Atelier Docker

IFT-2001: Systèmes d'exploitation
GLO-2001: Systèmes d'exploitation pour l'ingénierie

1 Installation

Cet atelier nécessite l'utilisation d'une machine virtuelle, que vous pouvez télécharger à ce lien. Une fois que vous avez téléchargé la machine virtuelle, vous pouvez l'ouvrir à l'aide de VMware ou de VirtualBox. Pour vous authentifier sur la machine virtuelle, utilisez le nom d'utilisateur `glo2001` et le mot de passe `glo2001`.

Une fois authentifié, ouvrez un terminal avec `CTRL-ALT-T` où vous pouvez exécuter des commandes. Pour vérifier que la machine virtuelle fonctionne correctement, exécutez la commande suivante dans le terminal :

```
echo 'Hello world'
```

Si tout est configuré correctement, vous devriez voir s'afficher dans la console le texte `Hello world`. Pour activer le partage du presse-papier (*clipboard*), modifiez l'option dans `Devices -> Shared Clipboard -> Bidirectional`.

2 Consignes

La machine virtuelle fournie est déjà configurée correctement pour les ateliers. Les commandes de corrections sont de la forme `correction_nn`, où `nn` est un nombre entier représentant le numéro de l'exercice, par exemple `correction_03` pour l'exercice 3. Les instructions pour valider les commandes seront fournies avec la première question ayant un résultat attendu.

Pour réaliser cet atelier, vous devrez effectuer chaque exercice directement dans le terminal. Vous devrez utiliser l'éditeur de texte `nano` pour modifier les scripts demandés. Pour ouvrir un fichier avec nano, exécutez la commande suivante dans le terminal.

```
nano test.sh
```

Vous pourrez alors éditer le fichier, inscrivez y

```
echo 'Hello world'
```

Pour quitter nano, appuyez sur `CTRL-X`, puis répondez `y` pour sauvegarder les modifications. Exécutez les commandes suivantes pour valider que le tout fonctionne :

```
chmod +x test.sh
./test.sh
```

Ces commandes devraient afficher le texte `Hello world`.

Pour les aventuriers seulement : Si vous êtes à l'aise avec la ligne de commande, nous vous encourageons à essayer de réaliser cet atelier en utilisant `vim` comme éditeur de texte. Une brève introduction aux commandes de `vim` est disponible sur le site de The Missing Semester par le MIT. `vim` offre de nombreux raccourcis pour éditer du texte et du code de manière très efficace. En tant que programmeur, vous passerez beaucoup de temps à écrire du code, il est donc judicieux d'investir dans l'apprentissage de `vim` pour le reste de votre carrière. Il existe des *plugins* émulant les commandes `vim` pour la majorité des IDEs : Vim pour vscode ou IdeaVim pour les produits JetBrains **Veuillez noter qu'aucune assistance concernant `vim` ne sera fournie pendant cet atelier.**

3 Solution

Les solutions proposées aux exercices de ces ateliers sont disponibles dans le dépôt GitHub suivant : ulavalIFTGLOateliers/IFT2001-Docker.

4 Docker

Après votre succès avec Bash, votre patron vous a confié une nouvelle mission : trouver une solution au problème de déploiement des applications. Votre entreprise rencontre fréquemment des pannes dues à des incompatibilités entre les différentes versions des dépendances. En effet, chaque développeur peut choisir le système d'exploitation sur lequel il souhaite travailler. Par conséquent, certains développeurs utilisent différentes distributions Linux (Ubuntu, Arch, Gentoo), tandis que d'autres préfèrent Windows.

Cette situation entraîne souvent des problèmes de version, car les dépendances varient d'un système d'exploitation à l'autre. Et le pire, c'est que les versions utilisées diffèrent de celles déployées sur le serveur de production. L'excuse récurrente «Ça fonctionnait sur ma machine» lorsqu'un développeur provoque une panne sur les serveurs de production a finalement exaspéré votre patron. Il vous demande donc de trouver une solution à ce problème.

C'est ainsi que vous découvrez Docker, une plateforme qui vous permet de créer, déployer et exécuter des applications dans des conteneurs légers et isolés. Grâce à Docker, vous pouvez créer un conteneur spécifique pour chaque microservice, en incluant toutes les bibliothèques et dépendances nécessaires. Cette approche garantit que chaque application fonctionnera de manière cohérente, indépendamment des versions des bibliothèques utilisées par les développeurs individuels. De plus, Docker permet de reproduire l'environnement de développement sur le serveur de production, éliminant ainsi les problèmes liés aux différences d'environnement.

Avec Docker, vous allez pouvoir relever le défi consistant à exécuter les trois principaux microservices de votre entreprise. Voici les microservices que vous devrez faire fonctionner :

1. un application de surveillance de la santé des serveurs web, écrite en Haskell ;
2. une API de gestion de documents, écrite en Rust, nécessitant une base de données PostgreSQL ; et
3. une application en Python permettant de visualiser la santé des serveurs web.

4.1 Pourquoi la conteneurisation ?

Isolation L'un des principaux avantages de la conteneurisation est l'isolation. Il arrive souvent que différentes applications nécessitent des versions spécifiques de dépendances. Avec Docker, il est possible d'installer différentes versions de dépendances dans des conteneurs distincts, sans risquer de conflits ou de compromettre le fonctionnement des autres applications. De plus, si vous devez exécuter des applications sur différents systèmes d'exploitation, tels qu'Ubuntu et Arch Linux, vous pouvez simplement les encapsuler dans des conteneurs Docker, évitant ainsi la nécessité de réécrire les applications pour chaque système.

Portabilité La portabilité est un autre avantage clé de Docker. Une fois que vous avez configuré un environnement de développement avec toutes les dépendances nécessaires, il devient fastidieux de reproduire cette configuration sur d'autres machines ou pour de nouveaux membres de l'équipe. Grâce à Docker, vous pouvez créer une image contenant toutes les dépendances et configurations requises, qui peut être facilement distribuée et exécutée sur différentes plateformes, qu'il s'agisse de systèmes Linux, Windows ou autres. Cela permet à un nouveau membre de l'équipe de démarrer rapidement sans passer des semaines à configurer son environnement. Et c'est aussi un avantage lors du déploiement des applications sur un serveur.

Reproductibilité La reproductibilité est également simplifiée grâce à Docker. Les images Docker sont versionnées, ce qui signifie qu'il est facile de reproduire les builds à l'identique, en garantissant que les environnements de développement, de test et de production sont cohérents. Il suffit de spécifier la version de l'image Docker utilisée pour garantir la cohérence et éviter les problèmes liés aux variations entre les environnements.

Efficacité En termes d'efficacité, Docker présente un avantage significatif par rapport aux machines virtuelles (VM). Contrairement aux VM, Docker n'a pas besoin d'exécuter un système d'exploitation complet pour chaque conteneur, ce qui réduit considérablement la consommation de ressources. Les conteneurs Docker partagent le noyau (*kernel*) de l'hôte, ce qui les rend légers et rapides à démarrer.

Communauté La communauté autour de Docker est très active et propose une multitude d'images déjà configurées pour divers outils et technologies populaires tels que Node.js, Python, et bien d'autres. Ces images préconfigurées facilitent le déploiement et l'utilisation de ces technologies, permettant aux développeurs de gagner du temps en évitant de configurer manuellement chaque environnement.

Industrie Enfin, il est important de souligner que Docker est devenu un standard de facto dans l'industrie du développement logiciel. De nombreuses entreprises utilisent Docker pour le développement, les tests et le déploiement de leurs applications. Docker est dorénavent un outil essentiel pour les développeurs logiciels.

4.2 Qu'est-ce que la conteneurisation ?

Docker est une plateforme logicielle open source qui permet de créer, déployer et exécuter des applications dans des conteneurs légers et isolés. Un conteneur Docker est une unité d'exécution qui encapsule une application ainsi que tous ses éléments nécessaires, tels que les bibliothèques, les dépendances et les fichiers de configuration. Ces conteneurs sont autonomes et portables, ce qui signifie qu'ils peuvent être exécutés de manière cohérente sur différents systèmes, qu'il s'agisse d'un environnement de développement, de test ou de production.

Docker est similaire à une machine virtuelle qui isole une application du système hôte. Toutefois, Docker est beaucoup plus efficace qu'une machine virtuelle, qui demande un système d'opération complet pour chaque instance. Docker virtualise partage plutôt le noyau (`kernel`) du système d'opération hôte, ce qui permet d'économiser des ressources et de rendre les conteneurs plus rapides à démarrer et à exécuter.

Voici un peu de terminologie sur Docker. Vous n'êtes pas obligé de lire les sections avancées pour réaliser cet atelier. En supplément, vous pouvez visionner la vidéo «Never install locally»[2].

Image Docker Une image Docker est un modèle ou un plan de construction qui contient tous les éléments nécessaires pour exécuter une application. Elle comprend le système d'exploitation, les bibliothèques, les dépendances, le code source de l'application et les fichiers de configuration. Les images Docker sont créées à partir de fichiers appelés `Dockerfile` qui spécifient les étapes pour construire l'image.

Dockerfile Un Dockerfile est un fichier texte qui contient les instructions pour construire une image Docker. Il spécifie les couches de l'image, les dépendances à installer, les fichiers à inclure et les commandes à exécuter lors de la construction de l'image.

Conteneur Docker Un conteneur Docker est une instance en cours d'exécution d'une image Docker. Il s'agit d'un environnement isolé qui exécute l'application avec ses dépendances. Les conteneurs sont légers, portables et autonomes, ce qui permet de les déployer facilement sur différentes machines.

Registre Docker Un registre Docker est un référentiel centralisé qui stocke et gère les images Docker. Le registre public par défaut est Docker Hub, où vous pouvez trouver de nombreuses images prêtes à l'emploi. Vous pouvez également créer et utiliser votre propre registre privé pour stocker vos propres images.

***Union filesystem* (avancé)** L'*Union Filesystem*, également connu sous le nom de *UnionFS* ou *OverlayFS*, est une technologie utilisée par Docker pour gérer les images et les couches de conteneurs de manière efficace. L'*Union Filesystem* permet de superposer plusieurs systèmes de fichiers en une seule vue logique, sans les fusionner physiquement. Cela signifie que les images Docker et les conteneurs peuvent partager et réutiliser des couches de fichiers communs, ce qui permet d'économiser de l'espace de stockage. Cela permet aussi d'accélérer la construction des images en utilisant une cache des couches déjà construites. Lorsqu'un conteneur est démarré, une nouvelle couche en lecture-écriture est ajoutée au-dessus des couches d'image, permettant ainsi les modifications spécifiques à ce conteneur sans affecter les autres.

***cgroups* (avancé)** Les *cgroups*, ou *control groups*, sont une fonctionnalité du noyau Linux utilisée par Docker pour limiter et gérer les ressources système utilisées par les conteneurs. Les *cgroups* permettent de contrôler les ressources telles que le processeur, la mémoire, la bande-passante du disque et le réseau, afin de garantir une utilisation équilibrée et équitable des ressources système entre les conteneurs. Docker utilise les *cgroups* pour définir des limites et des quotas sur les ressources allouées à chaque conteneur, assurant ainsi une isolation et une performance prévisibles.

***Namespaces* (avancé)** Les namespaces sont une fonctionnalité du noyau Linux qui permet d'isoler les ressources système entre les processus. Docker utilise plusieurs types de namespaces pour fournir une isolation entre les conteneurs, notamment le namespace PID (isolation des processus), le namespace réseau (isolation du réseau), le namespace utilisateur (isolation des utilisateurs) et le namespace de montage (isolation des points de montage). Ces namespaces garantissent que chaque conteneur a sa propre vue isolée du système, ce qui empêche les processus d'un conteneur d'interférer avec d'autres conteneurs ou le système hôte.

***Chroot jail* (avancé)** Chroot, ou change root, est une fonctionnalité Unix/Linux qui permet de changer le répertoire racine d'un processus et de limiter son accès au système de fichiers. Docker utilise la fonction chroot pour créer un environnement isolé à l'intérieur du conteneur, où le répertoire racine du conteneur devient le nouveau point de départ pour tous les chemins de fichiers. Cela limite l'accès du conteneur aux fichiers et répertoires en dehors de son environnement isolé, renforçant ainsi la sécurité et l'isolation.

4.3 Conteneurs manuellement

Afin de bien illustrer ce que fait Docker, nous allons construire un conteneur simplifié manuellement sans l'aider de Docker. Cette section est inspirée par [4]. Dans un terminal, entrez les commandes suivantes :

```
# On valide que neofetch n'est pas installe
# La commande suivante devrait lancer une erreur, ne pas installer le paquet
neofetch

# Dossier pour les conteneurs
mkdir ~/containers; cd ~/containers

# Construit le systeme de fichier du conteneur
sudo apt update; sudo apt install -y debootstrap
sudo debootstrap jammy ./ubuntu-container http://archive.ubuntu.com/ubuntu/

# Cree un environnement isole (namespace)
sudo unshare --uts --pid --mount --ipc --fork

# Monter les dossiers de processus, systeme, les devices et les ppa
mount -t proc none ./ubuntu-container/proc/
mount -t sysfs none ./ubuntu-container/sys
mount -o bind /dev ./ubuntu-container/dev
mount -o bind /tmp ./ubuntu-container/tmp/
cp /etc/apt/sources.list ./ubuntu-container/etc/apt/sources.list

# On utilise chroot pour lancer une Shell dans le conteneur
chroot ./ubuntu-container/ /bin/bash
# et voila, on est dans un conteneur isole

# On installe neofetch
apt update
apt install -y neofetch

# On teste neofetch, la commande fonctionne!
neofetch

# Vous pouvez faire des commandes dans le conteneur ici.

# On quitte le conteneur
exit

# On quitte le unshare
exit

# Si on essait neofetch a nouveau, la commande n'est pas trouvee.
# On a donc bien isole le conteneur!
neofetch
```

Ainsi, comme vous pouvez le voir, il n'y a pas de magie dans les conteneurs. On utilise des outils de base de Linux afin d'isoler un processus dans son propre système de fichiers. Voici une autre ressource intéressante sur les dessous de Docker : p8952/bocker: Docker implemented in around 100 lines of bash.

4.4 Les bases de Docker

Exercice 1: Exécution d'une image déjà faite

Exécutez l'image Docker `hello-world`.

Notez bien: Pour exécutez une image Docker, utilisez la commande `run`.

```
docker run nom-image
```

Si la commande fonctionne, vous verrez un message s'afficher.

Solution: 1

```
docker run hello-world
```

Notez bien: Pour exécuter une commande dans le conteneur, il suffit de spécifier la commande à la fin de la commande `run`. Dans les exemples suivants, on passe la commande `bash -c 'echo "Hello world"'` qui interprète la chaîne de caractère dans l'interpréteur Bash.

```
# Affiche 'Hello world' dans un contener ubuntu
docker run ubuntu bash -c 'echo "Hello world"'
# Affiche les informations du systeme hote
cat /etc/os-release
# Affiche les informations d'un conteneur sous arch linux
docker run archlinux bash -c 'cat /etc/os-release'
```

Notez bien: Pour lancer une commande intéractive (comme une Shell), utilisez l'argument `-it`. L'argument `--rm` permet de supprimer le conteneur une fois qu'il termine.

```
docker run -it --rm archlinux bash
```

Exercice 2: Gestion de conteneurs

Notez bien: Pour voir les conteneurs en cours d'exécution `docker ps`.
Pour arrêter un conteneur `docker stop container_id`.
Notez que la commande peut prendre un certain temps à s'exécuter.

- Lancer un terminal interactif avec Docker ;
- Ouvrir un autre terminal, et inspecter les conteneurs en cours d'exécution ;
- Arrêter le conteneur Docker depuis le second terminal.

Solution: 2

```
# Dans le premier terminal
docker run -it archlinux bash

# Dans le second terminal
docker ps
docker stop id
# ou
docker rm -f id
```

Notez bien: Afin de configurer des variables d'environnement dans un conteneur, utilisez l'argument `-e`.

```
docker run -it --rm -e HELLO=hello archlinux sh -c 'echo $HELLO'
```

Notez bien: Afin d'ouvrir un port réseau, utilisez l'argument `-p docker:host`, où `docker` est le numéro du port dans le conteneur et `host` est le numéro de port de l'hôte.

```
docker run -p 127.0.0.1:8080:80 nginx
# On peut acceder au port a partir de l'url http://localhost:8080/
curl http://localhost:8080/
```

Exercice 3: Lancez une base de données PostgreSQL avec Docker.

- Le nom de l'image est `postgres` ;
- Configurez une variable d'environnement `POSTGRES_PASSWORD=postgres` ; et
- Redirigez le port 5432 du Docker vers le port 5432 de la machine hôte ;
- Laisser ce conteneur en cours d'exécution.

Solution: 3

```
docker run -e POSTGRES_PASSWORD=postgres -p 5432:5432 --rm postgres
# ou
docker run --name postgres -e POSTGRES_PASSWORD=postgres -p 5432:5432 --rm -d postgres
```

4.5 Dockerfile

Les Dockerfiles sont des fichiers de configuration utilisés pour créer des images Docker personnalisées. Ils permettent de définir de manière reproductible et automatisée l'environnement d'exécution d'une application à l'intérieur d'un conteneur Docker.

Un Dockerfile contient une série d'instructions qui spécifient les étapes nécessaires à la construction d'une image Docker. Ces instructions incluent des actions telles que la sélection de l'image de base, l'installation de dépendances, la configuration de variables d'environnement, la copie de fichiers, l'exécution de commandes et bien plus encore.

Voici un exemple annoté de Dockerfile pour une application Python.

```
# Utilise l'image de base Python 3.9 slim
FROM python:3.9-slim

# Definit le repertoire de travail a l'interieur du conteneur
# Specifie le repertoire pour les commandes RUN, CMD, ENTRYPOINT, COPY et ADD
WORKDIR /app

# Copie le fichier requirements.txt depuis la machine hote vers le conteneur Docker
COPY requirements.txt .

# Execute une commande Bash afin d'installer les dependances Python
RUN pip install -r requirements.txt

# Copie le code source dans le conteneur Docker
COPY . .

# Definit la commande par defaut lorsque le conteneur demarre
CMD ["python", "app.py"]
```

Pour construire l'image, on utilise la commande `docker build -t <tag> .`. L'argument `-t <tag>` permet de donner le nom `<tag>` à l'image créee. Par exemple, pour l'application précédente, on pourrait utiliser "`docker build -t python-app .`". Il est à noter que l'ordre des instructions est important.

Il est crucial de bien définir l'ordre des opérations dans un Dockerfile en raison de la façon dont Docker effectue le processus de construction de l'image. Chaque instruction dans le Dockerfile crée une nouvelle couche

dans l'image Docker, et l'ordre des opérations peut avoir un impact significatif sur l'efficacité et les performances de la construction de l'image.

Docker utilise un système de cache pour accélérer le processus de construction des images. Lorsque vous exécutez une instruction, Docker vérifie si cette instruction a déjà été exécutée dans une couche précédente. Si c'est le cas et que les paramètres sont identiques, Docker réutilise la couche en cache au lieu de la reconstruire. Cela permet d'économiser du temps de construction. Cependant, si vous modifiez une instruction plus haut dans le Dockerfile, toutes les instructions suivantes seront invalidées dans le cache et devront être reconstruites. Par exemple, si vous effectuez des opérations lourdes en termes de ressources, telles que la compilation de code qui changera à chaque fois que vous modifiez le code, il peut être préférable de les placer vers la fin du Dockerfile, afin de profiter du cache autant que possible.

Exercice 4: Utilisez le `README.md` du dossier `~/Applications/status-checker` afin d'écrire un Dockerfile permettant d'exécuter l'application Haskell.

Voici ce que votre Dockerfile devra faire :

- Utilisez l'image de base `haskell:9.0-buster` ;
- Exécutez `stack setup --install-ghc` ;
- Choissiez le dosser `/app` comme *workdir* ;
- Copiez le code source dans le dossier `/app` du conteneur ;
- Exécutez `stack build` ;
- Lancez le serveur avec `stack run`.

Ensuite, lancer le Docker en exposant le port 8080. Si tout est fonctionnel, vous devriez être capable d'exécuter la requête suivante :

```
curl http://localhost:8080/endpoints
# Reponse attendue
{}%
# Ou
{}
```

Laisser ce conteneur en cours d'exécution.

Solution: 4 Dockerfile :

```
FROM haskell:9.0-buster
RUN stack setup --install-ghc
WORKDIR /app
COPY . .
RUN stack build
CMD ["stack", "run"]
```

Exécution :

```
docker build -t status-checker .
docker run --rm -p 8080:8080 status-checker
```

Exercice 5: Utilisez le `README.md` du dossier `~/Applications/python_app` afin d'écrire un Dockerfile permettant d'exécuter l'application Python.
Ensuite, lancez l'application en mode interactif et avec le bon mode réseau.

> **Notez bien:** Cette application aura besoin d'accéder au réseau local de votre ordinateur pour faire des requêtes au conteneur de l'exercice précédent, pour ce faire passer l'argument `--network="host"` lorsque vous aller exécuter le conteneur.

Si tout est fonctionnel, vous devriez être capable d'utiliser l'application afin d'ajouter des URL à monitorer et de voir les résultats.

Solution: 5 Dockerfile :

```
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
CMD ["python", "main.py"]
```

Exécution :

```
docker build -t python_app .
docker run --rm -it --network="host" python_app
```

4.6 Docker avancé

4.6.1 *Construction d'image multi étape*

Les builds multi étapes (multi-stage builds) sont une fonctionnalité avancée de Docker qui permet de créer des images Docker de manière optimisée en utilisant plusieurs étapes distinctes dans le processus de construction. Cela permet de séparer les étapes de construction et de production, ce qui peut conduire à des images finales plus légères et plus sécurisées en minimisant ce qui reste dans l'image finale.

Par exemple, pour une application React en JavaScript, on peut séparer la construction en deux étapes.

Création de l'application :

```
npx create-react-app my-app
```

Dockerfile :

```
# Etape 1: Construire l'application

# Ici on utilise 'as build' pour nommer cette etape de construction
FROM node:14-alpine as build
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

# Etape 2: Preparer le serveur
# Ici, on commence une nouvelle image de zero
FROM nginx:alpine
# On copie l'application compilee depuis l'etape precedente
COPY --from=build /app/build /usr/share/nginx/html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

Pour exécuter :

```
docker build -t web-app .
docker run --rm -p 80:80 web-app
```

Exercice 6: Utilisez le `README.md` du dossier `~/Applications/rust_api` afin d'écrire un Dockerfile permettant d'exécuter l'application Python.

- Lancer la base de données PostgreSQL d'un exercice précédent ;
- Faites une première étape pour compiler le code Rust ;
- Faites une deuxième étape pour exécuter le code ;
  - Démarrer cette image à partir de `debian:bulleye-slim` ;
  - Installer le package `libpq-dev` avec la commande `apt update; apt install -y libpq-dev` ;
  - Assigner la variable d'environnement `DATABASE_URL=postgres://postgres:postgres@localhost`
  - Copiez `/app/target/release/rust_api` depuis l'étape précédente ;
- Lancer le conteneur en spécifiant `--network="host"` et exposer le bon port ;
- Tester que le tout fonctionne avec `curl http://localhost:8081/documents`.

Solution: 6

```
FROM rust:latest AS build
WORKDIR /app
COPY . .
RUN cargo build --release

FROM debian:bullseye-slim
RUN apt update; apt install -y libpq-dev
WORKDIR /app
COPY --from=build /app/target/release/rust_api ./rust_api
ENV DATABASE_URL=postgres://postgres:postgres@db
CMD ./rust_api
```

Exécution

```
docker run -e POSTGRES_PASSWORD=postgres -p 5432:5432 postgres
docker build -t rust_api .
docker run --rm --network="host" rust_api

# Valider que le serveur fonctionne
curl http://localhost:8081/documents
```

4.6.2 Volumes

En Docker, les volumes sont utilisés pour permettre aux conteneurs d'accéder, de partager et de persister des données entre le système hôte et le conteneur lui-même. Les volumes Docker permettent de stocker des données en dehors du cycle de vie des conteneurs. Cela signifie que même si vous détruisez ou recréez un conteneur, les données stockées dans le volume restent intactes. Cela permet de séparer la persistance des données de l'environnement du conteneur, offrant ainsi une meilleure gestion des données. Les volumes sont également utiles pour donner accès au conteneur à des données trop volumineuses pour être copiées à l'intérieur de l'image, comme un jeu de données d'entraînement d'un système d'apprentissage machine.

Par exemple, pour stocker les données de la base de données PostgreSQL dans un dossier de la machine hôte, on spécifie la variable `PGDATA` et on monte un volume :

```
docker run -e POSTGRES\_PASSWORD=postgres -p 5432:5432 --rm \
    -e PGDATA=/var/lib/postgresql/data/pgdata \
    -v custom/mount/path:/var/lib/postgresql/data \
    postgres
```

Afin éviter de devoir reconstruire l'image Docker après chaque modification du code source, et ainsi accélérer le développement, il est possible de rendre accessible le code source du projet à l'aide d'un volume. Ainsi, un DockerfileDev pour l'application Python pourrait ressembler à ceci :

```
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
# On a enleve la ligne COPY . .
CMD ["bash"]
```

Exécuter avec :

```
docker build -t python_app -f DockerfileDev .
docker run --rm -v .:/app my-app
# Dans le conteneur
python main.py
# On modifie le code source
python main.py
# Les changements se repercutent dans le conteneur
```

Ainsi, on peut réutiliser la même image même si le code source a changé.

4.7 Extras

4.7.1 Nettoyage

Lorsque vous utilisez Docker, il est important de prendre en compte l'espace de stockage utilisé par les images, les conteneurs, les volumes et autres artefacts Docker. Une mauvaise gestion de l'espace de stockage peut entraîner une utilisation excessive de l'espace disque et rendre difficiles la maintenance et la gestion des ressources Docker. Docker propose plusieurs commandes de `prune` afin de supprimer les conteneurs et les images non utilisées. Pour faire le ménage des différents artefacts Docker, la commande `docker system prune -a -f` supprime toutes les ressources inutilisées.

4.7.2 Docker Compose

Docker Compose est un outil qui permet de définir et de gérer facilement des applications multi conteneurs. Il simplifie le déploiement et l'orchestration des conteneurs Docker en utilisant un fichier de configuration simple et lisible.

Avec Docker Compose, vous pouvez spécifier les services, les réseaux, les volumes et autres configurations nécessaires pour exécuter une application composée de plusieurs conteneurs. Vous pouvez également définir les dépendances entre les conteneurs, les variables d'environnement, les ports exposés, etc.

Docker Compose utilise un fichier de configuration YAML pour décrire l'infrastructure de l'application. Ce fichier contient des sections telles que *services*, *networks*, *volumes*, etc., où vous pouvez définir les différentes parties de votre application et leurs configurations.

Voici un exemple de `docker-compose.yml` pour l'application Rust qui lance à la fois le server et la base de données PostgreSQL :

```
# Liste des conteneurs que l'on va lancer
services:
  # Conteur de la base de donnees
  # On retrouve les memes parametres que dans la commande ‘docker run‘
  db:
    # On specifie l'image que l'on veut lancer
    image: postgres:latest
    restart: always
    environment:
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=postgres
      - PGDATA=/var/lib/postgresql/data/pgdata
    expose:
      - 5432
    volumes:
      - db:/var/lib/postgresql/data
  # Serveur Rust
  api:
    # Va construire le Dockerfile du repertoire ou se trouve le docker-compose.yml
    build: .
    # Construit le service db avant api
    depends_on:
      - db
    ports:
      - 8081:8081
    # L'URL de la base de donnee change, plutot que @localhost
    # docker-compose rend db accessible sous le nom @db
    environment:
      - DATABASE_URL=postgres://postgres:postgres@db
    # On attend que la base de donnes soit prete avant de lancer le serveur
    entrypoint: bash -c "sleep 5 && ./rust_api"
# Necessaire pour le volume de la base de donnes
volumes:
  db:
    driver: local
```

Et on peut l'exécuter avec :

```
docker-compose up --build
```

4.7.3 Podman

Podman est une alternative open source à Docker. Docker et Podman sont deux outils populaires de conteneurisation qui partagent des fonctionnalités similaires, mais ils diffèrent dans leur architecture et leur approche de la sécurité. Docker utilise une architecture client-serveur, où le démon Docker s'exécute en tant que processus distinct et les commandes Docker sont exécutées via l'interface en ligne de commande (CLI). En revanche, Podman utilise une architecture sans démon (daemonless), ce qui signifie qu'il s'exécute directement en tant qu'utilisateur régulier et ne nécessite pas de processus démon distinct. Ainsi, Podman peut s'exécuter sans privilèges spéciaux, et ainsi limite les risques potentiels associés à l'exécution en tant que superutilisateur. [3]

Podman offre aussi des outils pour gérer des *pods*, un sujet hors de la portée de cet atelier.

4.7.4 Kubernetes

Kubernetes est un système open source d'orchestration de conteneurs qui facilite le déploiement, la gestion et la mise à l'échelle d'applications conteneurisées. L'utilisation de Kubernetes offre de nombreux avantages, tels que la tolérance aux pannes, la facilitation des déploiements, la gestion et la mise à l'échelle des applications conteneurisées.

Références

[1] Ken Cochrane, Jeeva S. Chelladhurai, and Neependra K. Khare. *Docker Cookbook : Over 100 Practical and Insightful Recipes to Build Distributed Applications with Docker, 2nd Edition*. Packt Publishing, 2nd edition, 2018. ISBN 1788626869.

[2] Coderized. Never install locally. URL https://www.youtube.com/watch?v=J0NuOlA2xDc.

[3] Red Hat. What is podman ? URL https://www.redhat.com/en/topics/containers/what-is-podman.

[4] Akash Rajpurohit. Build your own docker with linux namespaces, cgroups, and chroot : Hands-on guide. URL https://akashrajpurohit.com/blog/build-your-own-docker-with-linux-namespaces-cgroups-and-chroot-handson-guide/.