Présentation succincte de Docker

Afin de simplifier cette description, certaines notions sont décrites d'une manière abstraite qui ne reflète pas leur comportement réel, mais qui permet de se faire une idée de comment les utiliser.

Docker est essentiellement une interface permettant de gérer des conteneurs containerd. Un conteneur peut être vu comme un ou plusieurs processus isolé·s des autres au sein d'une machine. Cette isolation est gérée par le noyau de l'OS, et concerne l'espace disque, réseau, utilisateur, les communications inter-processus, etc. Ces processus peuvent aussi être gérés ensemble, pouvant être stoppés, mis en pause ou relancés.

La gestion de conteneurs Docker suit les principes suivants. Tout d'abord, lorsqu'on souhaite lancer un tel conteneur, un espace disque est créé et "copié" d'un modèle décrit par ce qu'on appelle une image. On trouve typiquement dans cette image les binaires du·es programme·s qu'on souhaite lancer, les fichiers de configurations par défaut, les binaires des librairies nécessaires, etc. Cet espace, et cet espace uniquement, sera accessible aux processus qui seront lancés à l'intérieur du conteneur. Bien sûr, il est possible d'ajouter à cet espace des fichiers ou des répertoires hors conteneur qui seront visibles par ces processus (on parle de volumes), accessibles en lecture/écriture ou en lecture seule. Il est aussi possible de "remplacer" certains fichiers ou répertoires de l'image par le même mécanisme, par exemple pour définir une configuration propre ou rediriger un espace de stockage.

Pour Docker, lancer un conteneur revient à lancer un processus à l'aide d'une commande prédéterminée exécutée au sein du conteneur, libre à ce dernier d'en lancer d'autres ou non. Cette commande sera désignée ci-dessous comme la ***commande principale***. La vie du conteneur sera liée à celle de ce processus. Ainsi, si ce processus s'arrête, le conteneur sera considéré comme arrêté et les autres processus seront également stoppés. L'espace disque réservé ne sera toutefois pas détruit et le conteneur pourra être relancé, en exécutant la même commande de départ. L'espace disque dédié pourra être libéré sauf si le conteneur est en cours d'exécution.

Afin de pouvoir communiquer vers l'extérieur (egress), chaque conteneur disposera (d'au moins) une interface réseau propre. Ainsi, il est tout à fait possible pour deux conteneurs d'écouter sur un même port (par exemple 80), ces deux conteneurs ciblant des interfaces réseau différentes. Cette interface réseau relaiera les messages sortants du conteneur via le noyau et ainsi les processus en conteneur auront les mêmes possibilités d'accès au réseau que ceux hors conteneur. Pour accéder aux conteneurs depuis l'extérieur (ingress) plusieurs solutions sont disponibles.

Dans tous les cas, il est possible de rediriger un port d'une interface réseau de la machine vers un port de l'interface réseau d'un conteneur. Ainsi, si une application en conteneur écoute sur le port 80, il sera possible de rediriger le port 81 de la machine vers le port 80 du conteneur. De même, on pourrait rediriger le port 82 vers le port 80 d'un autre conteneur. Il sera par contre impossible de rediriger le port 80 sur le port de plusieurs conteneurs différents (sauf à jouer avec les différentes interfaces réseau et/ou adresses IP de la machine).

Pour des conteneurs résidant sur la même machine, il est possible de définir un réseau virtuel auquel seront connectés les conteneurs désignés avec une adresse IP propre ; une interface réseau sera ajoutée dans chaque conteneur, connectée à ce réseau virtuel. Un conteneur pourra alors accéder à un autre via cette interface, soit par son l'adresse IP, soit en utilisant son nom ou son identifiant (chaque conteneur a un nom et un identifiant) comme nom DNS. Ainsi, si on trouve dans un premier conteneur nommé *db* une base de donnée écoutant le port 3306, et dans un second conteneur une application ayant besoin de cette base de donnée, il suffit de connecter ces deux conteneurs par un réseau virtuel pour que l'application dans le second conteneur puisse accéder à la base de données en utilisant le nom d'hôte *db* et le port *80*. Attention, si aucun réseau virtuel n'est défini, tous les conteneurs seront connectés à un réseau virtuel par défaut, qui lui ne s'occupe pas de résolution DNS ; tous les conteneurs pourront alors se retrouver par leur adresse IP, mais pas par leur nom (SAUF si les conteneurs sont explicitement liés, ce que nous verrons dans cet exercice).

L'avantage de cette approche est qu'il est possible de faire "tourner" différentes application dans différents conteneurs sur la même machine, de manière sécurisée, et dont le contenu est totalement libre tant que lesdites applications sont compatibles avec le noyau de l'OS de la machine. De plus, de nombreuses images sont disponibles et il est simple de créer ses propres images, comme on le verra ici. Pour lancer une application, il suffit donc de disposer d'une image (qui peut être automatiquement téléchargée) et de lancer un conteneur ; plus besoin d'installation, de vérifier les dépendances et leurs versions, et dans une certaine mesure de configuration. Cette approche ressemble aux machines virtuelles, auxquelles les conteneurs sont (trop) souvent comparés, en ceci qu'il est aussi possible de lancer une machine virtuelle simplement à partir d'une image contenant la configuration souhaitée. Les grandes différences sont que les machines virtuelles, si elles permettent de changer le noyau utilisé, consomment plus de ressources (car elles nécessitent un hyperviseur, une couche de simulation de matériel virtuel, et un OS invité complet), ont des images plus volumineuses (notamment du fait de la présence d'un OS), sont plus lentes à démarrer et à stopper, et l'applicatif contenu plus difficile à monitorer.

Dans cet exercice, on découvrira comment gérer ces différents aspects des conteneurs, ainsi que comment gérer des groupes de conteneurs, tout en restant limité à une seule machine. Pour lever cette dernière contrainte, il faudra se diriger sur un orchestrateur, tel Docker SWARM ou Kubernetes.

Mise en route

Préférez un terminal Linux ou Mac si vous en disposez. Ceux sous Windows utiliseront PowerShell en mode administrateur (!).

Docker Desktop est déjà installé sur les machines de l'école.

Pour installer le démon:

- Sous Linux : https://docs.docker.com/engine/install/ (préférez la méthode " set up Docker's repositories")
- Sous Windows ou Mac : https://docs.docker.com/desktop/
- En cas de problème, vous pouvez vous construire une machine virtuelle

(Note pour l'installation en salle machine : ouvrir tous les droits à la socket docker se fait en indiquant le code suivant à `sudo systemctl edit docker.socket`)

```
[Socket]
SocketMode=666
```

Lancer un conteneur

Commandes importantes

L'interaction avec Docker se fait en ligne de commande. On résume ici les plus courantes, sans oublier que la documentation officielle reste la source la plus complète.

- `docker run image[:tag]` lance un nouveau conteneur à partir d'une image (avec un tag - désignant souvent sa version et/ou sa configuration ; par défaut `latest`) ; on utilisera régulièrement les options
  - `-name nom_conteneur` pour donner un nom au conteneur, sans quoi on ne pourra le désigner (pour les autres commandes) que par son identifiant ou un nom généré
  - `-p port_machine:port_conteneur` pour rediriger un port de l'OS vers un port du conteneur
  - `-d` (*daemonize*) pour lancer le conteneur en arrière plan
  - `--rm` pour supprimer l'espace disque dédié au conteneur lorsqu'il s'arrête
  - `--restart [no(défaut)|on-failure[:max-retries]|always|unless-stopped]` pour indiquer dans quelles condition redémarrer un conteneur qui s'est arrêté
  - `-v fichier_ou_répertoire_machine:fichier_ou_répertoire_conteneur` pour monter un fichier ou un répertoire dans l'espace disque accessible au conteneur
  - `-e NOM_VARIABLE=valeur_variable` ajoute une variable d'environnement dans le conteneur ; les applications qui s'exécutent dans le conteneur pouvant potentiellement en tirer des informations, en fonction de leur implémentation
  - `-l nom_label=valeur` ajouter une métadonnée au conteneur a priori sans effet particulier
- `docker ps` pour lister les conteneurs actifs en indiquant leurs caractéristiques principales (identifiant, nom, image, commande exécutée, ports redirigés, ...) ; on pourra utiliser l'option `-l` pour des informations plus détaillées
- `docker ps -a` pour lister les conteneurs, y compris ceux qui sont arrêtés (inactifs)
- `docker stop conteneur` pour stopper un conteneur actif ; le conteneur peut être désigné par son nom, son identifiant, ou le début de son identifiant tant qu'il n'est pas ambigü (par exemple 9b pour le conteneur d'identifiant 9b4841d73b6e)
- `docker start conteneur`; relance un conteneur inactif en exécutant la commande de base
- `docker restart conteneur`; stoppe un conteneur actif puis le relance ; les processus du conteneur sont donc arrêtés, puis la commande de base est relancée sans modification de la configuration du conteneur (ports redirigés, volumes, ...) ni de son espace disque après l'arrêt des processus
- `docker exec -ti conteneur commande` exécute une commande dans un conteneur, `-t` pour avoir un tty, `-i` pour une session interactive (permettant l'interaction au clavier) ; typiquement, pour explorer un conteneur actif équipé d'un bash : `docker exec -ti conteneur /bin/bash`
- `docker cp conteneur:destination fichier` ou `docker cp fichier conteneur:destination` copie un fichier/un répertoire depuis/vers un conteneur
- `docker rm conteneur` pour supprimer un conteneur inactif, y compris son espace disque dédié ; `docker rm $(docker ps -qa --filter status=exited )` supprimer tous les conteneurs inactifs
- `docker rename conteneur nouveau_nom` pour nommer un conteneur existant
- `docker logs conteneur` voir les logs émis par le conteneur (sortie standard l'appli lancée par la commande principale), `-f` pour les voir apparaître au fur et à mesure (comme pour less ou tail)
- `docker top conteneur` pour voir les processus s'exécutant dans un conteneur
- `docker inspect conteneur` montre en détail la configuration et l'état d'un conteneur
- `docker kill conteneur` envoie un signal SIGKILL au processus lancé par la commande principale ayant pour effet de le terminer brutalement, et du coup rendant le conteneur inactif

Mise en route de conteneurs

On peut lancer un premier conteneur basé sur l'image nginxdemos/hello : `docker run -p 8081:80 nginxdemos/hello`. Comme l'image *nginxdemos/hello* n'était pas connue du système, elle est d'abord téléchargée depuis Docker Hub. Cette image contient les binaires et bibliothèques nécessaires au fonctionnement d'un serveur http nginx. La commande demande aussi à ce que le port *8081* de la machine soit redirigé sur le port *80* du conteneur. Quand le conteneur est lancé, la commande de base est lancée à l'intérieur du conteneur, à savoir la mise

en route du processus nginx qui va écouter sur ce même port *80* (à l'intérieur du conteneur). Si le terminal semble bloqué, on peut tout de même constater qu'on a accès à l'application sur http://127.0.0.1:8081/. L'accès à cette url génère d'ailleurs des logs sur la sortie standard du terminal où on a lancé le conteneur.

Si on ouvre un second terminal, on peut lancer la commande `docker ps` : on voit alors dans la liste le conteneur lancé par la première console. Sont exposés son identifiant, le nom de l'image qui a été utilisée, la commande de base (toujours en cours d'exécution), son état, les ports redirigés et un nom qui a été généré, aucun nom n'ayant été donné lors du `run`. On peut d'ailleurs lui donner un nom plus adapté en invoquant `docker rename identifiant_conteneur srv1`.

Pour en savoir (beaucoup) plus sur le conteneur, on peut utiliser le commande `docker inspect srv1 | less` : on peut y trouver son identifiant (complet), la commande de base (*Config.Cmd*), son état détaillé (*State*), son nom (*Name*), son adresse IP (*NetworkSettings.IPAddress*), son nom d'hôte (*Config.Hostname*), les redirections de port (*HostConfig.PortBindings*), etc. On constate que la commande de base est *nginx*. Notez que dans *less*, on peut utiliser les flèches pour voir le contenu, la touche *'h'* pour le l'aide, et la touche *'q'* pour quitter.

On peut aussi lancer un shell s'exécutant **à l'intérieur** du conteneur : `docker exec -ti srv1 /bin/sh`. L'interprète de commande (*/bin/sh*) fait en effet partie de l'image qui a été utilisée pour lancer le conteneur, et peut donc être invoqué. On peut ainsi aller voir le fichier index.html présenté dans `/usr/share/nginx/html`. On peut également lister les processus qui s'exécutent avec la commande `ps -eaf`. On voit bien s'exécuter le shell `/bin/sh` dans lequel on est, la commande `ps` qui fait le listage des processus, des processus lancé par *nginx* et bien sûr *nginx* lui-même, qui a la particularité d'avoir l'identifiant 1, généralement utilisé pour les systèmes d'initialisation type systemd. Ici, c'est la partie de la commande de base qui est en cours d'exécution, et à qui la vie du conteneur est liée. On peut aussi être surpris du faible nombre de processus. En effet, on ne voit pas les processus exécutés en dehors de ce conteneur. On peut quitter l'interpréteur de commande de manière classique à l'aide de la commande `exit` ou de la combinaison de touche *Ctrl+'d'*. Une nouvelle invocation de `ps -eaf` hors du conteneur permet de se convaincre que les processus hors conteneur n'étaient ps listés à l'intérieur. On constate qu'ici le processus de PID 1 est un autre processus (ici *systemd*), et qu'on voit tout de même le processus *nginx*, mais sous un autre PID.

On peut maintenant revenir au premier terminal (celui qui a servi à lancer le conteneur) et l'arrêter à l'aide de la combinaison de touches *Ctrl+'c'*. Ceci a pour effet d'arrêter le conteneur (ce que confirme la commande `docker ps`). Remarquez que le même effet aurait été obtenu en stoppant nginx dans le conteneur (par exemple en faisant `docker exec srv1 kill -SIGTERM 1`, 1 étant le PID utilisé par l'exécution de la commande principale). Cependant, si le conteneur est inactif (http://127.0.0.1:8081/ ne répond plus), il n'est pas supprimé du système, comme on peur le voir avec `docker ps -a`. On peut donc le relancer avec `docker start srv1`, rendant l'application à nouveau disponible.

On peut lancer une seconde instance de l'application en invoquant `docker run -p 8082:80 -d --name srv2 nginxdemos/hello`. Notez que l'option `-d` rend son exécution en "arrière plan" dans un nouveau processus. Notez aussi que le port de la machine redirigé n'est plus le 8081 (ce qui n'aurait pas été possible, il est déjà utilisé par *srv1*) ; par contre, c'est toujours le port *80* qui est utilisé au sein de ce conteneur, qui utilise une autre interface réseau que le premier. Notez encore que l'image n'a pas eu besoin d'être téléchargée (ni même vraiment copiée, ce que nous avons déjà vu ou détaillerons plus tard dans le cours théorique). Notez enfin qu'on lui donne immédiatement un nom plus présentable dès sa création. On voit maintenant la même application sur http://127.0.0.1:8082/.

Modification d'un conteneur

Entrons exécuter quelques commandes dans ce second conteneur (`docker exec -ti srv2 /bin/sh`). On se propose de peut modifier la page affichée qu'on pourra trouver dans `/usr/share/nginx/html/index.html`. Pour ce faire, il faudra ajouter un éditeur de texte ; l'image se basant sur un environnement Alpine Linux, on pourra l'ajouter à l'aide du gestionnaire de paquet apk : `apk add --no-cache nano`. On pourra par exemple remplacer le logo de nginx (balise `img`) par celui de l'UHA : https://svn.ensisa.uha.fr/bd/ensisa.svg.

Notez qu'il vous faudra peut-être ignorer le cache de votre navigateur pour constater les changements que vous aurez effectué (souvent en rechargeant la page avec *Ctrl+Shift+'R'*).

Répartition de charge

On souhaite maintenant répartir la charge entre les deux conteneur présentant la même application (au changement de logo près). Pour cela, on se propose d'utiliser Træfik. Une image docker est disponible. On peut voir dans la documentation de l'image comment lancer rapidement une répartition de charge. Pour plus de simplicité, on créera le fichier de configuration de la manière suivante (vous pouvez utiliser `nano`) :

```
# traefik.yml
providers:
  docker:
    defaultRule: "PathPrefix(`/`)"
    exposedbydefault: false

# API and dashboard configuration
api:
  insecure: true
```

Træfik peut alors être lancé avec la commande donnée dans la documentation : `docker run -d -p 8080:8080 -p 80:80 -v $PWD/traefik.yml:/etc/traefik/traefik.yml -v /var/run/docker.sock:/var/run/docker.sock traefik:v2.9`. Il est possible que sous Windows cette commande se plaigne que le répertoire /var n'existe pas ; essayez alors //var/run/docker.sock:/var/run/docker.sock. Træfik

écoutera sur le port *80* du conteneur, publié sur le port *80* de la machine, toutes les requêtes arrivant en les redirigeant sur les "services" en fonction de filtres sur les requêtes (les "routers"). Cette commande publie également sur le port *8080* le port *8080* du conteneur qui met à disposition un dashboard permettant de comprendre la configuration effectivement appliquée (quels sont les routeurs et les services).

On voit aussi ici l'utilisation de montage du fichier *traefik.yml* à un endroit prédéterminé dans le conteneur qui sera lu par l'application pour sa configuration. Un autre montage permet de rendre disponible */var/run/docker.sock* à l'intérieur du conteneur. Ce dernier élément est une socket unix qui permettra à Træfik, à l'intérieur du conteneur, de dialoguer avec le gestionnaire Docker pour en lister les conteneurs, et en avoir les détails (attention, il faut avoir confiance en l'application utilisant cette technique). Ceci permettra à Træfik de déterminer quels conteneurs sont à exposer et avec quelle configuration sur quel service en utilisant quel routeur.

Cette configuration d'exposition est déterminée par Træfik en lisant les labels des conteneurs. Un label est une simple étiquette donnant des informations supplémentaires sur un conteneur. On ne peut malheureusement pas ajouter un label à un conteneur existant. Il faut donc les recréer en ajoutant les labels traefik.enable=true et traefik.http.services.*srv*.loadbalancer.server.port=80 (qui permettra à Træfik de repérer que les 2 conteneurs doivent faire partie du même service *srv* et qu'il pourra rediriger les requêtes sur leur port *80*) :

```
docker stop srv1 srv2
docker rm srv1 srv2
docker run -d --name srv1 -l traefik.enable=true -l traefik.http.services.srv.loadbalancer.server.port=80 nginxdemos/hello #
inutile de publier le port à l'extérieur
docker run -d --name srv2 -l traefik.enable=true -l traefik.http.services.srv.loadbalancer.server.port=80 nginxdemos/hello
```

Le dashboard montre alors l'apparition d'un service srv, constitué de deux serveurs dont les IP correspondent aux IPs des deux conteneurs *srv1* et *srv2*. Ce service est exposé par la route par défaut (le chemin est */*, comme défini comme chemin par défaut dans le fichier *traefik.yml*), et en effet, on retrouve l'application sur http://127.0.0.1/ (sur le port *80* publié par le conteneur de Træfik), "load-balancée" avec l'algorithme par défaut tourniquet entre les deux conteneurs comme on peut le constater avec le changement du nom d'hôte donné par la page (à nouveau, attention au cache du navigateur, utilisez les touches *Ctrl+Shift+'R'*, ou observez ces changements à l'aide de curl ou de l'affichage de la source de la page que vous rechargerez). On pourrait changer ce comportement en utilisant un algorithme permettant une affinité de session en utilisant des labels supplémentaires. Vous pouvez lancer de nouveaux conteneurs ou en arrêter, le répartiteur de charge s'adaptera.

Réseaux virtuels

Les conteneurs que nous avons lancé l'ont tous été sur le réseau virtuel par défaut créé par Docker. Les conteneurs peuvent ainsi tous interagir entre eux par le biais de leur adresse IP, comme le fait Træfik pour transmettre les requêtes vers les conteneurs "backend". Cependant, il ne leur est pas possible de communiquer par nom DNS (nom de conteneur ou identifiant. Par exemple, on peut voir le nom d'hôte et l'IP de srv2 : `docker exec srv2 hostname` et `docker exec srv2 ip address show`, puis se connecter à *srv1* pour tenter d'accéder à l'application :

```
docker exec -ti srv1 /bin/sh
curl http://ip_de_srv2 # montre bien la page
curl http://hostname_de_srv2 # erreur
curl http://srv2 # erreur
exit
```

Plus grave, on n'a mis en place aucune sécurité dans le sens où toutes les applications en conteneur peuvent accéder aux services sur IP publiés par les autres dans d'autres conteneurs. On va donc définir un réseau dédié au montage fait précédemment : `docker network create hwnw` où *hwnw* est le nom du réseau créé. On peut voir la liste des réseaux avec `docker network list` ; on y trouve les réseaux créés automatiquement par Docker (dont le réseau par défaut *bridge*) et le réseau nouvellement créé.

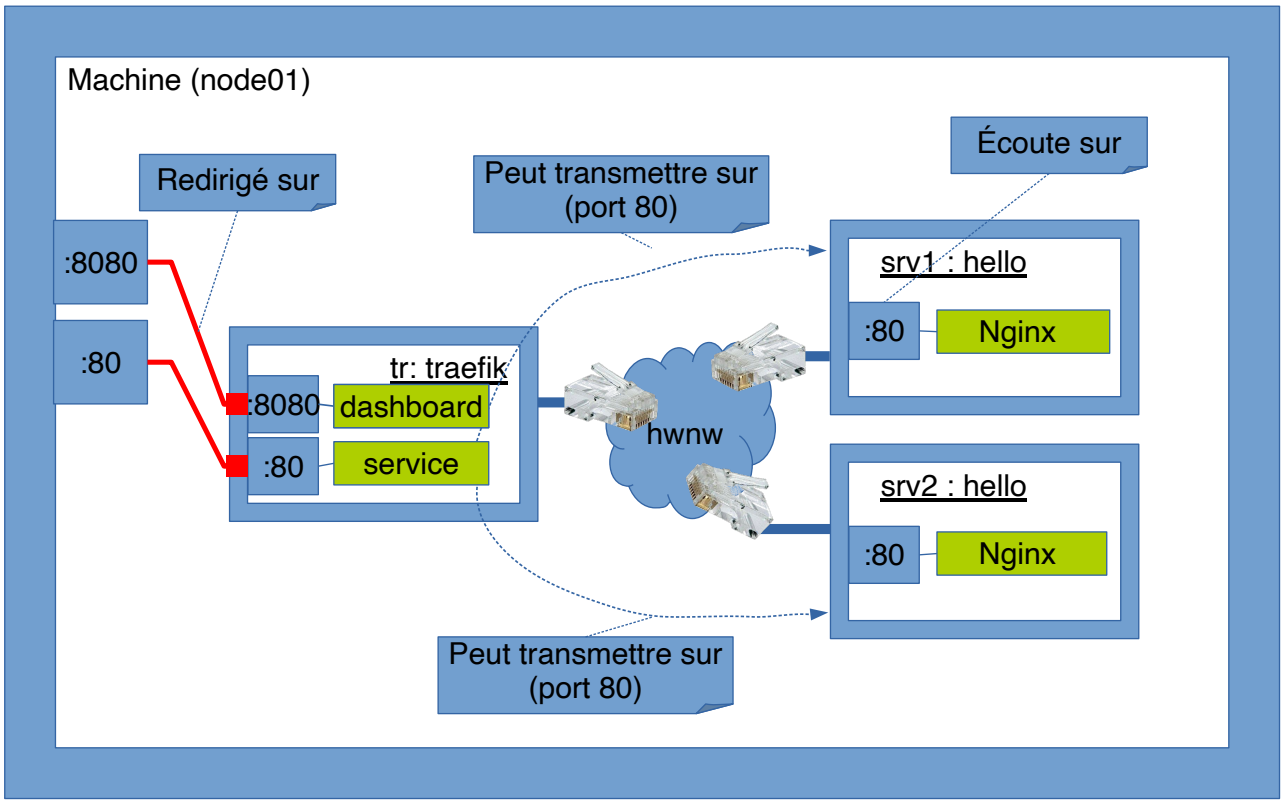
Reste maintenant à recréer les conteneurs sur le modèle précédent, en définissant cette fois le réseau avec l'option `--network` :

```
docker stop srv1 srv2
docker rm srv1 srv2
docker run -d --name srv1 --network hwnw -l traefik.enable=true -l traefik.http.services.srv.loadbalancer.server.port=80
nginxdemos/hello
docker run -d --name srv2 --network hwnw -l traefik.enable=true -l traefik.http.services.srv.loadbalancer.server.port=80
nginxdemos/hello
```

On aurait pu aussi utiliser les commandes `docker network connect` et `docker network disconnect`.

Ce faisant, on constate que Træfik, s'il voit bien arriver les nouveaux conteneurs, ne parvient plus à leur transférer les messages, même s'il utilise leur adresse IP : le réseau virtuel a bien isolé les conteneurs connectés (*srv1* et *srv2*) des conteneurs qui ne le sont pas (*traefik*). De plus, les instructions curl ci-dessus fonctionnent toutes : les conteneurs sur le même réseau virtuel peuvent tous se voir entre eux, y compris avec leur noms DNS (identifiant et nom). On pourra donc finir en connectant le conteneur Træfik au nouveau réseau virtuel : `docker network connect hwnw traefik; docker network disconnect bridge traefik` (on aurait aussi pu le détruire et le recréer comme ce qui a été fait pour les services). On constate alors que http://127.0.0.1/ montre à nouveau l'application, le répartiteur de charge pouvant à nouveau accéder aux conteneurs nécessaires.

On se retrouve in fine sur la configuration suivante :



Gestion des images

Commandes importantes

La documentation officielle reste toujours la source la plus complète.

- `docker images` la liste des images disponibles en local
- `docker pull nom_image[:nom_tag]` récupère une image d'un tag donné ; le tag par défaut est *latest*
- `docker search recherche` pour trouver une image
- `docker rmi nom_image[:nom_tag]` pour supprimer une image du système local ; `docker rmi $(docker images -q)` pour tout détruire (les images utilisées par des conteneurs - actifs ou non - seront conservées)
- `docker tag image` pour ajouter un tag à une image ; l'image peut être identifiée par son nom:tag (ou son nom, le tag étant latest par défaut) ou son identifiant
- `docker history image` ce qu'il s'est passé sur une image et la taille de chaque changement

Gestion binaire

Pour créer une image, une possibilité est de promouvoir un conteneur existant en tant qu'image. Par exemple, on peut changer le logo dans le conteneur *srv2* : `docker exec -ti srv2 /bin/sh -c 'sed -i "s;src=\"[^\"]*;src=\"https://svn.ensisa.uha.fr/bd/ensisa.svg\";" /usr/share/nginx/html/index.html` (ou en ré-appliquant le changement dans le fichier html comme fait en début d'exercice). On voit alors apparaître le logo une fois sur 2 dans http://127.0.0.1/ (*Ctrl+Shift+'R'*). On peut aussi mettre à jour les bibliothèques utilisées à l'aide du gestionnaire de paquets disponible dans l'image utilisée : `docker exec -ti srv2 /bin/sh -c 'apk update;apk upgrade;rm -rf /var/cache/apk/*'`.

Une fois les changements faits, pour définir une image, on lance la commande `docker commit -m "Changed logo to ENSISA's one" -a "Frédéric Fondement" srv2 fondement/hello` (il est d'usage de mettre un nom d'organisation ou d'utilisateur pour les images non officelles). `docker images` fait alors apparaître la nouvelle image, avec le tag *latest*, aucun tag n'ayant été défini lors du `commit` ; on peut en rajouter un avec `docker tag fondement/hello fondement/hello:2`. `docker images` montre alors les deux noms référençant une image de même identifiant.

On peut maintenant faire une mise à jour pour utiliser la nouvelle image :

```
docker stop srv1
docker rm srv1
# À ce moment, seul srv2 permet d'assurer http://127.0.0.1/
docker run -d --name srv1 --network hwnw -l traefik.enable=true -l traefik.http.services.srv.loadbalancer.server.port=80
fondement/hello:2
docker stop srv2; docker rm srv2
# À ce moment, seul srv1, en nouvelle version, permet d'assurer http://127.0.0.1/
docker run -d --name srv2 --network hwnw -l traefik.enable=true -l traefik.http.services.srv.loadbalancer.server.port=80
fondement/hello:2
```

On vient là de faire une mise à jour sans interruption de service ! En effet, le répartiteur de charge a pu à tout moment compter sur au moins un conteneur "backend" actif pour lui transmettre les requêtes. Cependant, une meilleure approche aurait été de mettre en place une affinité de session, un même utilisateur pouvant voir à un moment donné (après le démarrage de *srv1* et avant l'arrêt de *srv2*) alternativement les deux versions de l'application. Avec l'affinité de session, en cas d'affinité avec *srv1*, il aurait été redirigé sur *srv2* dès son arrêt pour y rester (en restant sur l'ancienne version), puis à nouveau sur *srv1* dès l'arrêt de *srv2* en voyant la nouvelle version. Dans le cas où l'affinité aurait été sur *srv2*, l'utilisateur n'aurait vu aussi la nouvelle version qu'à l'arrêt de ce conteneur quant il aurait été redirigé vers *srv1* dans sa nouvelle version.

Dockerfile

On a donc vu que la création d'une nouvelle image se fait en modifiant le contenu d'un conteneur et en le promouvant en tant qu'image. Il est possible de scripter ces changements (en plus de de définir d'autres paramètres) dans un fichier *Dockerfile*. En plus d'offrir une procédure répétable par rapport à des manipulations manuelles comme décrite dans le précédent chapitre, ces fichiers texte sont très petits, et donc très simples à versionner (git et autre) et à partager.

Tout d'abord, un Dockerfile doit définir en tant que première instruction une clause `FROM` définissant une image existante à modifier pour créer la nouvelle image. Les modifications à faire seront données par les clauses `ADD` pour ajouter un fichier disponible au moment de la construction (sur le système de fichiers local ou accessible par une URL) et `RUN` permettant de donner les commandes à réaliser *à l'intérieur de l'image*, ou plus exactement d'un conteneur basé sur l'image de base donnée par `FROM`, pour la modifier (dans notre exemple l'ajout du logo, ou la mise à jour des logiciels). D'autres clauses sont aussi couramment utilisées :

- `ENTRYPOINT` et `CMD` permettent elles de modifier la commande de base (l'instruction donnée par `CMD` étant donnée en paramètre à l'instruction donnée par `ENTRYSET`, ce dernier étant typiquement positionné à "/bin/sh") ; notez qu'il n'est nécessaire de définir ces clauses que si celles de l'image de base ne conviennent pas ;
- `EXPOSE` permet d'indiquer sur quels ports les processus en conteneur écouteront et seront potentiellement à exporter ;
- `ENV` indique quelles sont les variables d'environnement à positionner dans les conteneurs (toujours modifiables avec l'option `-e` de la commande `docker run` - on pourra utiliser à la place `ARG` pour les variables d'environnement qui ne devront être disponibles qu'au moment de la construction de l'image avec les `RUN`) ;
- etc.

Une fois un Dockerfile décrit (de préférence dans un fichier de nom Dockerfile - attention à la casse), on peut construire une image à l'aide de la commande `docker build répertoire_du_Dockerfile`. La commande va alors interpréter chaque ligne du Dockerfile pour construire clause après clause la nouvelle image. Si la construction réussit, l'identifiant de l'image est donné, et on pourra lancer (sur la même machine, cette image n'étant pas exportée) un conteneur s'y basant à l'aide de `docker run identifiant_image`. Pour simplifier l'utilisation de la nouvelle image, il est d'usage de lui donner un nom à l'aide de l'option `-t` de la commande de construction. Ainsi, si vous vous trouvez dans le répertoire où se trouve le Dockerfile, vous construirez ou mettrez à jour une nouvelle image avec la commande `docker build -t nom_image .` (`.` représentant le répertoire courant).

Une bonne manière de se familiariser est d'étudier quelques exemples. Il est assez facile de trouver les *Dockerfile*s utilisés pour construire les images publiées ; je vous conseille d'ailleurs de toujours lires ceux des images que vous utiliserez dans la clause `FROM`.

Exemple 1:

```
FROM debian # forcément la première instruction ; cette image là offre l'environnement habituellement trouvé dans la
distribution Debian
LABEL maintainer=toto@bigcorp.com # métadonnée indiquant qui maintient cette image
RUN apt-get update # instruction pour générer l'image ; comme une commande faite sur une image en cours de route (ici
Debian) AVANT le commit
ADD script.sh /dest/script.sh # ajout d'un fichier dans l'image depuis le disque de la machine où la construction de l'image
se fait
EXPOSE 80 # un port qui devrait être accessible depuis l'extérieur ; essentiellement à titre documentaire
VOLUME /volume/data # un bout de disque qui devrait être accessible depuis l'extérieur à des fins de synchronisation entre
le conteneur et son contenant
CMD ["/bin/bash"] # la commande de base exécutée lorsqu'on lance le conteneur par docker run ; il ne peut y en avoir qu'une
```

Exemple 2: le nginxdemos/hello de tout à l'heure ; on voir qu'il se base sur l'image nginx:mainline-alpine, qui elle-même se base sur nginx:mainline-alpine-slim (où on trouve la commande de base CMD), qui se base sur alpine, etc.

Réalisez le dockerfile qui ajoute le logo dans la page en partant de l'image *nginxdemos/hello*.

Une fois qu'on a ce Dockerfile, on peut fabriquer l'image : `docker build -t fondement/hello .` (attention à ne pas oublier le . à la fin de la commande, indiquant le répertoire où se trouve le fichier Dockerfile à construire). On peut voir ce qu'il s'est passé sur l'image et son impact sur la taille finale : `docker history fondement/hello`.

On peut relancer *srv2* avec cette nouvelle image : `docker stop srv2 && docker rm srv2 ; docker run -d --name srv2 --network hwnw -l traefik.enable=true -l traefik.http.services.srv.loadbalancer.server.port=80 fondement/hello`

Chaque ligne d'un Dockerfile crée un "layer" ; les changements sont cumulatifs ; changer le Dockerfile réutilise les layers précédemment construits. C'est pourquoi on préfèrera

- cumuler sur un seul RUN les commandes qui réalisent une opération (jointes avec des ';' ou des '&&') - il est possible de les mettre sur plusieurs lignes en mettant un '\' à chaque fin de ligne)
- ajouter les clauses changeant fréquemment en fin de dockerfile
- effacer les caches devenu inutiles au runtime (d'où le `rm -rf /var/cache/apk/*`) pour limiter la taille finale d'une image

Une autre clause intéressante est `HEALTHCHECK`, qui permet à docker de tester l'état de santé d'un conteneur plus finement que la simple existence du PID 1. Par exemple, *HEALTHCHECK --interval=20s --timeout=3s CMD curl -f http://localhost/ || exit 1* marquera un conteneur comme *unhealthy* si, depuis l'intérieur du conteneur, il n'aura pas été possible de récupérer *http://localhost* avec un code de retour correct (ex: 200) en moins de 3s (ce contrôlé étant réalisé toutes les 20 secondes).

Il est simple de partager son image sur docker hub : il faut se créer un compte, puis un dépôt, puis lancer `docker login`, et `docker push image` (le nom de l'image devant suivre le format *nom_utilisateur/nom_dépôt*). On peut aussi passer par la création un dépôt de code (Github, Bitbucket ou autre) et le lier au dépôt docker hub ("automated build") : c'est docker hub qui se charge de construire l'image et de la mettre à disposition. Il est aussi possible d'héberger son propre dépôt d'images (un simple `docker run` suffit).

Composition de services

Pour déployer un service, comme par exemple notre hello-world lié à Træfik, il est possible de rédiger des scripts shell pour le déploiement. Une solution plus simple consiste à utiliser docker compose, qui permet de lancer un ensemble de conteneurs à l'aide d'un fichier descripteur en YAML. Ce descripteur permet de définir quels sont les réseaux nécessaire, de créer des volumes (espace disques réservés par Docker destinés à être montés dans un conteneur comme le fait l'option `-v` de `docker run`), et de définir des services. Ces services sont rendus par un conteneur créé pour l'occasion et pouvant être répliqué à la demande.

Un exemple de fichier docker-compose.yml (voir la documentation pour chacune des sections) :

```
networks:
  dbnw: # définition d'un réseau virtuel
services:
  srv: # premier service : l'application hello-world
    image: nginxdemos/hello # son image
#    ports:
#      - 80 # inutile, car exposé par le répartiteur de charge
    networks:
      - dbnw # docker virtual network
    labels:
      - traefik.enable=true
#      - traefik.http.services.srv.loadbalancer.server.port=80 # Inutile, 80 est indiqué par EXPOSE du Dockerfile de nginx:mainline-alpine-slim (ancêtre de nginxdemos/hello), et les conteneurs d'un même service sont vu comme un seul service
  lb: # second service : le répartiteur de charge
    image: traefik:v2.9
    ports: # exposition de ports à l'extérieur
      - 80:80
      - 8080:8080
    networks:
      - dbnw # comme pour srv
    volumes:
      - $PWD/traefik.yml:/etc/traefik/traefik.yml
      - /var/run/docker.sock:/var/run/docker.sock
```

On peut alors lancer d'un coup les deux services : `docker compose up -d` (n'oubliez pas de supprimer les conteneurs créés dans les parties précédentes pour éviter les conflits). On peut voir qu'un conteneur pour chaque service *lb* et *srv* sont lancés, tous le même réseau virtuel que le conteneur correspondant au service *srv*. À nouveau, le service est disponible sur http://127.0.0.1/ et http://127.0.0.1:8080/.

Beaucoup des commandes docker sont également disponibles pour docker-compose (ps, start, stop, rm, logs restart, kill, top, ...), que vous devrez toujours invoquer dans le répertoire où se trouve le fichier *docker-compose.yml*. Par exemple `docker compose logs srv` donne les logs pour tous les conteneurs (ici un seul) du service *srv*.

Pour lancer de nouveaux conteneurs pour le service *srv*, on pourra invoquer `docker compose up -d --scale srv=3`. On voit avec `docker ps`, mais aussi `docker compose ps srv` que deux nouveaux conteneurs ont été lancés avec l'image *nginxdemos/hello*, et sont assemblés dans le même service par Træfik, toujours grâce à la communication entre ce processus en conteneur avec Docker via la socket unix */var/run/docker.sock*. Il est aussi possible de définir le nombre de conteneurs pour un service dans le fichier *docker-compose.yml* à l'aide de la section *deploy.replicas*.

On pourra mettre à jour Træfik à une version plus récente en modifiant le tag de son image par exemple à *v3*, et en invoquant à nouveau `docker compose up -d --scale srv=3` (pour ne pas perdre deux des conteneurs du service *srv*). Une fois de plus, on constate que docker-compose ne fait qu'appliquer les changements nécessaires pour faire correspondre la réalité de la topologie Docker au descripteur. De même, si une nouvelle version (tag *latest*) de *nginxdemos/hello* venait à sortir, `docker pull` téléchargerait automatiquement la nouvelle image et un nouveau `docker compose up -d` recréerait les conteneurs du service *srv* avec la nouvelle image.

Ceci dit, il est d'usage de toujours indiquer une version majeure, afin d'éviter les problèmes de mise à jour (incompatibilité de la configuration, ou problématiques de migration de données par exemple).

On peut stopper et supprimer les conteneurs créées avec `docker compose stop;docker compose rm -f`, ou avec `docker-compose down` qui lui supprimerait aussi le réseau virtuel.

health check

Il est désormais possible de vérifier la bonne marche de son conteneur automatiquement. Il faut une commande qui contrôle la bonne exécution du conteneur. On pourra indiquer :

- la commande à exécuter (qui devra retourner 0 si tout est OK, sinon 1)
- un timeout sur cette commande
- le nombre de ré-essais avant de marquer le conteneur comme défaillant

Pour docker run: `docker run -d --health-cmd 'wget -O - http://localhost 2>/dev/null | grep -q html || exit 1' --health-interval 10s --health-timeout 5s nginxdemos/hello`

On pourra voir le bon (ou mauvais) fonctionnement du conteneur par `docker ps` : un conteneur en bon état aura un statut marqué healthy. Il sera possible de voir les logs de la commande de healthchecking par `docker inspect`, dans la section *State.Health.Log*.

Pour docker-compose, on utilise la directive *healthcheck*. On pourra y associer la directive *depends_on* qui permet d'attendre qu'un service soit disponible avant d'en lancer un autre:

```
services:
  srv:
    image: nginxdemos/hello
    ...
    depends_on:
      - lb # implique le démarrage de lb avant (ici pour l'exemple)
    healthcheck:
      test: "wget -O - http://localhost 2>/dev/null | grep -q html || exit 1"
      interval: 10s
      timeout: 5s
      retries: 3
    lb:
    ...
```

On pourrait à la place l'intégrer au Dockerfile pour que ce soit actif dans tous les déploiements :

```
FROM nginxdemos/hello
LABEL maintainer=frederic.fondement@uha.fr
RUN apk update && apk upgrade && rm -rf /var/cache/apk/*
RUN sed -i 's;src="[^"]*;src="https://svn.ensisa.uha.fr/bd/ensisa.svg";' /usr/share/nginx/html/index.html
EXPOSE 80
HEALTHCHECK --interval=5s --timeout=2s --retries=3 CMD /usr/bin/wget -O - http://localhost 2>/dev/null | grep -q html ||
exit 1
```

Pour utiliser une image construite à partir d'un *Dockerfile* dans un *docker-compose.yml*, on peut utiliser la directive *build* et donner le nom de l'image en *image* :

```
services:
  srv:
    image: fondement/hello
    build: . # le Dockerfile se trouve dans le même répertoire que docker-compose.yml
    ...
```

On peut alors mettre à jour : `docker-compose build && docker-compose up -d --scale srv=3`.

Exercice

Ecrivez un Dockerfile pour l'application todo. Cette application demande un serveur d'application. Je vous suggère d'utiliser l'image officielle tomcat:9-jdk17-temurin.

Vous effacerez le contenu de *${CATALINA_HOME}/webapps/* (`CATALINA_HOME` est une variable d'environnement positionnée par tomcat) et mettrez le war de l'application sur *${CATALINA_HOME}/webapps/ROOT.war*. Il vous faudra aussi définir une sonde `HEALTHCHECK` que vous pourrez définir avec la commande `curl`, qu'il faudra installer avec `apt-get`.

Vous vous connecterez à une base MySQL que vous pourrez lancer simplement par `docker run --name db -e MYSQL_ROOT_PASSWORD="xxxx" -e MYSQL_DATABASE="todo" -d mysql`. On voit dans cet exemple l'utilisation de variables d'environnements pour adapter le conteneur à ses besoins, ici de créer une base et définir le mot de passe administrateur au moment du lancement du conteneur. Les processus lancés dans le conteneur sont sensés en tenir compte pour réaliser les promesses données dans la documentation. Vous configurerez également un volume afin que les données ne soient pas perdues lorsque le conteneur de la base est récréé.

Afin de vous y connecter, il vous faudra définir une variable d'environnement `DATABASE_URL` (utilisée par l'application) dont la valeur sera : *mysql://root:xxxx@db:3306/todo?useSSL=false&allowPublicKeyRetrieval=true&serverTimezone=GMT* (attention à bien échapper le caractère & pour qu'il ne soit pas interprété par bash). On voit ici que le serveur MySQL a pour nom *db*, correspondant au nom du conteneur MySQL lancé plus haut : le nom du conteneur fait office de nom DNS qui peut être résolu par les conteneurs qui y sont liés.

Définissez enfin la variable d'environnement `AUTH` à *0* pour désactiver l'identification de l'application ; cette identification donne un état à l'application qui devrait être partagé. Une autre (et meilleure) solution serait d'utiliser votre implémentation de l'exercice sur le partage de sessions avec Hazelcast.

Les deux conteneurs (application et base de données) devront être connectés par un réseau virtuel.

Réalisez un fichier *docker-compose.yml* pour lancer automatiquement ces deux services, plus Træfik pour load-balancer les différents services Tomcat. Notez qu'il vous sera peut-être nécessaire de clairement indiquer quel est le réseau virtuel à utiliser entre le conteneur Træfik et l'application au cas où vous utilisiez plusieurs réseaux virtuels.

Pour vérifier MySQL, on pourra utiliser la commande `sh -c 'mysqladmin -p$MYSQL_ROOT_PASSWORD ping'`. On voit bien ici que cette commande est exécutée à l'intérieur du conteneur et peut ainsi utiliser la variable d'environnement qui contient le mot de passe root de la base.

Vous utiliserez deux réseaux virtuels distincts : un pour que Træfik puisse dialoguer avec l'application, et un autre que pour que l'application puisse dialoguer avec la base. Il vous faudra alors labelliser le service de l'application pour que Træfik utilise l'IP correspondant au bon réseau virtuel, les conteneurs du service de l'application ayant chacun deux adresses IP différentes (une par réseau). Attention, le nom de ce réseau devra être celui qu'on peut retrouver par la commande `docker network ls`, qui n'est pas le même que celui défini dans votre fichier *docker-compose.yml*.

Pour vérifier l'application, simplement vérifier le bon fonctionnement de l'URL *http://127.0.0.1:8080/* suffira.

Vous veillerez à configurer la clause depends_on prenne en compte une sonde de healthchecking afin que l'application Java ne démarre pas avant que la base de données soit prête à accepter des requêtes.

Enfin, vous essaierez de limiter la taille de l'image de votre application en n'utilisant que le strict nécessaire des composants de la machine virtuelle java comme décrit dans l'article suivant.

Modifié le: mercredi 19 mars 2025, 16:04

Retour au cours