TP Docker

Consignes

Environnement de travail

Pour réaliser ce TP, vous disposerez d'un environnement de travail complet dans le CLOUD.

- Tout d'abord une machine de travail (bureau RDP) par étudiant qui contient l'ensemble des outils nécessaires (docker, kubernetes, terraform, ansible, vscode, …)

L'accès à la VM de travail se fait de multiples manières :

- *Portail HTTPS guacamole **(solution préférée)** : https://access.tpcsonline.org login/mdp de type vmXX (le formateur vous attribuera le numéro - identique au numéro de la VM)*
    - ***Le raccourci ctrl + shift + alt** vous permet de transférer des fichiers depuis et vers votre machine personnelle si besoin*
- *Accès direct en RDP ou SSH sur vmXX.tpcsonline.org (user/pwd = vmXX)*

Lors de la première connexion ou après un reboot la VM lance automatiquement pour vous certains outils (notamment un navigateur Web et un IDE : VScode déjà ouvert dans un répertoire qui contient les éléments de travail)

Je vous encourage fortement à utiliser VScode pour éditer les fichiers et passer les commandes dans le terminal. En effet, directement dans VScode, vous pouvez dans le menu "Terminal / New Terminal" lancer un fenêtre directement dans VScode où vous pourrez passer vos commandes tout en éditant facilement vos fichiers (*Dans de très rares cas, le terminal VScode peut avoir un comportement anormal et rendre certaines commandes ou sorties inopérantes*)

Il y a d'autres outils sur la VM et vous pouvez en installer si vous le souhaitez. Votre user à les droits sudoers permettant de prendre l'identité du user root pour réaliser certaines commandes (via sudo)

Enfin depuis Guacamole, n'oubliez pas le raccourci ***ctrl + shift + alt*** qui ouvre un volet permettant de copier/coller du texte quand cette fonctionnalité (rarement) ne fonctionne pas directement dans votre environnement.
Également depuis ce volet vous pouvez télécharger des fichiers de la VM de travail vers votre poste local ou inversement suivant les besoins.
Enfin, relancer le raccourci pour faire disparaître le volet guacamole.

Attention, lors des copier coller de commandes depuis le fichier PDF, les caractères "tiret" - sont souvent mal interprétés (surtout quand il y en a deux --), si vous avez une erreur, vérifiez sur la ligne de commande si les tirets sont corrects.

Fonctionnement du TP

Vous devez réaliser le TP dans l'ordre, vous pourriez sinon manquer des étapes et informations nécessaires pour la suite

Evaluation

Durant le TP, vous devrez répondre à des questions et effectuer des actions (encadrés en

couleur).

Elles sont identifiées et numérotées (ex: DQ1 - Docker Question 1). Un encadré est numéroté et peut regrouper plusieurs questions.

Au début du TP, constituez-vous un fichier avec votre traitement de texte favori et remplissez le au fur et à mesure du TP en reprenant bien la référence de l'encadré (ex: DQ1), N'hésitez pas à inclure des captures d'écrans lorsque c'est pertinent ou même des schémas si vous voulez illustrer un choix d'architecture de votre part.

En fin de TP, vous devrez poster le résultat de votre travail au format PDF dans la section devoirs sur EDUNAO

Le fichier doit être nommé par <votre nom>-tpdocker-<date>.pdf

Par exemple : claude-tpdocker-20230525.pdf

Dans les questions (cadres en jaune) il est parfois indiqué BONUS : ces questions sont souvent assez/très difficiles et demandent du temps de recherche. N'hésitez pas à les sauter (vous y reviendrez plus tard s'il vous reste du temps), elles comptent peu ou pas pour l'évaluation, il s'agit plus d'un challenge ou d'une façon d'aller plus loin pour les étudiants déjà familiers avec le thème

Objectifs pédagogiques

Créer et manipuler des containers

Créer un container avec une image préexistante sur Docker Hub et les lister

Notre premier objectif est simplement de créer un container en utilisant l'outil en ligne de commande (CLI) pour donner des ordres au daemon (service) docker.

Pour démarrer simplement, on utilisera des images déjà existantes que docker ira télécharger depuis Internet. Tapez la commande suivante :

```
docker run nginx
```

DQ1:

Suite à la première commande : utiliser ctrl + c pour revenir au shell et sortir de l'exécution du container

Lister les containers `docker ps -a` (à quoi sert l'option -a, que se passe-t-il si on ne l'indique pas ?)

Est-ce que le container est encore en fonctionnement ?

Quel est le nom du container ? Quel est son ID ?

Tester maintenant la commande suivante :
`docker run -d --name mynginx nginx`

DQ2:

Lister à nouveau les containers `docker ps` (est-ce que l'option -a est toujours nécessaire ?)

Pouvez-vous indiquer l'effet de l'option -d ?

Quel est le nom du container ? Quel est à votre avis l'intérêt de spécifier un nom particulier au démarrage des containers ? Quel est l'inconvénient ?

BONUS : vous pouvez utiliser la commande docker inspect mynginx pour voir le détail des éléments concernant votre container

Afficher les logs du container

Comment afficher les logs des composants qui tournent dans le container ? Si vous avez donnez un nom à votre container comme indiqué au paragraphe précédent, il suffit de taper la commande suivante (sinon vous devez indiquer l'id de votre container que vous récupérez avec docker ps)

```
docker logs mynginx
```

DQ3:

Que voyez-vous pour les logs de l'image nginx ?

à quoi sert l'option -f dans la commande docker logs ?

Vous pouvez également afficher les events qui vous permettent de suivre les grandes actions réalisées par le daemon docker et comprendre éventuellement d'où vient un problème avant même que votre conteneur n'arrive à démarrer (problème de pull d'image par exemple)

```
docker events --since "1m"
      2023-07-01T05:06:10.616155533Z image pull nginx:latest
      (maintainer=NGINX Docker Maintainers <docker-maint@nginx.com>,
      name=nginx)
      2023-07-01T05:06:12.903966380Z container create
      6d54f38f7bfac3a81364c9d6b2830baa2fa57d649bdf76ed6dfb5f70b9b3e5
      1b (image=nginx, maintainer=NGINX Docker Maintainers
      <docker-maint@nginx.com>, name=exciting_germain)
```

```
2023-07-01T05:06:12.907412535Z container attach
6d54f38f7bfac3a81364c9d6b2830baa2fa57d649bdf76ed6dfb5f70b9b3e5
1b (image=nginx, maintainer=NGINX Docker Maintainers
<docker-maint@nginx.com>, name=exciting_germain)
2023-07-01T05:06:12.977434482Z network connect
baa60e832c62c8414d96feb5e9efee0bb9c091992906907097a26ef71f13ab
bc
(container=6d54f38f7bfac3a81364c9d6b2830baa2fa57d649bdf76ed6df
b5f70b9b3e51b, name=bridge, type=bridge)
[...]
```

Arrêt, démarrage des containers et comportement interne

Nous allons maintenant tester le démarrage, arrêt destruction d'un container ainsi que sa création

Pour commencer vous allez stopper votre container

```
docker stop mynginx
```

Vous pouvez ensuite aller consulter les logs
Vous pouvez redémarrer le container à l'aide de la commande start. Regarder à nouveau les logs.

Maintenant utilisez la commande `docker kill` pour forcer l'arrêt de votre container. (consultez à nouveau les logs)

DQ4:

Que se passe-t-il lors des commandes start et stop ? (aidez vous des logs)

Quelle est la différence entre la commande kill et stop ?

Après avoir kill votre container, vous pouvez en démarrer un autre :

```
docker run -d --name mynginx nginx
```

Notez ce qui se passe et essayez maintenant de supprimer définitivement un container avec la commande rm

Démarrer ensuite un nouveau conteneur avec le nom mynginx, consulter régulièrement les logs quand vous faites toutes ces opérations

DQ5:

Est-ce que l'on peut créer deux containers avec le même nom ?

Quelle est la différence au niveau des logs entre stop/kill et rm ? BONUS : Pouvez-vous l'illustrer au niveau du filesystem ? (regarder sous /var/snap/docker/common/var-lib-docker/ ; si besoin passez la commande en préfixant par sudo ou passez totalement en tant que root sudo -i puis exit ou ctrl+d pour revenir à l'utilisateur courant)

Est-il possible de supprimer (rm) un container qui est en état démarré ? (consultez la documentation de docker rm si besoin)

BONUS : quel est l'intérêt de la commande docker create par rapport à docker run -d ?

BONUS : quel est l'intérêt de la commande docker pause/unpause (par rapport à stop/start)

exécuter des commandes dans le container

Que diriez-vous maintenant de lancer des commandes dans notre container ?

Tester la commande suivante sur un container démarré :

```
docker exec -it mynginx bash
```

Essayer ensuite de passer des commandes de base, ls, whoami, ps, cd, pwd

Essayer aussi une commande de type echo "bonjour"

Sortez du bash en tapant exit ou ctrl + D

Passez ensuite la commande suivante :

```
docker exec mynginx cat /etc/passwd
```

DQ6:

Est-ce que toutes les commandes standards de l'OS sont présentes ? Avez-vous une idée de pourquoi ?

Est-ce que le résultat de la commande echo est visible dans les logs ? (ou d'autres commandes ?) BONUS : avez-vous une idée de pourquoi ?

Quelle est la différence entre le exec -it et exec sans ces options ? Particulièrement quand vous lancez le exec avec bash sans -it , que se passe-t-il ?

Nous allons maintenant nous connecter dans le processus existant avec la commande attach

```
docker attach --sig-proxy=false mynginx
```

Il est normal que rien ne s'affiche, vous pouvez même avoir l'impression que la commande n'a pas marché
Essayer de saisir des commandes (ps, pwd, …)

Depuis un autre terminal lancer la commande suivante (vous pouvez la jouer plusieurs fois)

```
docker exec mynginx curl localhost
```

Consulter ensuite les logs ainsi que la fenêtre du terminal docker attach.
Sortez ensuite du terminal avec le docker attach via la commande ctrl+C

Relancez la commande suivante : `docker attach mynginx`

Refaites quelques tests et sortez à nouveau avec ctrl+C , essayez à nouveau de faire un attach sur votre container.

DQ7:

Est-ce que vous avez pu exécuter des commandes après le docker attach ?

Que donne le résultat de la commande curl ? BONUS : Que voyez-vous dans le terminal de la session docker attach et comment l'expliquez-vous ?

Quelle est la différence de comportement suivant l'option –sig-proxy ?

Quel est l'intérêt de la commande attach ?

Passer des variables disponibles au runtime

Nous pouvons facilement passer des variables d'environnement au container que l'on lance. Cela peut servir à changer son comportement suivant l'image utilisée ou ce que l'on fait ensuite dans l'image.

Nous allons utiliser ici une commande permettant simplement de créer le conteneur, lancer la commande que l'on souhaite et le supprimer définitivement. (C'est très pratique quand on veut faire des tests de mise au point)

```
docker run --rm -it nginx
```

Pour passer une variable on peut utiliser -e ou --env

```
$ docker run --rm -it -e TESTING=45 --env TEST=23 nginx printenv
TESTING TEST
45
23
```

Comprendre la notion d'immutabilité

Nous allons maintenant vérifier s'il est possible d'installer des outils dans notre container et s'ils sont persistants.

Lancer les commandes suivantes

```
docker run -d --name myubuntu ubuntu:22.04 sleep 1d

docker exec -it myubuntu bash
```

Les commandes ping et netstat ne sont pas présentes dans cette image de ubuntu, nous allons les installer (attention à bien passer les commandes suivantes dans le bash du container ubuntu)

```
apt-get update
apt-get install net-tools inetutils-ping
```

Maintenant vous pouvez utiliser les commandes, par exemple

```
ping -c 2 localhost
netstat -ntlp
```

Sortez du bash, nous allons tenter de trouver où ont été installés ces exécutables.
Nous allons chercher à récupérer le répertoire overlay correspondant à notre container

```
docker inspect myubuntu | jq -r '.[] | .GraphDriver.Data.UpperDir'
```

Vous devriez avoir une sortie similaire à celle ci-dessous mais avec un ID de container différent
`/var/lib/docker/overlay2/b20db5f666348b706010f37866d38f0eb6ded21709bcdd767e42d4ac4852c395/diff`

Vous pouvez la mettre dans une variable pour l'utiliser plus facilement :
```
UPDIR=$(docker inspect myubuntu | jq  -r '.[] | .GraphDriver.Data.UpperDir')
```

Et ensuite lister les fichiers dans certains de ces répertoires :
```
sudo ls -l $UPDIR/usr/bin
sudo tail -10 $UPDIR/var/log/apt/history.log
```

Stoppez / killez le container (ne le supprimer pas pour le moment) et ensuite relancez le
```
docker kill myubuntu
docker start myubuntu
```

Testez si les commandes sont encore là. (et prenez note du résultat : ie. `sudo ls -l $UPDIR/usr/bin`)

Ensuite vous pouvez supprimer le container et le redémarrer

```
docker rm --force myubuntu
docker run -d --name myubuntu ubuntu:22.04 sleep 1d
```

Vérifiez si les commandes sont toujours présentes (ping)
Vérifiez ce qui se trouve dans l'overlay FS du nouveau container *(en utilisant une commande similaire au docker inspect que l'on a réalisé auparavant)*

DQ8:

Est-ce que les commandes ping et netstat fonctionnent bien après installation ?

Que voyez-vous dans le dossier usr/bin du overlay diff ? Qu'y a-t-il dans les lignes du fichier de log apt/history.log ?

Que se passe-t-il après le stop/kill puis start du container, est-ce que tout est encore présent ?

Que se passe-t-il après la destruction et recréation d'un nouveau container ? Que constatez-vous au niveau du filesystem ?

Est-ce qu'après un reboot (sudo reboot) de la machine virtuelle (vous pouvez redémarrer le container ?), les installations de binaires ou modifications seront toujours là ? (est-ce que le répertoire overlay est encore là ?) - ceci bien sûr dans le cas où vous n'avez pas supprimé le container avec docker rm –force

Est-ce à votre avis une bonne pratique d'installer des outils ou modifier des fichiers directement une fois le container instancié ? Quels sont les avantages et inconvénients ?

BONUS : est-ce que vous pouvez retrouver au niveau du file system overlay où est la commande ls qui est déjà dans l'image ?

Construire et gérer les images docker (container image)

Manifest de déclaration d'une image container

Nous avons vu en détail comment manipuler des containers sur le runtime Docker. A chaque fois, nous avons utilisé des images (qui proviennent d'Internet) sans en savoir plus, mais de quoi s'agit-il ?

Pour créer une image de container, nous avons besoin de décrire comment on souhaite la packager et ce que nous allons mettre dedans.
Avec Docker (et le terme est utilisé partout), on utilise un fichier DockerFile

Nous allons ici simplement nous familiariser avec les principales directives (les plus utilisées) pour construire une image

Exemple de fichier dockerfile

```
FROM ubuntu:latest
```

```
ENV VERSION=3
WORKDIR /home/myapp
RUN apt-get install -y gpg unzip
USER 1001
COPY my-local-files ./
ADD https://mywebsite/${VERSION}/myremote-file.zip ./
CMD ["/home/myapp/bin/myapp", "-f", "/home/myapp/conf/my.config"]
```

La directive la plus importante étant sans doute FROM qui va vous permettre d'indiquer à partir de quelle image source (pré-existante) vous allez construire cette image.

Essayer à la lecture du Dockerfile de comprendre ce qui va se passer lors du build (chaque ligne sera évaluée l'une après l'autre)

La référence pour les directives disponibles dans un Dockerfile est ici : https://docs.docker.com/engine/reference/builder/

DQ9:

Quelle est la différence principale entre la directive COPY et la directive ADD ?

BONUS : Que va faire la directive USER ? Ne vaut-il pas mieux conserver le user ID par défaut qui est 0 ?

Construire (build) une image à partir de son dockerfile

Vous allez donc copier le contenu ci-dessous dans un fichier nommé Dockerfile dans un répertoire de la machine de travail.

```
FROM ubuntu:latest
ARG version_arg=3
ENV VERSION=${version_arg}
# RUN apt-get update
# RUN apt-get install -y cowsay
# CMD ["sh", "-c", "/usr/games/cowsay Hello World v$VERSION"]
CMD ["/bin/sh", "-c", "echo Hello World v${VERSION}"]
```

Aller dans ce répertoire puis :

```
docker build .
```

Vous devriez obtenir quelque chose du type

```
Sending build context to Docker daemon  2.048kB
Step 1/4 : FROM ubuntu:latest
 ---> 1f6ddc1b2547
Step 2/4 : ARG version_arg=3
```

```
 ---> Running in 2d4b256e5545
Removing intermediate container 2d4b256e5545
 ---> 245f35f4de2a
Step 3/4 : ENV VERSION=${version_arg}
 ---> Running in 8230228168a2
Removing intermediate container 8230228168a2
 ---> 53004eb9bfd5
Step 4/4 : CMD ["/bin/sh", "-c", "echo Hello World v${VERSION}"]
 ---> Running in 61fa933e2dc4
Removing intermediate container 61fa933e2dc4
 ---> 67e3294296c8
Successfully built 67e3294296c8
```

Si vous ne voyez pas de détails, c'est peut-être dû à la version de docker (si vous êtes sur mac par exemple ou si vous utilisez buildah). Dans ce cas, vous pouvez setter une variable permettant de voir la sortie habituelle

```
DOCKER_BUILDKIT=0 docker build .
```

Taper simplement : `docker images` pour voir/lister les images du registry local (et donc celle que vous venez de builder qui n'a normalement pas de nom). Vous voyez aussi celles que vous avez pull auparavant pour créer des containers qui sont stockées en local dans le registry.

Nous allons maintenant rebuilder l'image en lui donnant un tag (qui est un nom humainement lisible). Attention, le nom de l'image est toujours constitué ainsi :

```
<registry-name>/<repository>:<version>
```

- le registry-name est optionnel, si on en met pas, il s'agit du registry local (c'est ce que nous utiliserons avant de voir plus loin d'autres options)
- repository : le nom que vous voulez comme mon-image ou nginx
- version : ce que vous voulez comme un numéro de version. Si vous n'en précisez pas, latest sera automatiquement utilisé

```
docker build . -t hello:v3

docker images | grep hello
hello  v3        62e1205e46b4   2 weeks ago     77.8MB
```

On peut bien sûr tagger une image sans la rebuilder en ciblant son id

```
$ docker tag 62e1205e46b4 hello:testtag
$ docker tag hello:v3 hello:othertag
$ docker images | grep hello
hello  othertag   62e1205e46b4   2 weeks ago      77.8MB
hello  testtag    62e1205e46b4   2 weeks ago      77.8MB
hello  v3         62e1205e46b4   2 weeks ago      77.8MB
```

Vous pouvez maintenant tester facilement votre image en utilisant le tag (ou l'id si vous n'avez pas créé de tag) :

```
$ docker run --rm hello:v3
```

```
Hello World v3
```

Encore plus fort, voyez que vous pouvez modifier le comportement de votre application lors du runtime en vous servant de variables que vous passez à docker run

```
$ docker run --rm -e VERSION=4 hello:v3
Hello World v4
```

Décommenter les lignes dans le DockerFile et commentez la dernière (il ne doit pas y avoir deux fois CMD). Rebuilder votre image avec un tag hello:cowsay par exemple, qui a dit que Linux n'est pas fun ?

```
$ docker run --rm hello:cowsay
 _________________
< Hello World v3 >
 -----------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

DQ10:

Lors du build avec la partie cowsay, qu'avez-vous vu dans les logs du build ?

Éditer le fichier Dockerfile pour changer le numéro de la variable ARG version_arg=4 par exemple et utiliser un autre tag. Est-ce que vous voyez bien le changement de version dans les logs du build ? Et à l'exécution ?

Comment pourriez-vous modifier la valeur de ARG lors du build sans pour autant devoir modifier le fichier DockerFile ? (Cherchez dans la documentation de docker build les options disponibles)

BONUS : Avez-vous vu des lignes "using cache" dans les logs du build ? Qu'est-ce que cela signifie d'après-vous ? Quels problèmes cela peut-il poser ? (renseignez-vous sur l'option --no-cache)

BONUS : Quelle serait la commande pour builder une image avec une version particulière dans le Dockerfile et en même temps taguer l'image avec un tag de fin correspondant à la version (sans toucher au Dockerfile)

BONUS : Essayer d'adapter l'exercice cowsay avec un autre outil d'ASCII ART : figlet

Lister et manipuler les images locales

Au cas où vous ne l'avez pas trouvé vous-mêmes avant pour faire du ménage sur la registry locale, il y a plusieurs manière de supprimer des images :

```
docker image prune -a
```

ATTENTION, c'est très pratique mais assez radical puisque toutes les images qui ne sont actuellement pas utilisées par un container tournant sur la machine seront supprimées du registry local.

Plus restrictif, une suppression des images que vous souhaitez en utilisant le nom du tag ou l'id :

```
$ docker rmi hello:othertag
Untagged: hello:othertag
```

Si vous tentez de supprimer directement un ID d'image qui a plusieurs tags, vous aurez une erreur :

```
$ docker rmi 6430722243fb
Error response from daemon: conflict: unable to delete 6430722243fb
(must be forced) - image is referenced in multiple repositories
```

Vous pouvez également consulter l'historique de construction d'une image :`docker history`
Il existe d'autres outils (comme dive) que nous verrons un peu plus loin dans un mode ADVANCED

```
% docker history hello:cowsay
IMAGE          CREATED        CREATED BY
SIZE      COMMENT
00370e4ff4f5   16 hours ago   /bin/sh -c #(nop)  CMD ["sh" "-c"
"/usr/game…   0B
246874e589cf   16 hours ago   |1 version_arg=3 /bin/sh -c apt-get
install …   55.3MB
cb5a21a8f2cf   16 hours ago   |1 version_arg=3 /bin/sh -c apt-get
update      40.1MB
53004eb9bfd5   16 hours ago   /bin/sh -c #(nop)  ENV VERSION=3
0B
245f35f4de2a   16 hours ago   /bin/sh -c #(nop)  ARG version_arg=3
0B
1f6ddc1b2547   2 weeks ago    /bin/sh -c #(nop)  CMD ["/bin/bash"]
0B
<missing>      2 weeks ago    /bin/sh -c #(nop) ADD
file:2fd2684e989d275c9…   77.8MB
<missing>      2 weeks ago    /bin/sh -c #(nop)  LABEL
org.opencontainers.…   0B
<missing>      2 weeks ago    /bin/sh -c #(nop)  LABEL
org.opencontainers.…   0B
<missing>      2 weeks ago    /bin/sh -c #(nop)  ARG
LAUNCHPAD_BUILD_ARCH     0B
<missing>      2 weeks ago    /bin/sh -c #(nop)  ARG RELEASE
0B
```

Utiliser une registry distante (Docker hub) et y pousser ses propres images

Les images que l'on build sont stockées localement mais ne sont pas accessibles à un daemon docker situé sur une autre machine (ni à un cluster Kubernetes par exemple).

Il existe des registry docker pour stocker et diffuser les images, le plus connu et historique est dockerhub (https://hub.docker.com/)
Vous pouvez gratuitement y ouvrir un compte et ainsi disposer d'un registry où vous seul pouvez publier des images mais vous pouvez les mettre à disposition du monde entier.

Pour le TP, nous n'allons pas nous obliger à utiliser un registry INTERNET (mais vous pouvez le faire si vous le souhaitez). Nous allons utiliser un registry fourni avec microk8s (la distribution kubernetes que nous utiliserons plus tard) et qui est déjà installé sur votre VM de TP.

Pour pusher une image locale dans un registry docker quel qu'il soit, vous devez taguer votre image locale avec la référence complète du registry et ensuite faire le push.

```
docker images
docker tag xxxxx my-registry/my-image:version
docker push my-registry/my-image:version
```

Si la registry est sécurisée et nécessite une authentification (cas de dockerHub) vous devez auparavant vous authentifiez (via docker login par exemple ou configuration directement dans les fichiers du daemon docker)

Le registry local fourni par microk8s https://microk8s.io/docs/registry-built-in est le suivant : `localhost:32000`

Vous pouvez donc taguer une de vos images et la pousser dans ce registry puis réaliser un run en utilisant cette image. **Il est important de réussir cette partie car vous en aurez besoin impérativement pour le TP Kubernetes.**

Tapez les commandes suivantes une par une en essayant de comprendre la logique (il est normal d'avoir certaines commandes qui renvoient des erreurs, essayez de comprendre pourquoi)

```
docker tag <local-image-id> localhost:32000/hello:cowsay
docker push localhost:32000/hello:cowsay
docker rmi localhost:32000/hello:cowsay
docker rmi <local-image-id>
docker run --rm localhost:32000/hello:cowsay
docker run --rm hello:cowsay
```

Vous pouvez lister les images (repositories) dans la registry microk8s :

```
curl http://127.0.0.1:32000/v2/_catalog
{"repositories":["testrepo"]}
curl http://127.0.0.1:32000/v2/testrepo/tags/list?n=10
{"name":"testrepo","tags":["v1"]}
```

DQ11:

Quelle image avez-vous réussi à push sur le registry microk8s ?

Est-ce que l'image du registry microk8s est visible en local avec docker images ?

Que se passe-t-il quand vous essayez de faire un run avec l'image locale hello:cowsay ?

Quelle est la différence entre une première exécution de `docker run --rm localhost:32000/hello:cowsay` et les suivantes

Communication réseau avec les container

Par défaut, le network Docker est un bridge avec celui du Host, ainsi tout est accessible depuis le Host en utilisant les adresses IP des containers.

```
docker network ls
```

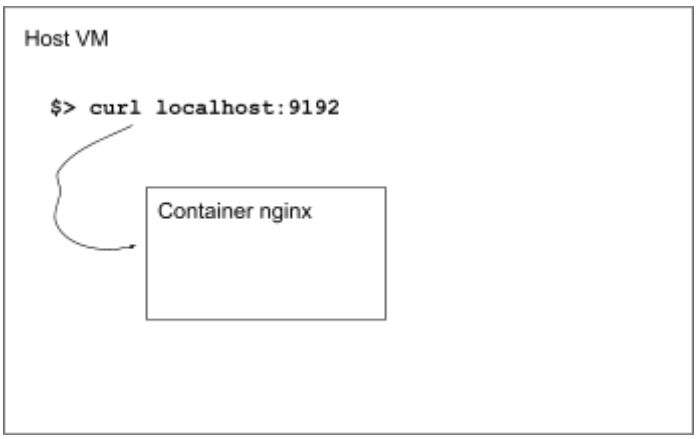
On peut créer de nouveaux networks dans Docker pour des uses cases avancés mais un orchestrateur comme Kubernetes avec un SDN (Software Defined Network) complet permettra d'aller beaucoup plus loin (surtout avec de nombreux hosts diférents).

On peut conclure que la mise en réseau des containers docker est assez simple sur un host unique en utilisant le default bridge (sachant qu'il y a une porosité qu'on peut ne pas souhaiter avec le host)

Accéder depuis la machine hôte à des services TCP/IP tournant sur un container

Dans la première partie, nous avons démarré un container nginx (serveur web) et avons même fait des commandes curl dans le container pour accéder à la page d'index servie par nginx.

Mais nous souhaiterions pouvoir accéder au service nginx qui tourne dans le container depuis la machine hôte par exemple (depuis son navigateur ou depuis un curl sur le terminal mais sans utiliser docker cette fois)



On va donc pouvoir accéder au service directement en utilisant l'adresse IP du container :
`docker run -it --rm --name mywebserver nginx`

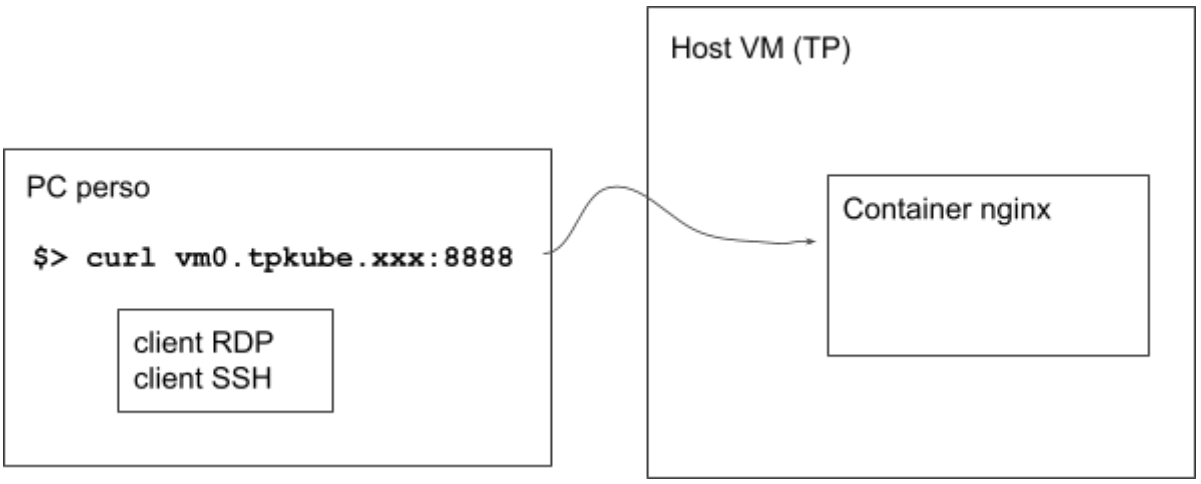
Depuis un autre terminal, vous allez récupérer l'adresse IP attribuée pour le container nginx et réaliser une requête

```
WS_IP=$(docker inspect -f \
      '{{range.NetworkSettings.Networks}}{{.IPAddress}}{{end}}' \
      mywebserver)
curl http://${WS_IP}:80 | grep Welcome
<title>Welcome to nginx!</title>
<h1>Welcome to nginx!</h1>
```

Vous pouvez bien sûr faire un echo de $WS_IP et utiliser l'adresse IP directement dans un navigateur de la VM (firefox ou chrome) pour afficher la page d'accueil de Nginx

Accéder depuis une machine différente de l'hôte qui fait tourner docker

Ici, nous souhaitons accéder à un workload qui tourne sur docker depuis une autre machine (par exemple votre PC personnel qui vous sert à vous connecter en RDP et SSH à la machine du TP qui porte docker)



Comme vous allez utiliser INTERNET, nous n'avons pas ouvert au niveau du firewall CLOUD de la VM TP tous les ports (toutefois, le port 8888 est justement ouvert à tout INTERNET pour différents tests, donc vous pouvez l'utiliser)

```
docker run -it --rm --name mywebserver -p 8888:80 nginx
```

Adaptez à votre numéro de votre vm !!

```
curl -s vm00.tpcsonline.org:8888 | grep Welcome
<title>Welcome to nginx!</title>
<h1>Welcome to nginx!</h1>
```

Vous pouvez voir sur la host VM (TP) que le port 8888 est bien en écoute

```
$ netstat -ntlp | grep 8888
tcp        0      0 0.0.0.0:8888            0.0.0.0:*
LISTEN      -
tcp6       0      0 :::8888                 :::*
LISTEN      -
```

On peut publier les ports d'un container de plusieurs manières :

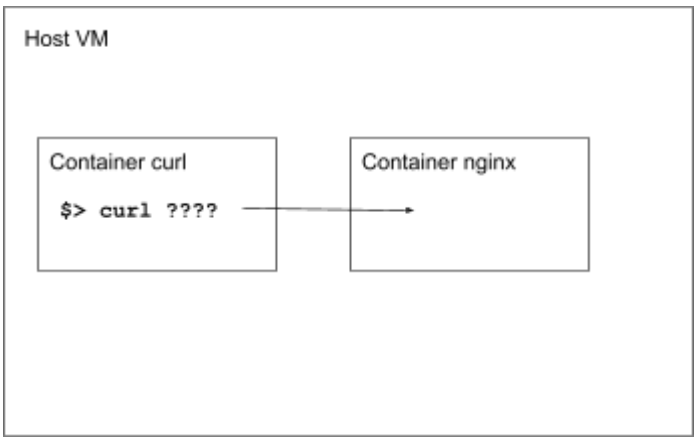
- L'option -P permet de publier les ports qui ont été prévus via EXPOSE dans le DockerFile
- L'option -p permet de publier des ports que l'ont choisi (en surchargeant EXPOSE)
- Si on ne met rien, les ports sont accessibles uniquement depuis le réseau docker (donc depuis les autres containers sur le même network et depuis le host docker lui-même)

Vous pouvez également regarder les ports exposés pour un conteneur particulier avec la commande docker port

```
$ docker port mywebserver
80/tcp -> 0.0.0.0:8888
80/tcp -> :::8888
```

Faire communiquer des containers entre eux au sein du réseau docker

Nous souhaitons maintenant accéder à un container depuis un autre container (tournant sur le même hôte)



Comme nous l'avons vu rapidement, les containers sont tous démarrés par défaut sur le network bridge par défaut de Docker.
Cela permet que tous les containers peuvent discuter entre eux sans limite (ce qui n'est pas toujours souhaitable d'un point de vue sécurité).

Une des limites de ce mode est qu'il faut obligatoirement connaître l'adresse IP de l'autre container (qu'on peut obtenir via la commande docker inspect que l'on a vu plus haut) mais ce n'est pas idéal.

BONUS

Si vous n'êtes pas en avance, vous pouvez sauter ce paragraphe.

Voyons comment on peut faire discuter un container avec un autre en utilisant son nom de deux manières différentes

En ajoutant des entrées dans le /etc/hosts du container

C'est vraiment à la limite du workaround mais cela peut fonctionner (utilisation de la directive add-host) et peut servir pour d'autres use cases.

Dans un terminal, lancer :
```
docker run -it --rm --name mywebserver nginx
```

On va maintenant lancer un autre container qui fera just un curl vers le nom du container initial ning mywebserver

```
$ docker run -it --rm curlimages/curl curl http://mywebserver:80
curl: (6) Could not resolve host: mywebserver
```

ça ne fonctionne pas.

On va donc ajouter l'IP du webserver dans le /etc/hosts du container curl
On regarde ce qu'il y a par défaut dans le etc/hosts
```
docker run -it --rm curlimages/curl cat /etc/hosts
```

Et maintenant les commandes suivantes pour récupérer l'adresse IP et ensuite l'ajouter au fichier etc/hosts du container curl

```
WS_IP=$(docker inspect -f \
      '{{range.NetworkSettings.Networks}}{{.IPAddress}}{{end}}' \
      mywebserver)
docker run -it --rm --add-host mywebserver:${WS_IP} \
      curlimages/curl sh -c 'cat /etc/hosts ; \
      curl http://mywebserver:80'
```

Via un dedicated network pour utiliser les fonctions de Docker

On va devoir créer un nouveau réseau dans docker

```
docker network create --driver bridge mynet
docker network connect mynet mywebserver

docker network disconnect bridge mywebserver
```

On peut voir les networks auquel un container est attaché avec la commande docker inspect (observer la partie networkSettings / networks)

Remarquez que nous n'avons désormais plus besoin de spécifier une entrée dans /etc/hosts et que la résolution de nom fonctionne (nous sommes au sein d'un dedicated network et c'est une feature apportée dans ce cas : https://docs.docker.com/network/bridge/#differences-between-user-defined-bridges-and-the-default-bridge)

```
docker run -it --rm --network=mynet \
      curlimages/curl sh -c 'cat /etc/hosts ; \
      curl http://mywebserver:80'
```

Si vous voulez aller plus loin sur la partie réseau de Docker : https://docs.docker.com/network/network-tutorial-standalone/

Gestion des volumes et de la persistance

https://docs.docker.com/storage/

Cette partie n'est pas indispensable, si vous êtes en retard dans le TP, sautez cette dernière partie sur les volumes, vous y reviendrez ultérieurement si vous le pouvez.

Accéder à des fichiers du host

Un premier cas est de pouvoir accéder aux fichiers/répertoires du host, ce n'est pas forcément une bonne pratique car on peut alors modifier les fichiers du host depuis le container (risque de tout casser)

```
cd
mkdir -p mydir
touch mydir/testfile1
docker run -it --rm --mount \
      type=bind,source="$(pwd)"/mydir,target=/app,readonly \
      ubuntu bash

root@2caea555b41c:/# ls /app/
testfile1
```

Depuis un autre terminal :

```
touch $HOME/mydir/testfile2
```

Si vous retournez dans le shell du container et refaites un listing des fichiers, vous verrez le nouveau fichier. En revanche, il n'est pas possible d'écrire depuis le container

```
root@2caea555b41c:/# ls /app/
testfile1  testfile2
root@2caea555b41c:/# touch /app/myfile
touch: cannot touch '/app/myfile': Read-only file system
```

Sortez du container (exit) et lançons un montage de volume en modification, cette fois on peut écrire un fichier depuis le container sur le host directement (attention danger !)

```
sudo docker run -it --rm --mount \
      type=bind,source="$(pwd)"/mydir,target=/app \
      ubuntu bash
root@07661310b963:/# touch /app/myfile
root@07661310b963:/# ls /app
myfile  testfile1  testfile2
```

Utiliser des volumes et les partager avec d'autres containers

Il est préférable d'utiliser des volumes qui vont être réservés pour l'usage docker et ne nécessite pas l'accès aux répertoires du host. C'est la meilleure pratique et cela permet de maintenir l'isolation entre les containers et le host mais aussi entre les conteneurs eux-mêmes.

Nous allons donc créer un volume et on peut également lister les volumes présents au niveau docker

```
$ docker volume create vol1
vol1
$ sudo docker volume list
DRIVER    VOLUME NAME
local     vol1
```

A titre d'exemple nous allons ensuite monter le même volume sur deux conteneurs et voir que l'on peut ainsi partager un volume (ce n'est pas une bonne pratique d'utiliser un fileSystem pour des écritures concurrentes mais il s'agit d'un test)

```
docker run -it --rm --mount source=vol1,target=/app ubuntu bash
```

Vous pouvez lancer un second conteneur avec la même commande et réaliser des ls et créer des fichiers pour vérifier le bon fonctionnement

```
$ docker run -it --rm --mount source=vol1,target=/app ubuntu bash
root@cdf49b2c1568:/# ls /app
root@cdf49b2c1568:/# touch /app/file.from.$(hostname)
root@cdf49b2c1568:/# ls /app
file.from.cdf49b2c1568
```

```
$ docker run -it --rm --mount source=vol1,target=/app ubuntu bash
root@731d2c52748f:/# ls /app
file.from.cdf49b2c1568
```

```
$ docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED          STATUS
PORTS     NAMES
cdf49b2c1568   ubuntu    "bash"    21 seconds ago   Up 19 seconds
strange_mendel
731d2c52748f   ubuntu    "bash"    42 seconds ago   Up 40 seconds
goofy_mclaren
```

BONUS (DQ12) :
Nous avons rapidement parlé de l'outil Dive (dans les chapitres sur les images) qui permet d'analyser le contenu d'une image docker.
Pour cela, Dive a besoin d'accéder directement à l'API de docker (il a donc besoin d'accéder à la socket docker /var/run/docker.sock

Nous allons donc faire un exemple un peu tordu, monter la socket docker pour utiliser dive sous forme de container qui inspecte sa propre image car il a accès aux fichiers du registry local

```
docker run --rm -it  -v /var/run/docker.sock:/var/run/docker.sock \
       wagoodman/dive ubuntu
docker run --rm -it  -v /var/run/docker.sock:/var/run/docker.sock \
       wagoodman/dive wagoodman/dive
```

Est-ce que vous arrivez à faire fonctionner Dive ? Prenez une capture d'écran !

TIPS : Pour info la syntaxe -v et --mount sont quasi équivalentes (-v est plus compact). La seule différence est que -v créera le répertoire du point de montage demandé s'il n'existe pas alors que --mount sortira en erreur si le répertoire n'existe pas.

Dans dive, utiliser la touche <TAB> pour changer d'onglet, à gauche, vous pouvez sélectionner le layer, une fois passé surl'onglet de droite, vous pouvez avec ^U (Ctrl + U) ne pas afficher les fichiers non modifiés (unmodified) et donc voir facilement les fichiers qui ont changé dans ce layer

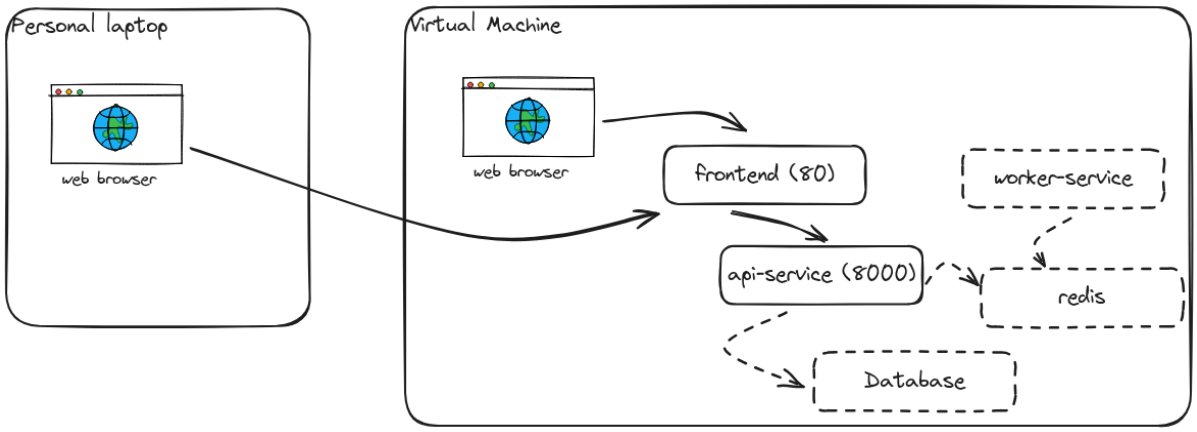
Un exemple concret : l'application demoboard

Afin de rendre cela plus concret, nous allons "containeriser" une application de démonstration. Ceci nous servira également et sera réutilisé pour le TP Kubernetes.

à des fins pédagogiques, nous allons donc utiliser l'application demoboard conçue pour illustrer les TP.

demoboard permet d'ajouter (et supprimer) des tâches fictives dans une liste et éventuellement de lancer un traitement associé à ces tâches en mode asynchrone et afficher le statut de ces traitements.

Architecture globale de l'application



Les éléments en pointillés correspondent au mode complet (permettant de lancer les traitements). Nous allons d'abord travailler sur le mode "simple" avec uniquement le frontend et l'API.

Pour vous aider, vous trouverez sur la VM dans le sous répertoire docker/demoboard le code source et des exemples de Dockerfile pour builder l'application. Attention : le répertoire "complete" contient une version fonctionnelle des Dockerfile avec toutes les réponses. Je vous encourage à commencer sans regarder et d'abord s'inspirer du répertoire "medium" qui contient des fichiers Dockerfile avec des placeHolder à compléter.

Frontend

répertoire : docker/demoboard/demoboard-source/frontend-service

Attention, le frontend sert à hoster/servir le code VUE.js qui s'exécute ensuite sur votre navigateur et va ensuite réaliser les appels à l'API (depuis votre navigateur).

Vous allez donc devoir écrire un DockerFile correspondant au frontend, allez dans demoboard-source/frontend et remplissez le DockerFile (en vous aidant du contenu des répertoires medium et/ou complete) :

- Remarquez que l'on va d'abord utiliser une image node JS pour builder l'application VUE.js
- optionnel : remarquez aussi que l'on définit VITE_ENABLE_WORKER, servez-vous des éléments précédents du cours pour envisager une stratégie pour modifier cette valeur au moment du docker build ou du docker run
- Si vous utilisez la version medium, vous voyez un premier placeholder "STUFF GOES HERE"
    - Vous devez en fait copier l'ensemble des fichiers du répertoire source du fronted
    - Et ensuite lancer la commande de build node : "npm run build"
- Vous voyez ensuite l'image de RUN : nginx:1.25-alpine
- Il y a ensuite (si vous travaillez sur la version medium) 3 TODO :
    - Copier le fichier de configuration nginx dans le futur container sous /etc/nginx/conf.d/default.conf
    - Copier depuis les résultats de l'image build (chercher dans la doc de COPY l'option –from) depuis /app/dist vers le répertoire par défaut (root) renseigné dans le fichier de configuration nginx
    - Exposer le port 80 du container (EXPOSE)

Vous pouvez ensuite builder votre image avec un tag (rappel l'ajout de DOCKER_BUILDKIT=0 est facultatif mais vous permet de plus facilement débugger le docker build en cas de soucis si vous ne voyez pas de détails lors du build)

```
docker build --tag frontend:v1 -f Dockerfile .
```

API (backend)

répertoire : docker/demoboard/demoboard-source/frontend

Vous allez maintenant écrire un DockerFile correspondant au backend api.

Commencer par copier le contenu du Dockerfile medium dans le fichier Dockerfile du code source :

- Vous remarquez qu'il s'agit d'une appli python on utilise un base image python:3.11-slim
- Notez que par défaut on utilise une base de données locale SQLite avec un répertoire de stockage qui est défini dans le DockerFIle
- vous voyez un placeholder "STUFF GOES HERE", vous devez :
    - Copier le fichier requirements.txt
    - Faire un RUN pip install -r requirements.txt pour installer les dépendances python nécessaires
    - Copier l'ensemble des fichiers sources dans le container

Vous pouvez ensuite builder l'image :

```
docker build --tag api:v1 -f Dockerfile .
```

Test de l'application

Maintenant que vous avez builder vos images, nous allons devoir les lancer et s'y connecter. Nous utilisons les ports 7000 et 8000 pour exposer et rediriger vers les ports 80 et 8000 des containers docker.

Essayons donc de lancer nos containers :

```
docker run -d --name frontend -p 7000:80 frontend:v1
docker run -d --name api -p 8000:8000 api:v1
```

Ouvrez le navigateur de la VM et tenter d'accéder à l'application : http://localhost:7000

DQ13:
Est-ce que tout fonctionne ? Spoiler : l'application ne devrait pas fonctionner !! D'où peuvent venir les problèmes ? Est-ce que l'on redirige bien le port en listen de la VM vers le port d'écoute du process dans le container ?

Regardez les logs. Après de premières investigations, continuez à lire ce document pour plus d'informations et un guide vers une solution.

Gérer l'accès réseau à l'application

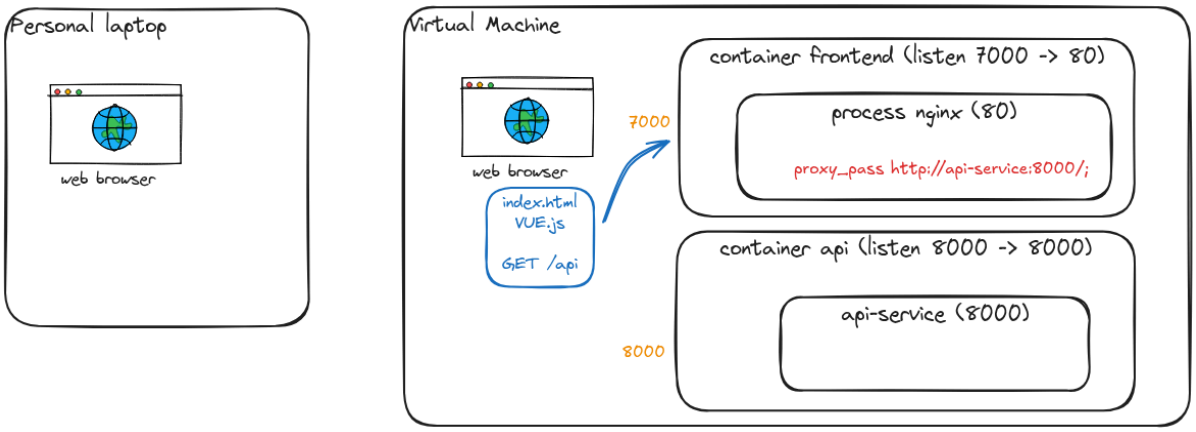
Vous avez peut-être remarqué dans les logs du container frontend une erreur de ce type :

```
host not found in upstream "api-service" in
/etc/nginx/conf.d/default.conf:13
```

C'est bien de là que vient notre problème. En effet, l'application "cliente" est du javascript VUE.js qui s'exécute dans le navigateur et va réaliser des appels à l'API. En fait, pour cela elle continue d'appeler le frontend qui fait reverse proxy pour l'API.

Il faut regarder le fichier de configuration de nginx qui est dans le répertoire du code source du frontend. Vous devez voir un fichier nginx.conf dans un sous répertoire docker. Regarder la directive location /api et la règle proxy_pass.

Quand nginx démarre il vérifie que les servers upstream vers lesquels il doit forwarder certaines requêtes (proxy_pass) sont disponibles. Dans notre cas, il commence par faire une résolution du nom DNS (ici api-service) et ce nom n'existe pas, c'est le plantage.



Alors que faire ? On peut utiliser les solutions évoquées plus haut dans le TP (Faire communiquer des containers entre eux au sein du réseau docker), mais c'est assez fastidieux et nous voulons juste faire un test simple donc nous allons utiliser une autre solution : utiliser un record DNS que nginx sait résoudre !

Nous allons utiliser le record DNS qui est associé et pointe sur l'IP publique de votre VM.
Il est de type vmXX.tpcsonline.org
Notre objectif est donc que l'on puisse appeler l'API via ce record (c'est ce que fera nginx en forwardant les requêtes).
Il faut donc que le container de l'API écoute sur un port qui est accessible depuis l'IP publique (pas filtré par le security group / firewall de AWS).

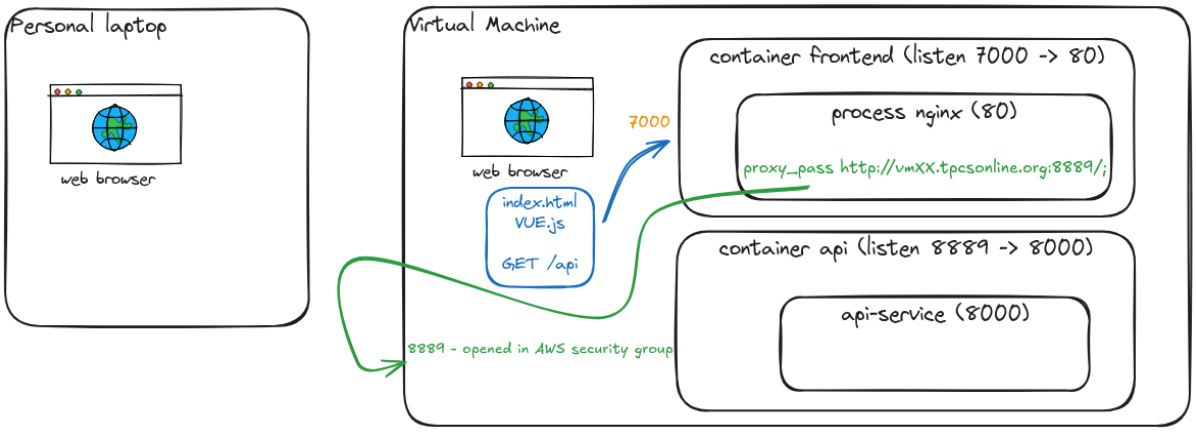
Le port 8889 a justement été pré paramétré pour cela, il est déjà ouvert au niveau du security group AWS, vous pouvez l'utiliser.

Il faut donc rebuilder l'image frontend et utiliser l'option no-cache du build pour éviter une non copie du nouveau fichier de configuration. Dans le fichier nginx.conf, vous devriez donc avoir quelque chose comme :

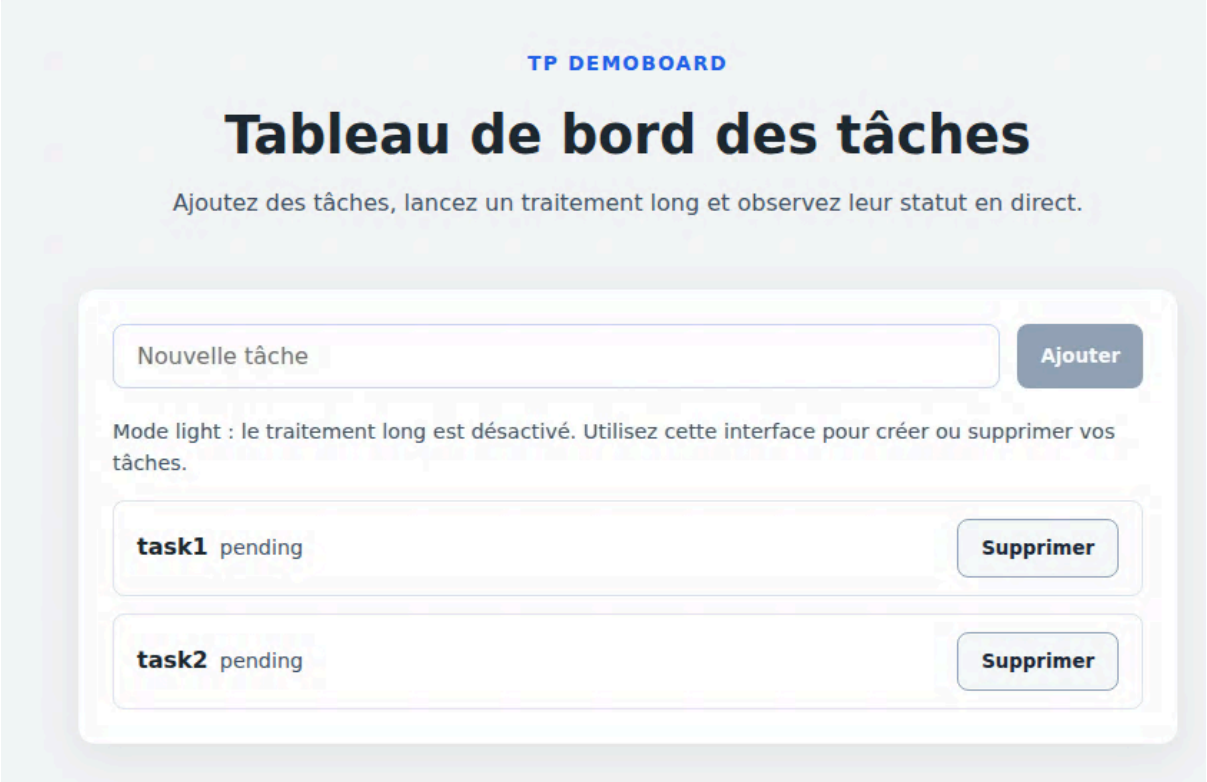
```
   location /api/ {
        proxy_pass http://vmXX.tpcsonline.org:8889/;
```

```
docker build --no-cache --tag frontend:v2 -f Dockerfile .
docker rm -f frontend
docker run -d --name frontend -p 7000:80 frontend:v2
docker rm -f api
docker run -d --name api -p 8889:8000 api:v1
```

Nous avons maintenant un fonctionnement différent récapitulé dans le schéma ci-dessous



Ouvrez le navigateur de la VM et tenter d'accéder à l'application : http://localhost:7000



DQ14:
Est-ce que tout fonctionne désormais ? Vous pouvez ajouter et supprimer des tâches ?

Vous voyez sans doute une différence avec la capture d'écran. Vous pouvez lancer des traitements mais ils ne fonctionnent pas. C'est normal car nous n'avons pas déployé les composants pour cela (nous le ferons dans la partie kubernetes).

Toutefois, vous pouvez paramétrer le client pour désactiver l'affichage et passer en mode light. Rechercher dans le code source VITE_ENABLE_WORKER et trouver une solution pour modifier le comportement du client. Indiquez la solution que vous avez trouvé.

Nous aimerions maintenant accéder à l'application depuis notre poste personnel (PC physique).
Vous pouvez tenter d'accéder à http://vmXX.tpcsonline.org:7000 , est-ce que cela fonctionne ?

Comment faire ? Le port 7000 n'est sans doute pas ouvert sur la VM au niveau des security groups. Le port 8889 l'est et nous l'avons utilisé pour l'API.
Le port 8888 est également ouvert et vous pouvez donc l'utiliser.

DQ15:
Est-ce que vous arrivez à faire fonctionner l'application depuis votre PC personnel ? Qu'avez-vous fait pour y arriver ?

En plus d'accéder au frontend à travers INTERNET depuis votre PC personnel, vous devriez remarquer que vous pouvez accéder directement à l'API (via curl ou même le navigateur), tester des URLs comme :

http://vmXX.tpcsonline.org:8889/tasks
http://vmXX.tpcsonline.org:8889/tasks/1

Vous devriez avoir des réponses correctes. C'est un peu gênant car nous ne souhaitons pas exposer l'API directement puisque nous voulons utiliser notre proxy_pass et que tout passe obligatoirement par là.
C'est une limite de la mise en réseau simplifiée que nous avons utilisé pour Docker. Nous verrons ultérieurement comment traiter cela différemment (et plus proprement) dans Kubernetes.

Base de données

Nous n'allons pas ajouter de base de données persistante pour cette partie docker (nous le ferons dans le TP Kubernetes ainsi que pour le mode complet de l'application avec la gestion des traitements)

Un fichier statique de base de données (sqlite) est créé et permet de fonctionner mais si vous supprimez le container docker, tout est perdu. (par contre un start/stop fonctionne)

BONUS (DQ16) :
Regarder dans le DockerFIle pour trouver l'emplacement de ce fichier sqllite par défaut. Vous pouvez alors créer un volume docker et le monter à l'emplacement où est écrit le fichier de data afin de rendre cette micro installation persistante même en cas de suppression du container.

Partager la configuration que vous avez réalisée et les commandes pour permettre de persister les données.