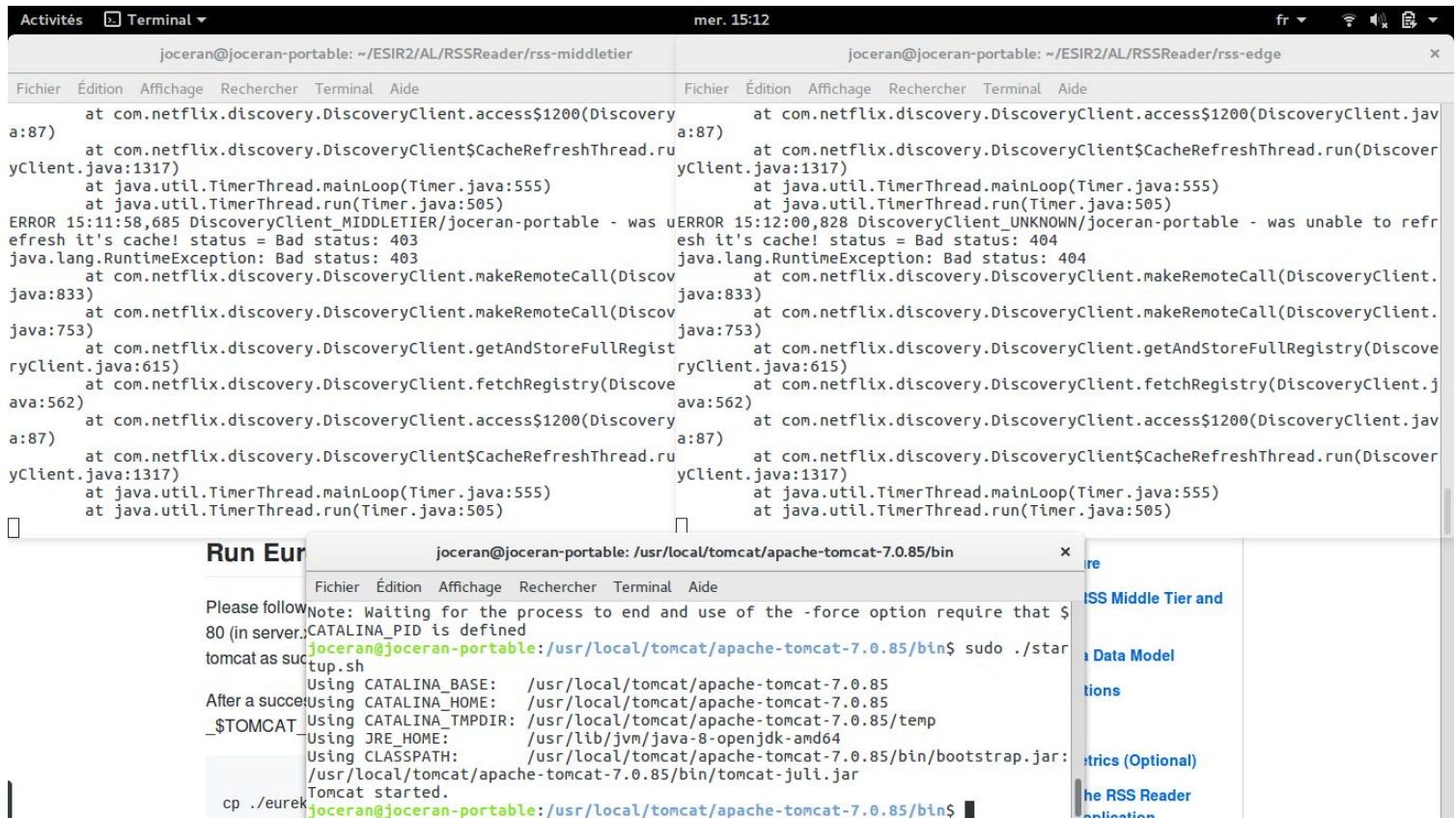
Voici à quoi ressemble l'architecture sous forme de schéma.



Il est possible de connecter l'application à une base de donnée, mais nous utiliserons ici qu'un stockage en mémoire sur le RSS-Middleware.

Mise en place de l'application

Les 3 services à lancer sont tous disponibles sur le Github de Netflix. Après les avoirs téléchargés et build les services, nous les lançons dans des terminaux séparés. L'application est donc lancée en local.



On obtient donc cette interface, on peut rajouter un flux RSS dans la barre en haut et il y a donc les news de cette source s'affichant dans la page.

Netflix OSS RSS Reader

Enter the feed Url

Add

Le Monde.fr - Actualités et Infos en France et dans le monde

Suivez Emmanuel Macron face au Congrès américain en direct

Chômage : « Les effets des réformes ne se font pas encore clairement sentir »

Carrefour : accord trouvé sur le plan social et le plan de départs volontaires

Peter Madsen condamné à la prison à vie pour le meurtre de la journaliste suédoise Kim Wall

Elisabeth Borne : « Je suis consciente de l'inquiétude des cheminots »

Netflix Inc. 2013

Cela marche bien, mais si on veut déployer l'application facilement sur n'importe quel environnement, il nous faut donc un moyen simple de tout installer rapidement. C'est ici qu'intervient Docker.

Mise en conteneurs des micro-services

Nous allons donc mettre les fichiers jar et war obtenus dans des conteneurs Docker. Pour ce faire, cela se passe par la création d'un fichier Dockerfile pour chaque micro service qui donne la marche à suivre pour mettre en place et lancer l'application au moment voulu.

Un Dockerfile est une série de commandes permettant de travailler à l'intérieur d'un conteneur. Tout d'abord avec une commande "FROM" on peut déclarer des dépendances, comme un OS, tel que Ubuntu, ou une application comme Java ou Tomcat. Ensuite on donne un répertoire de travail au sein du conteneur avec un mot clé "WORKDIR". Ce conteneur n'est pas visible dans le système de fichiers de l'utilisateur, le Dockerfile est donc l'unique moyen (que nous connaissons du moins à l'heure actuelle) de travailler à l'intérieur d'un conteneur. On ajoute les fichiers dans le conteneur avec ADD et si on a des commandes à exécuter à l'intérieur, EXPOSE permet d'ouvrir les ports de l'image vers la machine cible, le mot clé RUN est utile pour exécuter des commandes shell pendant le build, et pour finir, on peut utiliser CMD pour exécuter une commande lors du déploiement du conteneur.

Pour créer les conteneurs de notre application, il y a deux méthodes principales possibles. Une première méthode consiste à télécharger directement dans le conteneur les sources, les compiler pour utiliser le jar qui nous intéresse. Elle possède l'avantage qu'on peut juste avoir un Dockerfile à télécharger afin de lancer le micro-service, mais si plusieurs micro-services utilisent les mêmes sources mais pas les mêmes fichiers, cela peut prendre beaucoup de place sur la machine. Par exemple cette méthode me donnait une image de 1Gb environ rien que pour le rss-Middletier et donc 1Gb en plus pour edge. Ce n'est que quelques kilo-octets de fichiers à télécharger, mais ensuite cela devient lourd en créant les images Docker.

La deuxième technique, celle que nous avons utilisée pour ce TP est d'utiliser les fichiers déjà compilés dans la version précédente avec quelques modifications dans les sources au préalable. Notamment l'adresse d'eureka qui ne sera plus réellement localhost lors du déploiement mais une autre adresse IP que Docker lui attribue, et nous devons communiquer cette information à RSS-Middletier et RSS-Edge. Si au final il faudra télécharger environ 50 Mo à télécharger (les fichiers jar notamment), on gagne en place au niveau des images Docker.

Afin de séparer nos microservices, nous les avons séparés dans cette structure d'application, afin que si il y avait des fichiers à ajouter aux conteneurs cela soit plus facile de les mettre et aussi afin d'appeler plus facilement les différents Dockerfile. (les fichiers marqués de old sont ceux qui n'ont pas le changement localhost -> eureka)

```
joceran@joceran-portable:~/ESIR2/AL/RSSReader$ tree
.
├── docker-compose.yml
├── eureka
│   ├── Dockerfile
│   ├── eureka.war
│   ├── server.xml
│   └── tomcat-users.xml
├── rss-edge
│   ├── Dockerfile
│   ├── rss-edge-0.1.0-SNAPSHOT.jar
│   └── rss-edge-0.1.0-SNAPSHOT_old.jar
└── rss-middletier
    ├── Dockerfile
    ├── rss-middletier-0.1.0-SNAPSHOT.jar
    └── rss-middletier-0.1.0-SNAPSHOT_old.jar

3 directories, 11 files
```

Mes fichiers Docker sont donc les suivants :

Dockerfile pour rss-middletier

```
FROM java:8
#Définition de la variable environnement de dev, afin que l'application
sélectionne certaines options.
ENV APP_ENV=dev
#start jar
WORKDIR /
ADD rss-middletier-0.1.0-SNAPSHOT.jar .
EXPOSE 9090 9092
CMD ["java", "-jar", "rss-middletier-0.1.0-SNAPSHOT.jar"]
```

Dockerfile pour rss-edge

```
FROM java:8
#Définition de la variable environnement de dev, afin que l'application
sélectionne certaines options.
ENV APP_ENV=dev
#start jar
WORKDIR /
ADD rss-edge-0.1.0-SNAPSHOT.jar .
EXPOSE 9090 9092
CMD ["java", "-jar", "rss-edge-0.1.0-SNAPSHOT.jar"]
```

Dockerfile pour eureka

```
FROM tomcat:7.0.85-jre8
ADD eureka.war /usr/local/tomcat/webapps/
ADD server.xml /usr/local/tomcat/conf/
ADD tomcat-users.xml /usr/local/tomcat/conf/
EXPOSE 80
CMD ["catalina.sh", "run"]
```

Pour build les images ont lancé la commande

```
sudo docker build -t <nom du service>:latest <nom du service à build>.
```

L'option -t permet de tagguer une image afin qu'on utilise un nom usuel au lieu de l'ID qui prend une forme impossible à retenir.

Une fois ces Dockerfiles créés et buildés nous pouvons donc les run de cette façon et avoir une application fonctionnelle :

```
#Eureka
sudo docker run -d -p 80:80 --name eureka --privileged=true eureka:latest
#Middletier
sudo docker run -d --link eureka:localhost --name middletier --hostname
middletier rss-middletier-docker
#Edge
sudo docker run -p 9090:9090 --name edge --link eureka:localhost --link
middletier:middletier rss-edge-docker
```

L'option -d permet de passer l'exécution en arrière plan mais nous avons choisi de laisser edge en premier-plan pour voir les erreurs potentielles à l'ajout de news.

On peut remarquer que l'on met eureka en "privilege", pour qu'il ai la priorité sur middletier et edge du fait qu'il faut obligatoirement qu'il soit lancée pour que les services communiquent. Ensuite pour middletier on le relie ("link") à middletier, annonçant qu'il faut que eureka soit lancer, mais aussi qu'il va devoir communiquer avec ce dernier. De même pour edge.

L'option `--hostname` permet de définir un nom d'host auquel un conteneur qui a besoin de communiquer avec un autre puisse y accéder. De la même façon que www.google.fr se connecte à une adresse IP, si le conteneur eureka cherche à communiquer avec middletier, docker renverra eureka vers ce dernier si il l'appelle avec le hostname.

Après avoir lancé ses commandes, magie ! Cela marche comme avant, mais avec Docker. Mais ne reste pas totalement pratique. En effet, il faut build les dockerfile un à un et lancer les commandes tour à tour afin de déployer l'application. On pourrait regrouper les commandes dans un fichier bash, et lancer le tout, mais ce serait trop facile.

Docker a donc mis en place une fonctionnalité, Docker-Compose, qui permet de déployer plusieurs conteneurs en même temps.

Composer avec Docker

Après avoir installé Docker-Compose (qui n'était pas installé en même temps que Docker pour notre part, nous utiliserons donc un nouveau type de fichier, un `docker-compose.yml`, à la racine de l'application (voir structure). Ce dernier reprend, globalement, le principe des commandes run, mais change la structure pour plus de lisibilité.

```
version: "2"

services:
  eureka:
    build: ./eureka/
    image: eureka
    hostname: "eureka"
    privileged: true
    ports:
      - "80:80"
    networks:
      - mynetwork

  middletier:
    build: ./rss-middletier/
    image: rss-middletier-docker
    hostname: "middletier"
    links:
      - eureka:eureka
    networks:
      - mynetwork

  edge:
    build: ./rss-edge/
    image: rss-edge-docker
    hostname: "edge"
    links:
      - eureka:eureka
      - middletier:middletier
    ports:
      - "9090:9090"
    networks:
      - mynetwork

networks:
  mynetwork:
```

On énonce donc pour les différents services :

- **build** : c'est l'endroit où se trouve le dockerfile du service. En effet, on peut build avec docker-compose si cela n'est pas déjà fait les 3 services à la fois
- **image** : si l'image est déjà build on peut en effet dire laquelle utiliser. Il est recommandé

d'utiliser des tags (si build avec docker-compose, on peut utiliser docker tag <image id> <nouveau nom> avant de docker-compose), mais normalement le docker-compose connaît l'id des images construites.

- **privileged, hostname et links** : même chose que pour le docker un
- **ports** : cela permet de dire quels ports on ouvre de docker vers la machine
- **networks** : afin que les 3 services puissent communiquer on indique dans un réseau interne à docker déclaré plus bas. Les 3 services seront donc dans le même réseau.

Comme dit plus haut on lance donc, on lance ce docker-compose avec la commande

```
docker-compose up
```

Ce dernier lancera donc les 3 services, communiqueront entre eux, et nous pourrons créer mes flux RSS. Mais ça c'est en théorie.

En effet une communication est belle et bien établie entre les 3 services, cependant Edge ne trouve pas sa page jsp/rss.jsp, qui est la page web principale de l'application.

```
eureka_1 | ... 10 more
edge_1 | WARN 15:45:57,084 PWC6117: File "/rss-edge/webapp/jsp/rss.jsp" not found
eureka_1 | 2018-04-25 15:46:00.329 INFO com.netflix.discovery.DiscoveryClient:826 [Di
```

Cela est peut-être dû au fait que j'ai changé dans les fichiers de propriétés de edge ou middletier où nous changions localhost en eureka. Mais nous ne comprenons pas pourquoi cela marche avec docker run mais pas avec docker compose, d'où le fait que nous n'avons pas pu aller plus loin dans la pratique.

Nous avons donc décidé en dernier recours d'utiliser la première méthode de Dockerfile pour rss-middletier et rss-edge. Le tout se trouve dans le dossier RSSReaderH. On retélécharge donc toutes les sources, deux fois.

```
FROM java:8
RUN apt-get update && apt-get install -y git wget libgradle-core-java
RUN git clone https://github.com/Netflix/recipes-rss.git

WORKDIR ./recipes-rss/
ADD edge.properties ./rss-edge/src/main/resources/

RUN ./gradlew clean build -x javadoc

ENV APP_ENV=dev
CMD java -jar rss-edge/build/libs/rss-edge-*SNAPSHOT.jar
```

```
FROM java:8
RUN apt-get update && apt-get install -y git wget libgradle-core-java
RUN git clone https://github.com/Netflix/recipes-rss.git

WORKDIR ./recipes-rss/
ADD middletier.properties ./rss-middletier/src/main/resources/
RUN ./gradlew clean build -x javadoc

ENV APP_ENV=dev
CMD java -jar rss-middletier/build/libs/rss-middletier-*SNAPSHOT.jar
```

Le docker compose est le même que précédemment. Nous le lançons et ça marche. Nous ne comprenons pas pourquoi edge ne trouve pas ses fichiers statiques page web, alors que c'est exactement les mêmes sources, les mêmes fichiers de configurations et par extension les mêmes images. Les images sont juste plus lourdes pour cette méthode car elles ont les sources à l'intérieur.

Concernant le scaling du déploiement, lors du docker compose nous pouvons ajouter une option

--scale SERVICE=NUMBER, qui permet donc de lancer plusieurs instances des services.
Par exemple `docker-compose up --scale eureka=5 middletier=5 edge=5` lancera 5 instances de chaque service. Nous n'avons pas pu tester directement cette partie par manque de temps.

Conclusion

Ce TP fut dur. En effet, nous avons passé beaucoup de temps à trouver pourquoi nos solutions ne marchaient pas. Les solutions proposées n'étaient pas non plus évidentes à trouver, et une fois bloqués pour de bon à la dernière erreur du docker-compose, nous n'avons pu continuer. Cependant, ce TP fut intéressant car il nous a appris à manipuler un outil très utilisé en entreprise, que nous allons surement devoir utiliser plus tard dans nos carrières. Nous regrettons juste que le suivi du TP n'ai pas été présent et que nous soyons lâchés dans la nature sans assez d'indications et d'aide en cas de problèmes.