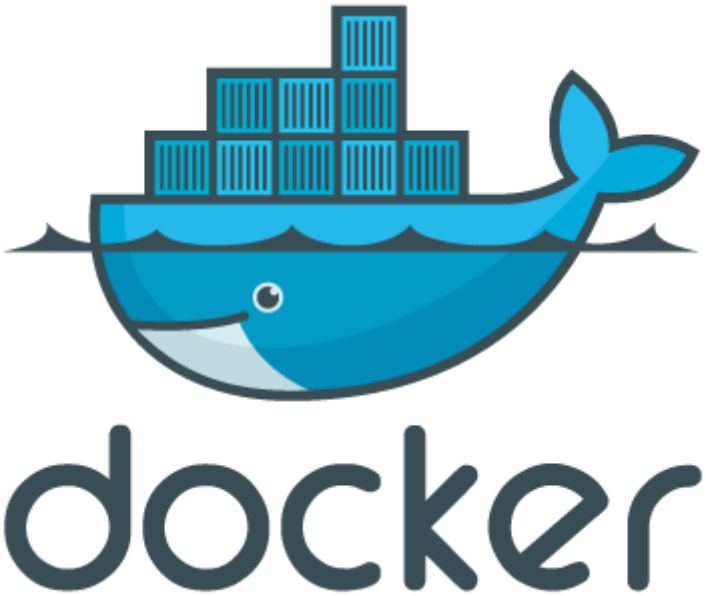
docker

Table des matières

Introduction à Docker

Pour rappel, j'avais déjà fait une introduction à Docker. Vous pourrez retrouver le document en suivant ce lien : Lecture individuelle – Dasek Joiakim, veuillez lire la section « Pour aller plus loin » qui sera le sujet de cette lecture individuelle axée pratique.

Les avancées technologiques, comme le développement agile et les pipelines d'intégration continue, ont accéléré la livraison de produits logiciels. Face à une demande croissante, de nombreuses organisations se tournent vers le cloud pour des solutions flexibles de virtualisation, réseau, et stockage, payables à l'usage.

Cependant, cette transition a engendré des défis financiers liés à la gestion continue de serveurs coûteux. Les machines virtuelles (VM) ont amélioré l'infrastructure en créant des serveurs virtuels, mais leur gestion intensive en ressources les rendait complexes à déplacer.

Pour automatiser davantage, optimiser la densité de calcul, et renforcer leur présence dans le cloud, les entreprises se dirigent vers la conteneurisation et les architectures de microservices. Les conteneurs isolent les processus, permettant l'exécution de services logiciels dans des sections isolées du noyau du système hôte, évitant ainsi le gaspillage de ressources contrairement aux VM.

Cette approche diffère de l'architecture VM traditionnelle où chaque machine est généralement dédiée à un seul serveur. La conteneurisation résout ce problème en permettant au runtime de planifier et d'exécuter des conteneurs sur le système hôte indépendamment du type d'application. Docker, apparu en 2013, se distingue en offrant non seulement l'exécution de conteneurs, mais aussi la construction et le déploiement via des référentiels, introduisant ainsi le concept d'immuabilité des conteneurs.

Docker Engine

Le Docker Engine est une interface essentielle pour exploiter les fonctionnalités d'isolation des processus du noyau Linux. Pour permettre l'exécution de conteneurs sur les hôtes Windows et macOS, qui ne supportent pas nativement ces fonctionnalités, Docker utilise une machine virtuelle Linux en arrière-plan. Les utilisateurs de Windows et macOS peuvent utiliser le package "Docker Desktop" pour déployer et exécuter cette machine virtuelle, facilitant ainsi l'utilisation des commandes Docker depuis leur terminal respectif.

Il est crucial de noter que les différences fondamentales entre les systèmes d'exploitation (Windows, macOS et Linux) peuvent entraîner des comportements différents, notamment en ce qui concerne le réseau, sur la station de travail de développement.

Le Docker Engine ne se contente pas d'exécuter des images de conteneurs, il offre également des mécanismes intégrés pour construire et tester ces images à partir de fichiers de code source appelés Dockerfiles. Une fois les images créées, elles peuvent être stockées dans des registres d'images de

conteneurs, qui servent de référentiels pour que d'autres hôtes Docker les téléchargent et les exécutent.

Lorsqu'un conteneur démarre, Docker télécharge l'image associée, la stocke localement, puis exécute la directive d'entrée du conteneur, qui représente la commande pour lancer le processus principal de l'application. Selon le type d'application, cette directive peut être un serveur démon ou un script de courte durée. Certains conteneurs exécutent également des scripts d'entrée pour configurer l'environnement avant de démarrer le processus principal. Il est recommandé de comprendre ces aspects avant d'exécuter un conteneur, en particulier pour déterminer s'il s'agit d'une exécution ponctuelle ou d'un serveur démon permanent.

Gérer les conteneurs Docker

La gestion des conteneurs Docker repose sur une série de commandes puissantes qui permettent aux utilisateurs de créer, exécuter, surveiller et gérer des applications dans des environnements isolés. Voici quelques-unes des commandes fondamentales de Docker, accompagnées d'exemples de situations simples :

1. docker run : Cette commande est utilisée pour créer et exécuter un conteneur à partir d'une image Docker. Par exemple, pour lancer un conteneur Ubuntu, on peut utiliser la commande :

```
docker run –it ubuntu
```

Cette commande démarre un conteneur Ubuntu en mode interactif (drapeau -it), permettant à l'utilisateur d'interagir directement avec le terminal du conteneur.

2. docker ps : Pour afficher la liste des conteneurs en cours d'exécution, on utilise la commande docker ps :

```
docker ps
```

Cela affiche des informations telles que l'ID du conteneur, le nom, le statut, les ports exposés, etc.

3. docker stop et docker start : Ces commandes sont utilisées pour arrêter et redémarrer un conteneur respectivement. Par exemple, pour arrêter un conteneur nommé "mon_conteneur", on peut utiliser la commande :

```
docker stop mon_conteneur
```

Et pour le redémarrer :

```
docker start mon_conteneur
```

4. docker exec : Permet d'exécuter une commande à l'intérieur d'un conteneur en cours d'exécution. Par exemple, pour ouvrir un shell bash à l'intérieur d'un conteneur nommé "mon_conteneur", on peut utiliser la commande :

```
docker exec -it mon_conteneur bash
```

5. docker build : Cette commande est utilisée pour construire une image Docker à partir d'un fichier Dockerfile. Par exemple, si le Dockerfile est dans le répertoire courant, on peut utiliser :

```
docker build -t mon_image .
```

Cela construit une image nommée "mon_image" à partir du contexte actuel (.) où se trouve le Dockerfile.

6. docker-compose : Permet de définir et gérer des applications multi-conteneurs. Un fichier docker-compose.yml décrit les services, les réseaux et les volumes nécessaires à une application. Par exemple, pour démarrer une application définie dans un fichier docker-compose.yml, on peut utiliser :

```
docker-compose up
```

Cela lance tous les services définis dans le fichier docker-compose.yml.

La gestion des conteneurs Docker offre une flexibilité et une efficacité significatives dans le déploiement et la gestion des applications. Ces commandes et concepts de base constituent une base solide pour explorer davantage les fonctionnalités avancées de Docker et optimiser la gestion des conteneurs dans divers scénarios d'utilisation.

Principes essentiels des Dockerfiles

Les Dockerfiles jouent un rôle central dans la création d'images Docker personnalisées. Ils permettent aux utilisateurs de définir de manière déclarative les étapes nécessaires à la construction d'une image, en spécifiant les dépendances, les configurations et les commandes à exécuter. Voici une explication des principes essentiels des Dockerfiles avec des exemples de situations simples :

1. FROM : La première instruction dans un Dockerfile spécifie l'image de base à utiliser. Il s'agit généralement d'une image de système d'exploitation ou d'une image de base contenant des dépendances spécifiques. Par exemple, pour utiliser une image Ubuntu 20.04 comme base :

```
FROM ubuntu:20.04
```

2. RUN : Cette instruction permet d'exécuter des commandes lors de la construction de l'image. Par exemple, pour installer des paquets dans un conteneur Ubuntu :

```
RUN apt-get update && apt-get install -y package1 package2
```

3. COPY et ADD : Ces instructions copient des fichiers depuis le système hôte vers l'image. Par exemple, pour copier tous les fichiers du répertoire local courant vers le répertoire "/app" dans l'image :

```
COPY . /app
```

4. WORKDIR : Définit le répertoire de travail pour les instructions suivantes. Toutes les commandes RUN, CMD et ENTRYPOINT seront exécutées dans ce répertoire. Par exemple, pour définir le répertoire de travail sur "/app" :

```
WORKDIR /app
```

5. CMD et ENTRYPOINT : Ces instructions spécifient la commande par défaut à exécuter lorsque le conteneur démarre. CMD est souvent utilisé pour spécifier les arguments par défaut, tandis que ENTRYPOINT est utilisé pour définir la commande principale du conteneur. Par exemple, pour lancer une application Python :

```
CMD ["python", "app.py"]
```

6. EXPOSE : Cette instruction indique les ports sur lesquels le conteneur écoutera au moment de l'exécution. Par exemple, pour exposer le port 8080 :

```
EXPOSE 8080
```

7. VOLUME : Déclare un ou plusieurs volumes pour le stockage persistant ou le partage de données entre conteneurs. Par exemple, pour créer un volume nommé "data" :

```
VOLUME /data
```

Les principes essentiels des Dockerfiles reposent sur la construction progressive d'une image à partir d'une image de base, en ajoutant des couches avec des dépendances et des configurations spécifiques. Ces principes permettent aux développeurs de créer des images Docker reproductibles et portables, simplifiant ainsi le déploiement d'applications dans des environnements variés.

Couches d'Images et Caching :

Les images Docker reposent sur un système de fichiers en couches, ce qui signifie que chaque instruction dans le Dockerfile crée une nouvelle couche. Docker utilise un mécanisme de mise en cache efficace pendant la construction des images. Si une instruction n'a pas changé depuis la dernière construction, Docker utilise la couche mise en cache plutôt que de la reconstruire, accélérant ainsi le processus de construction.

La construction d'images Docker est un processus essentiel qui transforme un Dockerfile en une image exécutable. Le processus de build s'appuie sur la séquence d'instructions du Dockerfile pour créer une image contenant tous les éléments nécessaires à l'exécution d'une application. Ce chapitre explorera le processus de build, y compris la gestion des couches, l'utilisation du cache, et des conseils pratiques pour optimiser vos builds Docker.

1. Commande de Build :

La construction d'une image à partir d'un Dockerfile s'effectue à l'aide de la commande docker build. Cette commande nécessite le chemin vers le répertoire contenant le Dockerfile. Par exemple, pour construire une image nommée "mon_app" à partir du Dockerfile dans le répertoire courant :

```
docker build -t mon_app .
```

L'option -t permet de taguer l'image avec un nom convivial.

2. Processus de Layering :

Chaque instruction dans un Dockerfile crée une nouvelle couche dans l'image. Ces couches sont empilées les unes sur les autres, et chacune représente une modification apportée à l'image. L'utilisation de layers permet la réutilisation et l'optimisation du processus de build.

3. Utilisation du Cache :

Docker utilise un mécanisme de mise en cache pendant le processus de build pour accélérer les opérations répétitives. Si une instruction n'a pas changé depuis la dernière build, Docker utilise la couche mise en cache plutôt que de la reconstruire. Cela est particulièrement utile lors de l'installation de dépendances ou de la copie de fichiers.

Exemple Pratique de Build :

Reprenons l'exemple du Dockerfile pour une application Flask en Python. Pour construire l'image, nous utilisons la commande suivante :

```
docker build -t mon_app .
```

Chaque ligne du Dockerfile générera une nouvelle couche dans l'image. Les couches sont stockées de manière efficace, ce qui permet une gestion optimale des modifications apportées à l'application.

Conseils Pratiques :

Évitez de surcharger vos images avec des couches inutiles. Regroupez les commandes pour minimiser le nombre de layers.

Placez les instructions les moins susceptibles de changer vers le haut du Dockerfile pour maximiser l'utilisation du cache.

Utilisez des images de base légères pour réduire la taille finale de votre image.

Les directives VOLUME et EXPOSE sont deux aspects importants des fichiers Dockerfile qui permettent de définir des configurations liées à la gestion des données et des ports réseau dans un conteneur Docker.

1. Directive VOLUME :

La directive VOLUME est utilisée pour déclarer un ou plusieurs volumes dans un conteneur Docker. Un volume est un emplacement de stockage persistant qui existe en dehors du cycle de vie du conteneur. Cela signifie que même si le conteneur est arrêté ou supprimé, les données stockées dans un volume persisteront.

Exemple :

```
VOLUME /data
```

Cette instruction crée un volume nommé /data dans le conteneur. Vous pouvez également spécifier un chemin de montage sur le système hôte pour le volume, par exemple :

```
VOLUME /var/www/html
```

Cela monte le volume /var/www/html du conteneur vers le chemin correspondant sur le système hôte.

L'utilisation de volumes est courante pour stocker des données telles que les bases de données, les fichiers de configuration, ou d'autres données nécessaires à l'application.

2. Directive EXPOSE :

La directive EXPOSE informe Docker qu'un conteneur écoute sur les ports spécifiés lors de son exécution. Cependant, cette directive n'ouvre pas effectivement ces ports ni ne les mappe sur le système hôte. Elle est principalement documentaire et sert à informer les utilisateurs sur les ports sur lesquels l'application à l'intérieur du conteneur écoute.

Exemple :

```
EXPOSE 80/tcp
```

```
EXPOSE 443/tcp
```

Ces instructions indiquent que le conteneur expose les ports 80 et 443 en mode TCP. Si vous souhaitez réellement mapper ces ports sur le système hôte lors du lancement du conteneur, vous devrez utiliser l'option -p lors de la commande docker run.

```
docker run -p 8080:80 -p 8443:443 mon_app
```

Cela mappe le port 80 du conteneur sur le port 8080 du système hôte et le port 443 du conteneur sur le port 8443 du système hôte.

Informations importantes :

La directive EXPOSE dans un Dockerfile n'effectue pas la publication directe des ports sur le système hôte. Elle sert principalement à documenter les ports sur lesquels l'application à l'intérieur du conteneur écoute. Même si vous utilisez EXPOSE dans votre Dockerfile, vous devrez spécifier explicitement le mappage des ports lors de l'exécution du conteneur avec la commande docker run pour permettre la communication avec le système hôte.

Si vous ne spécifiez pas explicitement le mappage des ports avec docker run -p, les conteneurs peuvent toujours communiquer entre eux, mais pas directement avec le système hôte sur les ports exposés. En d'autres termes, la communication entre conteneurs sur le même réseau Docker peut se faire sans le mappage explicite des ports, mais l'accès depuis l'extérieur du conteneur (par exemple, depuis le système hôte) nécessitera le mappage des ports.

Si deux conteneurs doivent communiquer entre eux sur des ports exposés, ils peuvent le faire en utilisant le réseau interne de Docker. Chaque conteneur dans le même réseau Docker peut atteindre les autres conteneurs sur les ports exposés sans nécessiter de mappage explicite.

La directive HEALTHCHECK dans un fichier Dockerfile permet de spécifier une commande qui sera exécutée périodiquement pour déterminer si le conteneur est en bonne santé. Cela offre une manière automatisée de vérifier l'état d'un service à l'intérieur du conteneur. Le health check est utile pour les orchestrations de conteneurs, les outils de gestion de conteneurs, et pour garantir la disponibilité et la fiabilité d'une application.

Exemple :

```
HEALTHCHECK --interval=30s --timeout=10s --retries=3 CMD curl -f http://localhost/ || exit 1
```

Dans cet exemple, la commande curl -f http://localhost/ est exécutée toutes les 30 secondes avec un timeout de 10 secondes et 3 tentatives possibles avant de déclarer le conteneur en échec. Si la commande échoue (par exemple, si la requête HTTP n'aboutit pas), le conteneur est considéré comme en mauvaise santé.

Options de la directive HEALTHCHECK :

--interval : Spécifie l'intervalle entre deux exécutions du health check.

--timeout : Définit le délai d'attente maximal pour l'exécution du health check.

--retries : Définit le nombre de tentatives avant de déclarer le health check en échec.

CMD : La commande à exécuter pour le health check.

Utilisation dans le contexte de Docker :

Lorsque vous construisez une image avec un health check, cela n'affecte pas le démarrage initial du conteneur. Cependant, une fois le conteneur en cours d'exécution, l'orchestrateur Docker ou tout autre outil de gestion peut utiliser le health check pour surveiller la santé du conteneur.

Exemple de Démarrage d'un Conteneur avec Health Check :

```
docker run --health-cmd="curl -f http://localhost/" --health-interval=30s --health-timeout=10s --health-retries=3 mon_app
```

Cela spécifie les mêmes paramètres de health check que dans le Dockerfile lors du démarrage du conteneur.

Gestion et création des images Docker

Nous avons abordé ci-dessus une thématique qu'est la performance pour le build, nous allons pouvoir approfondir cela.

Build rapide et réduction des couches

Optimiser la vitesse de construction des images Docker implique de regrouper judicieusement les commandes afin de minimiser le nombre de couches. Cela peut être réalisé en combinant des commandes similaires dans une seule instruction pour réduire le nombre total de couches.

Exemple :

```
FROM ubuntu:20.04

RUN apt-get update && \
    apt-get install -y \
    package1 \
    package2


COPY . /app
```

```
CMD ["python", "app.py"]
```

Dans cet exemple, les commandes `apt-get update` et `apt-get install` sont combinées en une seule instruction pour réduire le nombre de couches.

The Scratch Image

Parfois, il est nécessaire de créer une image à partir de zéro, sans aucun système d'exploitation de base. C'est là que l'image scratch entre en jeu. Elle est vide, offrant une toile vierge pour construire des images minimales.

Exemple:

```
FROM scratch

ADD my_app /

CMD ["/my_app"]
```

Cette image de base vide peut être utilisée pour créer des conteneurs ultralégers.

Docker Image Naming et Tagging

La nomination et le marquage des images Docker sont des pratiques essentielles pour organiser et gérer efficacement vos images, facilitant ainsi le suivi des versions, la distribution et le déploiement des applications. Voici quelques détails approfondis sur la manière de nommer et de marquer vos images Docker :

Nommer vos Images Docker

Conventions de Nommage :

Adopter des conventions de nommage claires facilite la compréhension du contenu et de l'utilisation prévue de l'image. Il est recommandé d'utiliser des noms en minuscules et de séparer les mots par des tirets bas (underscore) ou des tirets.

```
mon_app
```

Utilisation de Préfixes :

L'utilisation de préfixes dans les noms d'images peut aider à organiser vos images en fonction de projets, d'équipes ou de technologies spécifiques.

```
mon_projet/mon_app
```

Nommage basé sur la Version :

Il est judicieux d'inclure la version de l'application ou de l'image dans le nom pour faciliter le suivi des changements.

```
mon_app:1.0
```

Taguer vos Images Docker

Utilisation de Tags :

Les tags sont des références spécifiques à une version particulière de l'image. Utiliser des tags clairs et significatifs facilite le suivi des versions et le déploiement.

```
mon_app:latest
```

Dans cet exemple, le tag latest est souvent utilisé pour indiquer la dernière version stable de l'image.

Tags basés sur des Commits ou des Hashes Git :

Pour une traçabilité précise, certains développeurs préfèrent utiliser des tags basés sur des commits ou des hash Git spécifiques.

```
mon_app:commit_sha
```

Cela peut être utile pour revenir à une version spécifique de l'application.

Tags basés sur des Environnements :

Vous pouvez également utiliser des tags pour différencier les images destinées à différents environnements, tels que dev, test, et prod.

```
mon_app:dev
```

Docker Image Tagging Policies

La mise en place de politiques de marquage (tagging) pour les images Docker est essentielle pour assurer une gestion efficace des versions, la traçabilité des changements et la facilité de déploiement. Voici des détails approfondis sur la mise en place de politiques de marquage pour vos images Docker

Conserver et publier ses images Docker

Le stockage et la publication d'images sont essentiels pour partager vos applications avec d'autres développeurs et pour les déployer dans des environnements de production.

Exemple:

```
docker login

docker tag mon_app:latest mon_utilisateur/mon_app:latest

docker push mon_utilisateur/mon_app:latest
```

Ces commandes vous connectent à Docker Hub, taguent votre image, puis la poussent sur le registre.

Dockerfiles à plusieurs étapes

L'un des aspects puissants de Docker réside dans sa capacité à créer des images Docker, qui sont des packages auto-suffisants contenant tout le nécessaire pour exécuter une application, y compris le code, les bibliothèques, les dépendances, et même le système d'exploitation.

Construire normale d'image

Dans une construction Docker classique, vous créez une image en spécifiant une séquence d'instructions dans un fichier appelé Dockerfile. Cependant, cela peut entraîner la création d'images volumineuses, car tous les outils de construction et les dépendances sont inclus, même s'ils ne sont nécessaires que pendant la phase de construction.

Lorsque vous utilisez le processus de construction normal, le Dockerfile contient des instructions pour installer toutes les dépendances, compiler le code source, et configurer l'environnement d'exécution. Cependant, cela peut aboutir à une image finale plus grande que nécessaire.

Qu'est-ce que le Builder Pattern ?

Le modèle du constructeur (Builder Pattern) est une approche qui consiste à utiliser plusieurs étapes de construction pour optimiser la taille de l'image finale. Il repose sur l'idée de diviser la construction en plusieurs étapes logiques, permettant ainsi de séparer les outils de construction et les dépendances du produit final.

Avec le modèle du constructeur, les outils de construction et les dépendances nécessaires à la compilation du code sont isolés dans une première étape, tandis que la deuxième étape utilise uniquement les artefacts de compilation nécessaires pour exécuter l'application. Cela réduit considérablement la taille de l'image finale.

Construire une image Docker avec la stratégie multi-stage

Dans cet exercice, nous explorerons la création d'une image Docker en utilisant un processus de construction multi-étapes. Cela nous permettra de mettre en pratique le modèle du constructeur pour optimiser la taille de notre image Docker.

Exemple :

Supposons maintenant que nous ayons une application Java avec Maven à construire en utilisant un processus de construction multi-étapes. Voici un exemple de Dockerfile pour cela :

```
# Stage 1: Build Stage

FROM maven:3.8.4-openjdk-11 AS builder

WORKDIR /app

# Copier le fichier pom.xml pour télécharger les dépendances

COPY pom.xml .

RUN mvn dependency:go-offline

# Copier le reste du code source et compiler l'application

COPY src src

RUN mvn package

# Stage 2: Production Stage

FROM openjdk:11-jre-slim

WORKDIR /app

# Copier uniquement le JAR construit depuis la première étape

COPY --from=builder /app/target/my-application.jar .

# Définir la commande de démarrage de l'application

CMD ["java", "-jar", "my-application.jar"]
```

Dans cet exemple, la première étape (builder) utilise une image Maven pour construire l'application Java. Elle télécharge les dépendances, copie le code source, et crée le JAR. La deuxième étape utilise une image plus légère (openjdk:11-jre-slim) et copie seulement le JAR construit depuis la première étape, réduisant ainsi la taille de l'image finale.

Docker Compose

Docker Compose est un outil qui permet de définir et de gérer des applications multi-conteneurs Docker. Il simplifie le processus de déploiement d'applications en utilisant un fichier de configuration YAML pour décrire les services, les réseaux et les volumes nécessaires. Avec Docker Compose, vous pouvez déployer toute votre application avec une seule commande, ce qui facilite le développement, le test et le déploiement.

Docker compose CLI

La ligne de commande Docker Compose (CLI) offre un ensemble de commandes pour interagir avec les applications définies dans le fichier docker-compose.yml. Ces commandes permettent de construire, démarrer, arrêter et gérer les conteneurs, ainsi que de visualiser l'état des services définis dans le fichier de configuration. Voici un ensemble non exhaustif de commandes Docker compose :

1. `docker-compose version`

Description : Affiche la version de Docker Compose installée.

2. `docker-compose up`

Description : Construit et démarre les services définis dans le fichier docker-compose.yml.

3. `docker-compose up -d`

Description : Démarre les services en arrière-plan (en mode détaché).

4. `docker-compose down`

Description : Arrête et supprime les conteneurs, les réseaux et les volumes associés aux services.

5. `docker-compose ps`

Description : Affiche l'état des services.

6. `docker-compose logs [service]`

Description : Affiche les journaux des services. Vous pouvez spécifier un service particulier en remplaçant [service].

7. `docker-compose exec [service] [command]`

Description : Exécute une commande dans le conteneur du service spécifié.

8. `docker-compose build`

Description : Construit ou reconstruit les images des services.

9. `docker-compose stop [service]`

Description : Arrête un service spécifique.

10. `docker-compose restart [service]`

Description : Redémarre un service spécifique.

11. `docker-compose down -v`

Description : Arrête et supprime les conteneurs, réseaux, volumes, et images associés aux services.

Docker-compose.yml

Un exemple simple du fichier :

```
version: '3'

services:

  web:

    image: nginx:latest

    ports:

      - "80:80"

  app:

    build:

      context: ./my-app

    ports:

      - "5000:5000"

    depends_on:

      - db

  db:

    image: postgres:latest

    environment:

      POSTGRES_DB: mydatabase
```

Bibliographie

Docker. (s.d.). *https://docs.docker.com/*. Récupéré sur https://docs.docker.com/: https://docs.docker.com/

Joiakim, D. (2022). *Docker*. Récupéré sur https://jowadev.tech/: https://jowadev.tech/S2/Artefact/LI/Joiakim/Docker

Sesto, V. •. (2020). *The Docker Workshop*. Packt Publishing.