CRÉER SES PROPRES IMAGES DOCKER AVEC LE DOCKERFILE

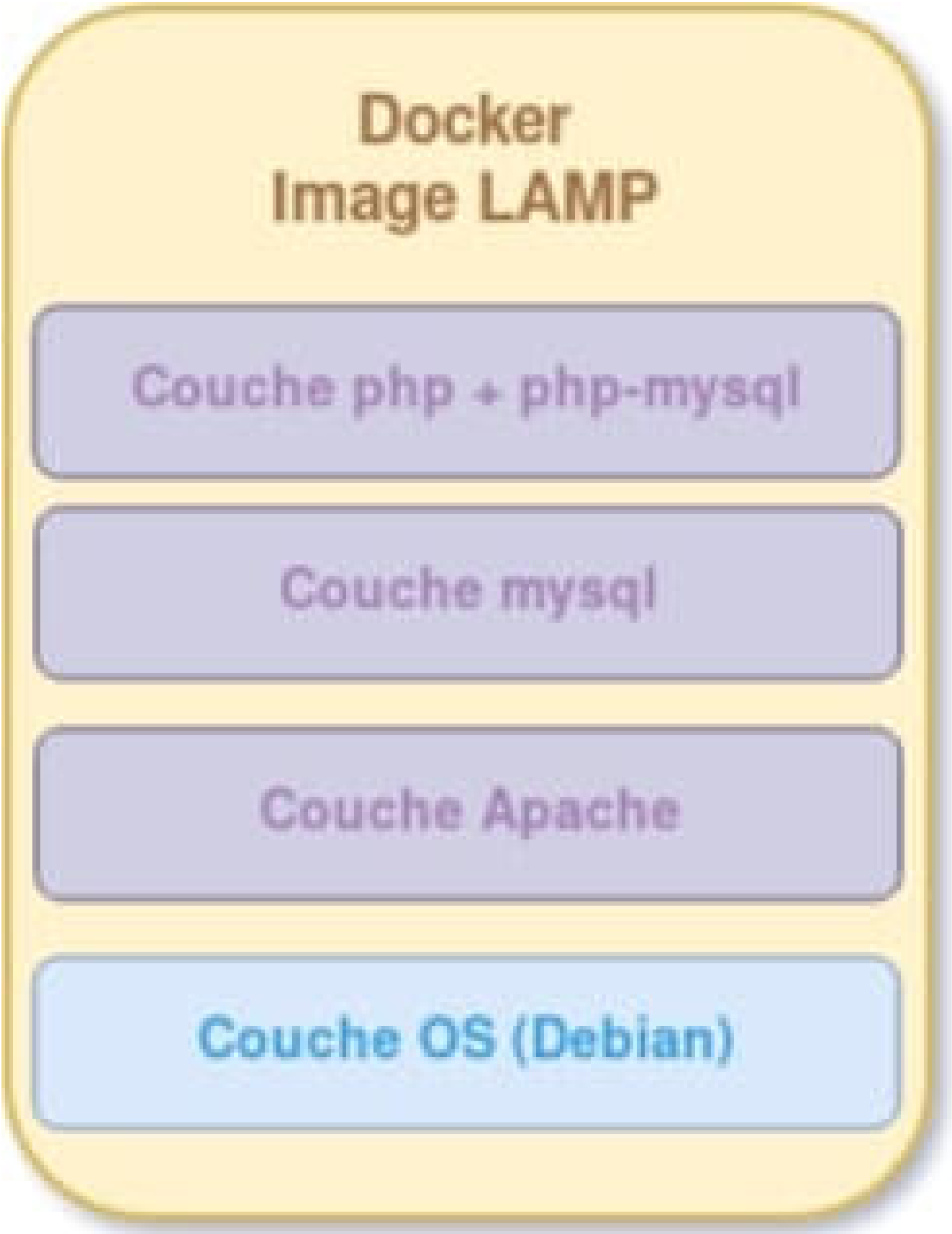
Introduction

Il est temps de **créer vos propres images Docker** à l'aide du fichier Dockerfile. Petit rappel, une image est un modèle composé de plusieurs couches, ces couches contiennent notre application ainsi que les fichiers binaires et les bibliothèques requises.

Pour s'exercer, nous allons créer notre propre stack LAMP (Linux Apache MySQL PHP) au moyen de Docker. Voici les différentes couches de cette image :

- Une couche OS pour exécuter notre Apache, MySQL et Php, je vais me baser sur la distribution Debian.
- Une couche Apache pour démarrer notre serveur web.
- Une couche php qui contiendra un interpréteur Php mais aussi les bibliothèques qui vont avec.
- Une couche Mysql qui contiendra notre système de gestion de bases de données.

Voici le schéma de notre image :



Les différentes instructions du Dockerfile

Avant de créer notre propre image, je vais d'abord vous décrire les **instructions Dockerfile** les plus communément utilisées.

- `FROM` : Définit l'image de base qui sera utilisée par les instructions suivantes.
- `LABEL` : Ajoute des métadonnées à l'image avec un système de clés-valeurs, permet par exemple d'indiquer à l'utilisateur l'auteur du Dockerfile.
- `ARG` : Variables temporaires qu'on peut utiliser dans un Dockerfile.
- `ENV` : Variables d'environnements utilisables dans votre Dockerfile et conteneur.
- `RUN` : Exécute des commandes Linux ou Windows lors de la création de l'image. Chaque instruction `RUN` va créer une couche en cache qui sera réutilisée dans le cas de modification ultérieure du Dockerfile.
- `COPY` : Permet de copier des fichiers depuis notre machine locale vers le conteneur Docker.
- `ADD` : Même chose que COPY mais prend en charge des liens ou des archives (si le format est reconnu, alors il sera décompressé à la volée).
- `ENTRYPOINT` : comme son nom l'indique, c'est le point d'entrée de votre conteneur, en d'autres termes, c'est la commande qui sera toujours exécutée au démarrage du conteneur. Il prend la forme de tableau JSON (ex : CMD ["cmd1","cmd1"]) ou de texte.
- `CMD` : Spécifie les arguments qui seront envoyés au `ENTRYPOINT`, (on peut aussi l'utiliser pour lancer des commandes par défaut lors du démarrage d'un

conteneur). Si il est utilisé pour fournir des arguments par défaut pour l'instruction `ENTRYPOINT`, alors les instructions `CMD` et `ENTRYPOINT` doivent être spécifiées au format de tableau JSON.

- `WORKDIR` : Définit le répertoire de travail qui sera utilisé pour le lancement des commandes `CMD` et/ou `ENTRYPOINT` et ça sera aussi le dossier courant lors du démarrage du conteneur.
- `EXPOSE` : Expose un port.
- `VOLUMES` : Crée un point de montage qui permettra de persister les données.
- `USER` : Désigne quel est l'utilisateur qui lancera les prochaines instructions `RUN`, `CMD` ou `ENTRYPOINT` (par défaut c'est l'utilisateur root).

Je pense, que vous avez sûrement quelques interrogations pour savoir quand est-ce-qu'il faut utiliser telle ou telle instruction. Ne vous inquiétez car à la fin de ce chapitre, je vais rédiger une FAQ, pour répondre à quelques une de vos interrogations.

Création de notre image

Normalement pour faire les choses dans les règles de l'art, il faut séparer l'image de l'application web par rapport à l'image de la base de données. Mais pour ce cours je vais faire une exception et je vais mettre toute notre stack dans une seule image.

Création des sources et du Dockerfile

Commencez par créer un dossier et téléchargez les sources de l'image, en cliquant ici.

Désarchivez le fichier zip, et mettez les dossiers suivants dans votre nouveau dossier :

- **db** : contient un fichier **articles.sql**, qui renferme toute l'architecture de la base de données.
- **app** : comporte les sources php de notre l'application web.

Création des sources et du Dockerfile

Ensuite dans la racine du dossier que vous venez de créer, créez un fichier et nommez le **Dockerfile**, puis rajoutez le contenu suivant :

```
# --------------- DÉBUT COUCHE OS -------------------
FROM debian:stable-slim
# --------------- FIN COUCHE OS ---------------------


# MÉTADONNÉES DE L'IMAGE
LABEL version="1.0" maintainer="AJDAINI Hatim <ajdaini.hatim@gmail.com>"


# VARIABLES TEMPORAIRES
ARG APT_FLAGS="-q -y"
ARG DOCUMENTROOT="/var/www/html"



# --------------- DÉBUT COUCHE APACHE ---------------
RUN apt-get update -y && \
    apt-get install ${APT_FLAGS} apache2
# --------------- FIN COUCHE APACHE -----------------



# --------------- DÉBUT COUCHE MYSQL ----------------
RUN apt-get install ${APT_FLAGS} mariadb-server

COPY db/articles.sql /
# --------------- FIN COUCHE MYSQL ------------------



# --------------- DÉBUT COUCHE PHP ------------------
```

```
RUN apt-get install ${APT_FLAGS} \
    php-mysql \
    php && \
    rm -f ${DOCUMENTROOT}/index.html && \
    apt-get autoclean -y

COPY app ${DOCUMENTROOT}
# --------------- FIN COUCHE PHP --------------------


# OUVERTURE DU PORT HTTP
EXPOSE 80


# RÉPERTOIRE DE TRAVAIL
WORKDIR  ${DOCUMENTROOT}


# DÉMARRAGE DES SERVICES LORS DE L'EXÉCUTION DE L'IMAGE
ENTRYPOINT service mariadb start && mariadb
```

Voici l'architecture que vous êtes censé avoir :

```
|___ app
|   |___ db-config.php
|
|___ index.php
|___ db
|
|___ articles.sql

|___ Dockerfile
```

Explication du Dockerfile

```
FROM debian:stable-slim
```

Pour créer ma couche OS, je me suis basée sur l'image debian-slim. Vous pouvez, choisir une autre image si vous le souhaitez (il existe par exemple une image avec une couche OS nommée alpine, qui ne pèse que 5 MB !), sachez juste qu'il faut adapter les autres instructions si jamais vous choisissez une autre image de base.

```
LABEL version="1.0" maintainer="AJDAINI Hatim <ajdaini.hatim@gmail.com>"
```

Ensuite, j'ai rajouté les métadonnées de mon image. Comme ça, si un jour je décide de partager mon image avec d'autres personnes, alors ils pourront facilement récolter des métadonnées sur l'image (ex: l'auteur de l'image) depuis la commande `docker inspect <IMAGE_NAME>`.

```
ARG APT_FLAGS="-q -y"
ARG DOCUMENTROOT="/var/www/html"
```

Ici, j'ai créé deux variables temporaires qui ne me serviront qu'au sein de mon Dockerfile, d'où l'utilisation de l'instruction **`ARG`**. La première variable me sert comme arguments pour la commande `apt`, et la seconde est le répertoire de travail de mon apache.

```
RUN apt-get update -y && \
apt-get install ${APT_FLAGS} apache2
```

Par la suite, j'ai construit ma couche Apache. Pour cela j'ai d'abord commencé par récupérer la liste de paquets et ensuite j'ai installé mon Apache.

```
RUN apt-get install ${APT_FLAGS} mariadb-server
COPY db/articles.sql /
```

Ici, je commence d'abord par télécharger le service mysql et ensuite je rajoute mon fichier **articles.sql** pour mon futur nouveau conteneur.

```
RUN apt-get install ${APT_FLAGS} \
    php-mysql \
    php && \
    rm -f ${DOCUMENTROOT}/index.html && \
    apt-get autoclean -y

COPY app ${DOCUMENTROOT}
```

Ici j'installe l'interpréteur php ainsi que le module php-mysql. j'ai ensuite vidé le cache d'apt-get afin de gagner en espace de stockage. J'ai aussi supprimé le fichier **index.html** du DocumentRoot d'Apache (par défaut **/var/www/html**), car je vais le remplacer par mes propres sources.

```
EXPOSE 80
```

J'ouvre le port HTTP.

```
WORKDIR  /var/www/html
```

Comme je suis un bon flemmard d'informaticien , j'ai mis le dossier **/var/www/html** en tant que répertoire de travail, comme ça, quand je démarrerai mon conteneur, alors je serai directement sur ce dossier.

```
ENTRYPOINT service mariadb start && mariadb < /articles.sql && apache2ctl -D FOREGROU
```

Ici, lors du lancement de mon conteneur, le service mysql démarrera et construira l'architecture de la base de données grâce à mon fichier **articles.sql** . Maintenant, il faut savoir qu'un **conteneur se ferme automatiquement à la fin de son processus principal**. Il faut donc un processus qui tourne en premier plan pour que le conteneur soit toujours à l'état running, d'où le lancement du service Apache en premier plan à l'aide de la commande `apache2 -D FOREGROUND`.

Construction et Execution de notre image

Voici la commande pour qui nous permet de construire une image docker depuis un Dockerfile :

```
docker build -t <IMAGE_NAME> .
```

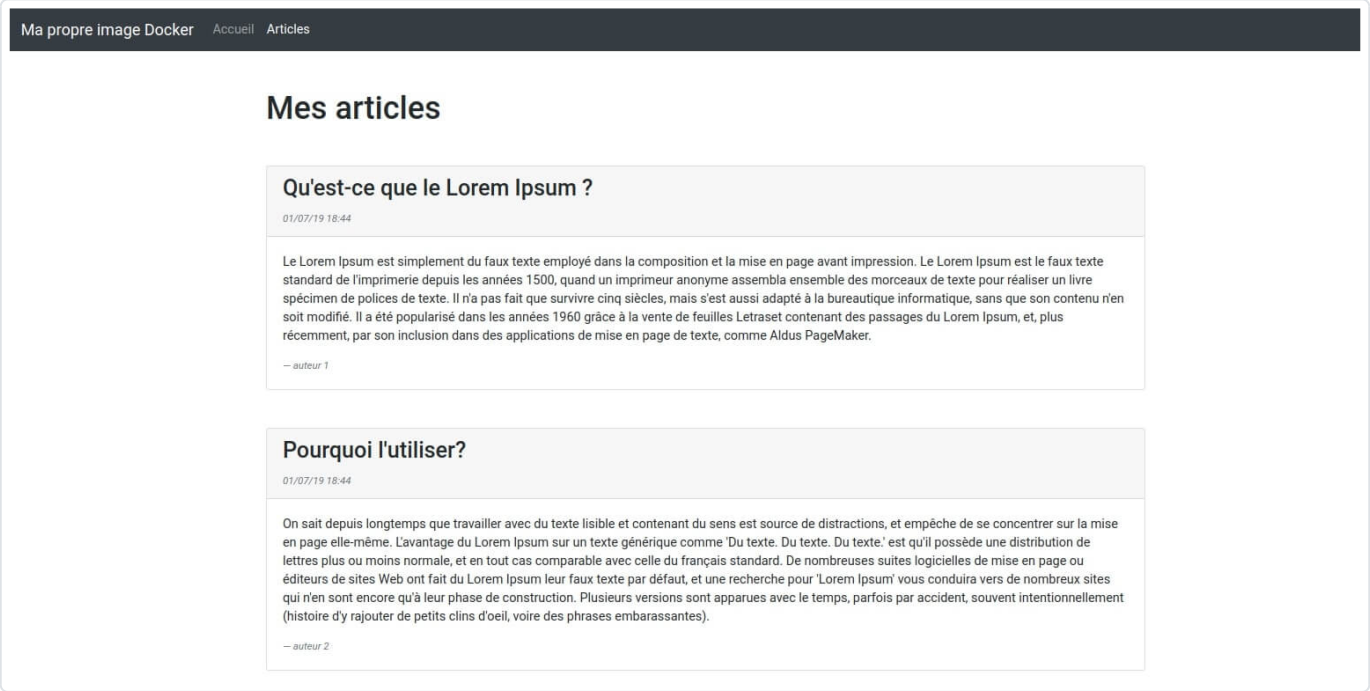
Ce qui nous donnera :

```
docker build -t my_lamp .
```

Ensuite, exécutez votre image personnalisée :

```
docker run -d --name my_lamp_c -p 8080:80 my_lamp
```

Visitez ensuite la page suivante http://localhost:8080/, et vous obtiendrez le résultat suivant :



Bravo ! vous venez de créer votre propre image Docker !

FAQ Dockerfile

Promesse faite, promesse tenue. Je vais tenter de répondre à quelques questions concernant certaines instructions du Dockerfile.

Quelle est la différence entre ENV et ARG dans un Dockerfile ?

Ils permettent tous les deux de stocker une valeur. La seule différence, est que vous pouvez utiliser l'instruction `ARG` en tant que variable temporaire, utilisable qu'au niveau de votre Dockerfile, à l'inverse de l'instruction `ENV`, qui est une variable d'environnements accessible depuis le Dockerfile et votre conteneur. Donc privilégiez `ARG`, si vous avez besoin d'une variable temporaire et `ENV` pour les variables persistantes.

Quelle est la différence entre COPY et ADD dans un Dockerfile ?

Ils permettent tous les deux de copier un fichier/dossier local vers un conteneur. La différence, c'est que `ADD` autorise les sources sous forme d'url et si jamais la source est une archive dans un format de compression reconnu (ex : zip, tar.gz, etc ...), alors elle sera décompressée automatiquement vers votre cible. Notez que dans les best-practices de docker, ils recommandent d'utiliser l'instruction `COPY` quand les fonctionnalités du `ADD` ne sont pas requises.

Quelle est la différence entre RUN, ENTRYPOINT et CMD dans un Dockerfile ?

- L'instruction `RUN` est **exécutée pendant la construction de votre image**, elle est souvent utilisée pour installer des packages logiciels qui formeront les différentes couches de votre image.
- L'instruction `ENTRYPOINT` est **exécutée pendant le lancement de votre conteneur** et permet de configurer un conteneur qui s'exécutera en tant qu'exécutable. Par exemple pour notre stack LAMP, nous l'avions utilisée, pour démarrer le service Apache avec son contenu par défaut et en écoutant sur le port 80.
- L'instruction `CMD` est aussi **exécutée pendant le lancement de votre conteneur**, elle définit les commandes et/ou les paramètres de l'instruction `ENTRYPOINT` par défaut, et qui peuvent être surchargées à la fin de la commande `docker`

`run`.

Comme expliqué précédemment, il est possible de combiner l'instruction **`ENTRYPOINT`** avec l'instruction **`CMD`**.

Je pense qu'un exemple sera plus explicite. Imaginons qu'on souhaite proposer à un utilisateur une image qui donne la possibilité de lister les fichiers/dossiers selon le paramètre qu'il a fournit à la fin de la commande `docker run` (Par défaut le paramètre sera la racine `/`).

On va commencer par créer notre image Dockerfile, en utilisant l'instruction **`ENTRYPOINT`** :

```
FROM alpine:latest

ENTRYPOINT ls -l /
```

Ensuite on construit et on exécute notre image :

```
docker build -t test .
```

```
docker run test
```

Résultat :

```
drwxr-xr-x    2 root     root          4096 Jun 19 17:14 bin
...
drwxr-xr-x   11 root     root          4096 Jun 19 17:14 var
```

Par contre si je tente de surcharger mon paramètre, j'obtiendrai toujours le même résultat :

```
docker run test /etc
```

Pour pouvoir régler ce problème, nous allons utiliser l'instruction `CMD`. Pour rappel l'instruction `CMD` combinée avec `ENTRYPOINT` doivent être spécifiées au format de tableau JSON. Ce qui nous donnera :

```
FROM alpine:latest

ENTRYPOINT ["ls", "-l"]
CMD  ["/"]
```

On va reconstruire maintenant notre image et relancer notre image avec le paramètre personnalisé.

```
docker build -t test .
```

```
docker run test /etc
```

Résultat :

```
-rw-r--r--    1 root     root             7 Jun 19 17:14 alpine-release
...
-rw-r--r--    1 root     root          4169 Jun 12 17:52 udhcpd.conf
```

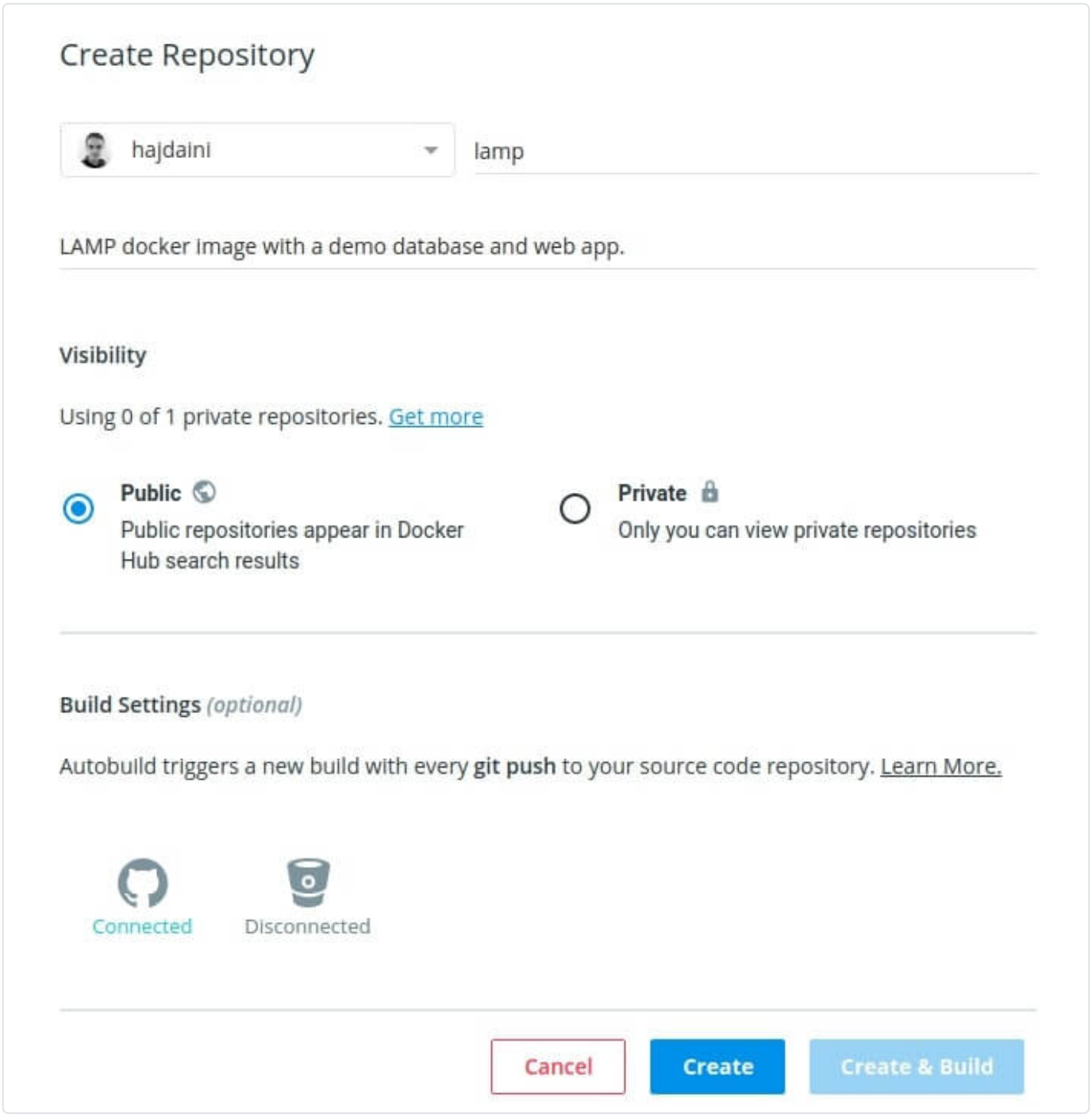
Voilà l'objectif est atteint .

J'espère, que vous avez bien compris la différence entre les différentes instructions, si ce n'est pas le cas alors n'hésitez pas à me poser des questions dans l'espace commentaire, il est prévu pour ça .

Publier son image dans le Hub Docker

Si vous souhaitez partager votre image avec d'autres utilisateurs, une des possibilités est d'utiliser le Hub Docker.

Pour cela, commencez par vous inscrire sur la plateforme et créez ensuite un repository public.



Une fois que vous aurez choisi le nom et la description de votre repository, cliquez ensuite sur le bouton **create**.

L'étape suivante est de se connecter au hub Docker à partir de la ligne de commande

```
docker login
```

Il va vous demander, votre nom d'utilisateur et votre mot de passe, et si tout se passe bien vous devez avoir le message suivant :

```
Login Succeeded
```

Récupérer ensuite l'id ou le nom de votre image :

```
docker images
```

Résultat :

```
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
my_lamp             latest              898661ad8fb2        35 seconds ago      497MB
alpine              latest              4d90542f0623        12 days ago         5.58M
debian              stable-slim         7279351ce73b        3 weeks ago         55.3M
```

Ensuite il faut rajouter un tag à l'id ou le nom de l'image récupérée. Il existe une commande pour ça, je vous passe d'abord son prototype et ensuite la commande que j'ai utilisée.

```
docker tag <IMAGENAME OU ID> <HUB-USER>/<REPONAME>[:<TAG>]
```

soit :

```
docker tag my_lamp hajdaini/lamp:first
```

Si vous relancez la commande `docker images`, vous verrez alors votre image avec le bon tag.

Maintenant envoyez la sauce , en pushant votre image vers le Hub Docker grâce à la commande suivante :

```
docker push <HUB-USER>/<REPONAME>[:<TAG>]
```

soit :

```
docker push hajdaini/lamp:first
```

Conclusion

Ce chapitre vous a appris à créer des images Dockers personnalisées. Dans le prochain chapitre, nous verrons comment persister nos données avec l'utilisation des volumes Docker.