Gérer des containers

Table des matières

Module 1 : La connaissance des techniques de virtualisation basées sur les conteneurs

La virtualisation est un concept clé dans la gestion des infrastructures modernes, permettant de faire fonctionner plusieurs systèmes ou applications indépendants sur une même machine physique. Dans ce module, nous allons explorer les techniques de virtualisation basées sur les **conteneurs**, en mettant l'accent sur la manière dont ils diffèrent des autres formes de virtualisation, leurs avantages, et leur architecture sous-jacente.

1.1 Introduction à la virtualisation et aux conteneurs

La **virtualisation** permet de créer des instances indépendantes d'un système ou d'une application sur une machine physique. Traditionnellement, cela se fait avec des machines virtuelles (VM), mais les **conteneurs** sont une approche plus légère et plus moderne de la virtualisation.

1.1.1 Virtualisation traditionnelle (Machines virtuelles)

Les **machines virtuelles (VM)** reposent sur un hyperviseur, qui est une couche logicielle permettant de créer et de gérer des VMs. Chaque VM fonctionne comme une machine physique complète, avec son propre système d'exploitation (OS), ses bibliothèques, et ses dépendances. Cette approche crée une certaine redondance, car chaque VM a besoin de son propre noyau système et d'un OS complet, ce qui peut devenir gourmand en ressources.

- **Avantages des VMs** :
  - Forte isolation des applications.
  - Chaque VM peut exécuter son propre OS, permettant de tester différentes configurations.
- **Inconvénients des VMs** :
  - **Consommation de ressources** : Chaque VM inclut un OS complet, ce qui entraîne des frais supplémentaires en termes de mémoire et de CPU.
  - **Démarrage lent** : Les VMs démarrent généralement plus lentement en raison de la surcharge liée à l'OS complet et au processus de virtualisation.

1.1.2 Virtualisation basée sur les conteneurs

Les **conteneurs** sont une forme de virtualisation **légère** qui fonctionne au-dessus du noyau de l'OS hôte, en utilisant des mécanismes de noyau comme **cgroups** (contrôle de ressources) et **namespaces** (isolation des processus). Contrairement aux VMs, les conteneurs ne nécessitent pas un système d'exploitation complet pour chaque instance. Ils partagent le noyau de l'OS hôte tout en étant isolés les uns des autres, ce qui les rend plus rapides et plus efficaces.

- **Avantages des conteneurs** :
  - **Légèreté** : Un conteneur contient uniquement les éléments nécessaires à l'exécution de l'application (code, dépendances, bibliothèques, etc.), ce qui le rend plus léger que les VMs.
  - **Démarrage rapide** : Les conteneurs démarrent beaucoup plus rapidement que les VMs, car il n'y a pas de système d'exploitation complet à charger.
  - **Portabilité** : Un conteneur peut être exécuté de manière identique sur n'importe quelle machine, tant que Docker ou un autre moteur de conteneurs est installé.
- **Inconvénients des conteneurs** :
  - **Moins d'isolation** que les VMs : Les conteneurs partagent le même noyau, ce qui signifie qu'ils sont potentiellement moins isolés les uns des autres que les VMs.
  - **Sécurité** : Bien que les conteneurs offrent un bon niveau d'isolation, ils sont plus vulnérables que les VMs à certaines attaques, notamment si des vulnérabilités sont présentes dans le noyau.

1.2 Architecture des conteneurs

Les conteneurs reposent sur un certain nombre de technologies sous-jacentes pour fournir un environnement isolé. La principale technologie utilisée est le **noyau Linux**, bien que des solutions similaires existent pour d'autres systèmes d'exploitation, comme Windows.

1.2.1 Namespaces

Les **namespaces** sont une fonctionnalité du noyau Linux qui permet d'isoler les ressources entre les différents conteneurs. Chaque conteneur peut avoir ses propres vues indépendantes des autres conteneurs sur le système d'exploitation.

- **Types de namespaces utilisés par les conteneurs** :
  - **PID namespace** : Permet d'isoler les processus en cours d'exécution, chaque conteneur ayant sa propre table des processus.
  - **Network namespace** : Chaque conteneur a sa propre pile réseau (interfaces, routage, etc.).
  - **Mount namespace** : Chaque conteneur peut avoir son propre système de fichiers virtuel.
  - **UTS namespace** : Permet à chaque conteneur de disposer de son propre nom d'hôte.
  - **IPC namespace** : Permet d'isoler les communications inter-processus (IPC) entre les conteneurs.
  - **User namespace** : Isole les utilisateurs et groupes, permettant à un conteneur de fonctionner avec des identifiants d'utilisateur et de groupe différents de ceux de l'hôte.

1.2.2 Cgroups (Control Groups)

Les **cgroups** permettent de limiter et de contrôler l'utilisation des ressources (CPU, mémoire, I/O, etc.) par chaque conteneur. Ils assurent qu'un conteneur ne consomme pas plus de ressources que ce qui lui a été alloué, garantissant ainsi une gestion plus efficace des ressources.

- **Exemples d'utilisation des cgroups** :
  - Limiter la quantité de mémoire qu'un conteneur peut utiliser.
  - Limiter la bande passante réseau utilisée par un conteneur.
  - Limiter le nombre de processus qu'un conteneur peut créer.

1.2.3 Images et Conteneurs

- **Image** : Une image Docker est une version figée d'une application et de son environnement d'exécution. Elle contient tout ce qu'il faut pour exécuter une application (fichiers binaires, bibliothèques, variables d'environnement, etc.). Les images sont utilisées pour créer des conteneurs.
- **Conteneur** : Un conteneur est une instance en cours d'exécution d'une image Docker. Il est créé à partir de l'image et peut exécuter une application ou un service.

Les images sont immutables, ce qui signifie qu'une fois créées, elles ne peuvent pas être modifiées. Si des modifications sont nécessaires, une nouvelle image doit être construite. Cette immutabilité contribue à la portabilité des conteneurs.

1.3 Avantages de la virtualisation par conteneurs par rapport aux autres approches

1.3.1 Meilleure utilisation des ressources

Les conteneurs utilisent beaucoup moins de ressources que les machines virtuelles, car ils partagent le noyau de l'OS de l'hôte au lieu de dupliquer le système d'exploitation pour chaque instance. Cela permet de créer un plus grand nombre de conteneurs sur une même machine physique, ce qui optimise l'utilisation des ressources.

1.3.2 Portabilité et cohérence

Les conteneurs garantissent que l'application fonctionnera de manière cohérente, peu importe où elle est déployée. Cela signifie que les développeurs peuvent tester leur application sur leur machine locale et être sûrs qu'elle fonctionnera de la même manière en production.

1.3.3 Développement rapide et flexible

Les conteneurs sont particulièrement adaptés aux environnements de développement et de test, car ils permettent de configurer des environnements d'exécution complexes de manière rapide et cohérente. Ils facilitent également l'intégration continue et la livraison continue (CI/CD).

1.4 Cas d'utilisation des conteneurs

1.4.1 Microservices

Les conteneurs sont particulièrement adaptés à l'architecture des **microservices**, où chaque service peut être déployé et mis à l'échelle indépendamment des autres. Les conteneurs offrent un moyen pratique de gérer ces services légers, qui peuvent communiquer entre eux tout en étant isolés.

1.4.2 Environnements de test et de développement

Les développeurs peuvent utiliser des conteneurs pour créer des environnements d'exécution identiques à ceux de production, garantissant ainsi qu'il n'y a pas de "fonctionne sur ma machine" dans le cycle de développement.

1.4.3 Déploiement d'applications en production

Les conteneurs peuvent être déployés sur n'importe quel serveur, ce qui facilite les déploiements dans des environnements distribués. Cela est souvent réalisé via des orchestrateurs comme **Kubernetes**, qui permettent de gérer des clusters de conteneurs.

Module 2 : L'outil Docker : principes, objectifs et solutions

Docker est l'un des outils les plus populaires pour la gestion des conteneurs, permettant de créer, déployer et exécuter des applications dans des environnements conteneurisés. Ce module explore les principes fondamentaux de Docker, ses objectifs, ainsi que les solutions qu'il offre pour la gestion des conteneurs.

2.1 Introduction à Docker

Docker est une plateforme open-source qui permet aux développeurs de créer, déployer et exécuter des applications dans des conteneurs. Un conteneur Docker est une unité exécutable légère et portable qui inclut tout ce qui est nécessaire pour exécuter une application : le code, les bibliothèques, les dépendances et l'environnement d'exécution.

Docker simplifie le processus de gestion des conteneurs en automatisant de nombreuses tâches complexes associées à la création et à l'exécution de conteneurs. Il utilise une approche de conteneurisation qui est plus efficace et plus légère que les solutions de virtualisation traditionnelles.

2.1.1 Prérequis à Docker

Avant de pouvoir utiliser Docker, vous devez avoir un système qui prend en charge la conteneurisation. Docker fonctionne principalement sur **Linux**, mais il est également disponible pour **Windows** et **macOS** via des machines virtuelles ou des adaptateurs comme Docker Desktop.

2.2 Principes de Docker

Les principes de Docker reposent sur des concepts de base qui facilitent la gestion des applications dans des environnements conteneurisés. Comprendre ces principes est essentiel pour utiliser Docker efficacement.

2.2.1 Conteneurisation

Le principe de base de Docker est **la conteneurisation** : exécuter des applications dans des conteneurs qui partagent le noyau du système d'exploitation sous-jacent, mais restent isolés les uns des autres. Contrairement à une machine virtuelle (VM) qui nécessite un OS complet pour chaque instance, un conteneur Docker est beaucoup plus léger et démarre plus rapidement.

- **Conteneurisation légère** : Docker partage le noyau du système hôte, ce qui rend la gestion des ressources plus efficace par rapport aux machines virtuelles.
- **Isolation** : Les conteneurs sont isolés les uns des autres, ce qui permet d'éviter les conflits d'environnement et de dépendances entre les applications.

2.2.2 Image Docker

Une **image Docker** est un modèle immuable, une version figée de l'environnement d'exécution d'une application, comprenant le système d'exploitation, les dépendances, les configurations et le code de l'application. Les images sont utilisées pour créer des conteneurs.

- **Immutabilité** : Une image est une version en lecture seule de l'application et de son environnement. Si vous souhaitez modifier une image, vous devez en créer une nouvelle.
- **Portabilité** : Une image Docker peut être déplacée et déployée sur n'importe quelle machine disposant de Docker, garantissant une cohérence entre les environnements de développement, de test et de production.

2.2.3 Conteneur Docker

Un **conteneur Docker** est une instance en cours d'exécution d'une image Docker. Contrairement à une image, un conteneur peut être modifié (par exemple, en enregistrant des fichiers générés à l'exécution). Cependant, dès qu'un conteneur est arrêté, il est détruit, à moins qu'il ne soit configuré pour conserver des données dans des volumes externes.

- **Démarrage rapide** : Les conteneurs démarrent en quelques secondes, ce qui permet un développement et un déploiement rapides.
- **Évolutivité** : Les conteneurs sont idéaux pour les applications qui nécessitent une mise à l'échelle dynamique, comme les applications microservices.

2.3 Objectifs de Docker

Docker est conçu pour résoudre plusieurs défis associés à la gestion d'applications dans des environnements modernes et distribués.

2.3.1 Portabilité et Consistance

Docker permet de garantir que les applications s'exécutent de manière cohérente sur n'importe quel environnement, qu'il s'agisse de votre machine locale, d'un serveur de développement ou d'un environnement de production. L'utilisation de conteneurs assure que l'application et ses dépendances sont toujours livrées dans le même état.

- **Exemple** : Une application Python peut être développée, testée et exécutée dans un conteneur sur un ordinateur portable, puis déployée de manière cohérente dans un cluster de serveurs de production sans modification.

2.3.2 Isolation des Environnements

Docker offre une isolation complète des environnements d'exécution. Cela signifie que plusieurs applications peuvent être exécutées sur la même machine sans interférer entre elles. Cela simplifie la gestion des dépendances et des conflits d'environnement.

- **Exemple** : Vous pouvez exécuter plusieurs versions d'une base de données MySQL dans des conteneurs séparés sans qu'elles ne se perturbent.

2.3.3 Meilleure Gestion des Ressources

Les conteneurs partagent le noyau du système d'exploitation sous-jacent, mais chacun d'eux est isolé et peut être limité en termes de ressources (CPU, mémoire, etc.). Docker vous permet de configurer ces limites pour éviter qu'un conteneur ne consomme trop de ressources et ne ralentisse les autres conteneurs ou l'hôte.

- **Exemple** : Vous pouvez allouer un nombre spécifique de cœurs CPU et de mémoire à chaque conteneur afin de mieux gérer les ressources du serveur.

2.3.4 Amélioration du Développement et de l'Automatisation

Docker facilite la création d'environnements reproductibles et permet une gestion automatisée du cycle de vie des applications grâce à des outils comme **Docker Compose** et **Docker Swarm**. L'automatisation du déploiement de conteneurs est un atout majeur dans les pratiques de **DevOps** et de **CI/CD (intégration continue et déploiement continu)**.

- **Exemple** : Utiliser des scripts de déploiement pour déployer automatiquement de nouvelles versions d'une application en production.

2.4 Solutions Docker

Docker fournit une série de solutions pour faciliter la gestion des conteneurs et des applications conteneurisées, notamment pour le développement, le déploiement et l'orchestration des conteneurs.

2.4.1 Docker Engine

Le **Docker Engine** est le moteur sous-jacent qui permet de créer, déployer et exécuter des conteneurs Docker. Il se compose de trois parties :

1. **Le serveur Docker** : Un démon qui exécute les conteneurs.
2. **L'API Docker** : Fournit une interface pour interagir avec le serveur Docker.
3. **L'interface en ligne de commande Docker (CLI)** : Permet aux utilisateurs d'exécuter des commandes Docker pour gérer les conteneurs et les images.

2.4.2 Docker Hub

Docker Hub est un service de registre de conteneurs qui permet de partager et de stocker des images Docker. Il offre une large bibliothèque d'images publiques, mais il est également possible d'héberger des images privées.

- **Docker Hub** permet de partager facilement des images entre différentes équipes et de réutiliser des images préexistantes.
- **Exemple** : Vous pouvez trouver une image officielle de **MySQL** ou de **Node.js** sur Docker Hub et l'utiliser dans vos projets.

2.4.3 Docker Compose

Docker Compose est un outil permettant de définir et de gérer des applications multi-conteneurs. Au lieu de lancer chaque conteneur manuellement, Docker Compose vous permet de définir toute l'application dans un fichier YAML. Il gère les interactions entre les différents conteneurs et les dépendances.

- **Exemple** : Un projet comprenant une application web et une base de données peut être défini dans un fichier `docker-compose.yml`, qui décrit les conteneurs nécessaires et la manière dont ils interagissent.

2.4.4 Docker Swarm et Kubernetes

Pour gérer des applications conteneurisées à grande échelle, **Docker Swarm** et **Kubernetes** sont des solutions d'orchestration populaires. Elles permettent de gérer un cluster de conteneurs en production, d'assurer la haute disponibilité, la mise à l'échelle automatique et le déploiement continu.

- **Docker Swarm** : Une solution d'orchestration native à Docker qui permet de gérer un cluster de conteneurs.
- **Kubernetes** : Une solution d'orchestration plus avancée, souvent utilisée pour des environnements de production complexes. Kubernetes est compatible avec Docker et offre des fonctionnalités plus poussées en termes de gestion des conteneurs à grande échelle.

2.4.5 Docker Volumes

Les **volumes Docker** sont utilisés pour persister les données générées et utilisées par les conteneurs. Contrairement aux conteneurs, qui sont éphémères, les volumes permettent de conserver les données même lorsque le conteneur est supprimé ou redémarré.

- **Exemple** : Utiliser un volume Docker pour stocker les données d'une base de données exécutée dans un conteneur afin de ne pas perdre les données lors du redémarrage du conteneur.

Module 3 : Le Dockerfile et ses instructions

Le **Dockerfile** est un fichier de configuration essentiel dans l'écosystème Docker. Il contient une série d'instructions qui permettent de définir une image Docker de manière déclarative. Ce module détaillera les différentes instructions utilisées dans un Dockerfile, leur rôle et leur utilisation pour construire des images Docker efficaces et optimisées.

3.1 Introduction au Dockerfile

Un **Dockerfile** est un fichier texte contenant une série d'instructions qui permettent de construire une image Docker. Il décrit toutes les étapes nécessaires pour assembler une image à partir d'une base existante, installer des logiciels, copier

des fichiers et définir les commandes qui seront exécutées lors de l'exécution du conteneur. Le Dockerfile est essentiel pour automatiser la création d'images Docker reproductibles et cohérentes.

Exemple basique de Dockerfile :

```
# Utilisation de l'image de base officielle Python
FROM python:3.8-slim

# Définition du répertoire de travail à l'intérieur du conteneur
WORKDIR /app

# Copie du fichier de dépendances
COPY requirements.txt .

# Installation des dépendances Python
RUN pip install -r requirements.txt

# Copie du code source de l'application
COPY . .

# Commande à exécuter lorsque le conteneur démarre
CMD ["python", "app.py"]
```

Dans cet exemple, le Dockerfile crée une image à partir d'une image de base Python, installe les dépendances et copie les fichiers nécessaires pour exécuter l'application Python.

3.2 Les Instructions du Dockerfile

Voici les instructions les plus courantes utilisées dans un Dockerfile, avec des exemples et explications pour chacune d'elles.

3.2.1 FROM

L'instruction **FROM** définit l'image de base sur laquelle l'image Docker sera construite. Il s'agit de l'instruction de départ pour chaque Dockerfile. Une image Docker peut être construite à partir d'une image officielle (par exemple `python:3.8-slim`), d'une image d'une autre application, ou d'une image personnalisée créée par l'utilisateur.

Exemple :

Dockerfile
```
FROM node:14
```

Cela indique que l'image Docker sera construite à partir de l'image officielle Node.js version 14.

3.2.2 WORKDIR

L'instruction **WORKDIR** définit le répertoire de travail dans lequel les instructions suivantes seront exécutées. Si le répertoire n'existe pas, il sera créé.

Exemple :

```
Dockerfile
WORKDIR /app
```

Cela définit `/app` comme répertoire de travail dans le conteneur. Toutes les instructions suivantes (comme `COPY` ou `RUN`) seront exécutées à partir de ce répertoire.

3.2.3 COPY

L'instruction **COPY** permet de copier des fichiers ou des répertoires depuis le système de fichiers de l'hôte vers le système de fichiers du conteneur. C'est une manière courante de transférer des fichiers nécessaires à l'application dans le conteneur.

Exemple :

```
Dockerfile
COPY . .
```

Cela copie tout le contenu du répertoire actuel (`.`) sur l'hôte vers le répertoire de travail dans le conteneur (`.` également, selon l'instruction `WORKDIR`).

3.2.4 ADD

L'instruction **ADD** est similaire à **COPY**, mais elle est plus puissante car elle permet de décompresser des archives (tar, zip, etc.) et de télécharger des fichiers depuis une URL. Cependant, son utilisation doit être limitée pour ne pas rendre l'image Docker inutilement complexe.

Exemple :

```
Dockerfile
ADD https://example.com/app.tar.gz /app/
```

Cela télécharge l'archive depuis l'URL et la décompresse dans le répertoire `/app/` du conteneur.

3.2.5 RUN

L'instruction **RUN** permet d'exécuter des commandes dans le conteneur lors de la construction de l'image. Ces commandes peuvent être utilisées pour installer des logiciels, mettre à jour des paquets, ou effectuer toute autre action de configuration.

Exemple :

```
Dockerfile
RUN apt-get update && apt-get install -y curl
```

Cette commande met à jour la liste des paquets et installe `curl` dans le conteneur.

3.2.6 CMD

L'instruction **CMD** définit la commande par défaut à exécuter lorsque le conteneur est démarré. Cette commande peut être une commande shell ou un programme. Il existe trois formes pour spécifier un **CMD** : la forme shell (par défaut), la forme exécutable, et la forme JSON.

Exemple :

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

Cela signifie que, lorsque le conteneur démarre, il exécutera `python app.py`.

3.2.7 ENTRYPOINT

L'instruction **ENTRYPOINT** est similaire à **CMD**, mais elle est utilisée pour spécifier un point d'entrée fixe pour le conteneur. Contrairement à **CMD**, l'instruction **ENTRYPOINT** ne peut pas être remplacée lors de l'exécution du conteneur, mais peut être combinée avec **CMD** pour ajouter des arguments par défaut.

Exemple :

```
Dockerfile
ENTRYPOINT ["python"]
CMD ["app.py"]
```

Cela signifie que le conteneur exécutera toujours `python`, et le fichier `app.py` sera passé comme argument par défaut.

3.2.8 EXPOSE

L'instruction **EXPOSE** permet de documenter les ports sur lesquels le conteneur écoutera. Cela ne publie pas le port, mais c'est utile pour la documentation et pour l'usage avec des outils d'orchestration de conteneurs comme Docker Compose.

Exemple :

```
Dockerfile
EXPOSE 80
```

Cela indique que le conteneur écoute sur le port 80, ce qui est souvent le cas pour les applications web.

3.2.9 ENV

L'instruction **ENV** permet de définir des variables d'environnement dans le conteneur. Ces variables peuvent être utilisées pour configurer l'application ou pour stocker des informations sensibles comme des clés d'API (bien qu'il soit préférable d'utiliser des solutions comme Docker Secrets pour cela).

Exemple :

```
Dockerfile
ENV APP_ENV=production
```

Cela définit la variable d'environnement `APP_ENV` avec la valeur `production`.

3.2.10 VOLUME

L'instruction **VOLUME** permet de créer un point de montage pour les volumes dans le conteneur. Les volumes sont des zones de stockage persistantes qui survivent au redémarrage des conteneurs.

Exemple :

Dockerfile
```
VOLUME /data
```

Cela crée un volume sur le répertoire `/data` du conteneur, permettant ainsi de stocker des données persistantes.

3.2.11 ARG

L'instruction **ARG** permet de définir des variables d'argument qui peuvent être utilisées lors de la construction de l'image Docker. Elles sont définies avec la commande `docker build --build-arg` et sont accessibles pendant la phase de construction.

Exemple :

Dockerfile
```
ARG VERSION=1.0
RUN echo "Version $VERSION"
```

Cela permet de définir un argument `VERSION` lors de la construction de l'image.

3.3 Exemple de Dockerfile Complet

Voici un exemple d'un Dockerfile complet pour une application Node.js :

```
# Étape 1 : Utiliser une image de base
FROM node:14

# Étape 2 : Définir le répertoire de travail
WORKDIR /usr/src/app

# Étape 3 : Copier le fichier package.json et installer les dépendances
COPY package*.json ./
RUN npm install

# Étape 4 : Copier le code source
COPY . .

# Étape 5 : Exposer le port que l'application va utiliser
EXPOSE 8080

# Étape 6 : Définir la commande de démarrage
CMD ["node", "server.js"]
```

Module 4 : Docker Compose : Introduction

Docker Compose est un outil qui permet de définir et de gérer des applications multi-conteneurs Docker. En d'autres termes, il vous permet de configurer et de gérer plusieurs conteneurs Docker qui interagissent ensemble au sein d'une même application, à partir d'un seul fichier de configuration. Docker Compose simplifie le processus de déploiement et de gestion des applications complexes, tout en améliorant la portabilité et la reproductibilité des environnements.

4.1 Introduction à Docker Compose

Docker Compose utilise un fichier de configuration, généralement nommé **docker-compose.yml**, pour définir l'ensemble des services, réseaux et volumes nécessaires au bon fonctionnement de l'application. Cela permet de gérer facilement plusieurs conteneurs Docker en les définissant de manière déclarative.

Principe de fonctionnement :

- Vous définissez les services qui composent votre application.
- Chaque service est une définition de conteneur, qui peut inclure une image Docker, un réseau, un volume de stockage et une configuration spécifique.
- Avec une simple commande `docker-compose up`, Compose crée et exécute tous les conteneurs définis dans le fichier **docker-compose.yml**.

4.2 Pourquoi utiliser Docker Compose ?

Docker Compose offre plusieurs avantages pour les développeurs et les équipes DevOps, notamment :

1. **Simplification du déploiement d'applications multi-conteneurs** : Compose permet de décrire une application complète (composée de plusieurs services, bases de données, API, etc.) dans un fichier unique, facilitant ainsi la gestion de l'infrastructure.
2. **Isolation des environnements** : Chaque service dans Docker Compose peut être isolé dans son propre conteneur, ce qui permet de séparer clairement les différentes parties de l'application.
3. **Portabilité** : Le fichier **docker-compose.yml** peut être partagé et utilisé par d'autres développeurs ou dans des environnements de production. Cela permet de garantir que l'application fonctionne de la même manière sur différentes machines et configurations.
4. **Automatisation** : En utilisant Docker Compose, vous pouvez automatiser la création, l'exécution, la mise à l'échelle et la gestion de votre application multi-conteneurs avec une seule commande.
5. **Tests** : Docker Compose facilite la configuration d'environnements de test ou de staging répliquant exactement l'environnement de production.

4.3 Structure d'un fichier `docker-compose.yml`

Un fichier **docker-compose.yml** contient des informations sous forme de YAML pour décrire les services de l'application, leurs configurations et la manière dont ils interagissent. Voici la structure de base d'un fichier `docker-compose.yml` :

```yaml
version: "3.8"  # Spécifie la version de Compose

services:
  web:
    image: nginx:latest  # Spécifie l'image du conteneur
    ports:
      - "8080:80"  # Mappe le port 80 du conteneur au port 8080 de la machine hôte
    networks:
      - front-end  # Associe le service à un réseau nommé "front-end"

  db:
    image: postgres:latest  # Spécifie l'image du conteneur
    environment:
      POSTGRES_USER: user  # Variables d'environnement pour la configuration de la base de donnée
      POSTGRES_PASSWORD: password
    volumes:
      - db-data:/var/lib/postgresql/data  # Volumes pour persister les données
    networks:
      - back-end  # Associe le service à un réseau nommé "back-end"

networks:
  front-end: {}  # Définition du réseau "front-end"
  back-end: {}  # Définition du réseau "back-end"

volumes:
  db-data: {}  # Volume nommé "db-data" pour persister les données de la base
```

Explications du fichier :

- **version** : Indique la version de Docker Compose que vous utilisez. La version détermine les fonctionnalités disponibles dans votre fichier **docker-compose.yml**.
- **services** : Cette section définit les conteneurs qui composent votre application. Chaque service est un conteneur, et vous pouvez définir plusieurs services dans le même fichier Compose.
    - **web** : Il s'agit d'un service qui utilise l'image **nginx:latest** (un serveur web Nginx). Le port 8080 de l'hôte est lié au port 80 du conteneur.
    - **db** : Un autre service représentant une base de données PostgreSQL. Il utilise l'image **postgres:latest** et définit des variables d'environnement pour la configuration du mot de passe et de l'utilisateur. Les données sont stockées dans un volume nommé **db-data**.
- **networks** : Déclare les réseaux sur lesquels les services peuvent communiquer. Les services peuvent être associés à un ou plusieurs réseaux, ce qui permet de contrôler leur communication.
- **volumes** : Permet de définir des volumes persistants qui sont utilisés pour stocker des données. Dans cet exemple, un volume **db-data** est utilisé pour persister les données de la base de données PostgreSQL, afin de ne pas les perdre lorsque le conteneur est arrêté ou redémarré.

4.4 Commandes principales de Docker Compose

Docker Compose offre plusieurs commandes qui permettent de gérer les applications multi-conteneurs. Voici quelques-unes des commandes les plus utilisées :

- `docker-compose up` : Cette commande crée et démarre les conteneurs définis dans le fichier **docker-compose.yml**. Si les conteneurs n'ont pas encore été créés, Compose les construira automatiquement.

Vous pouvez ajouter l'option `-d` pour exécuter les conteneurs en mode détaché (en arrière-plan).

```
docker-compose up -d
```

- **`docker-compose down`** : Cette commande arrête les conteneurs et supprime les ressources (réseaux, volumes, etc.) créées par `docker-compose up`.
- **`docker-compose build`** : Si vous avez des services qui nécessitent une construction d'image (par exemple, un Dockerfile personnalisé), cette commande permet de construire les images avant de démarrer les conteneurs.
- **`docker-compose logs`** : Cette commande permet de visualiser les logs des services en cours d'exécution.
- **`docker-compose ps`** : Affiche l'état des conteneurs en cours d'exécution, indiquant si tout fonctionne correctement.
- **`docker-compose stop`** : Cette commande arrête les conteneurs sans les supprimer, ce qui permet de les redémarrer plus tard sans avoir à reconstruire ou à relancer.

4.5 Exemple d'application multi-conteneurs avec Docker Compose

Voici un exemple d'un projet plus complexe qui utilise Docker Compose pour définir une application web avec un serveur web (Nginx), une base de données (MySQL), et un backend (Node.js).

docker-compose.yml :

```yaml
version: '3'

services:
  web:
    image: nginx:latest
    ports:
      - "8080:80"
    volumes:
      - ./html:/usr/share/nginx/html
    networks:
      - app-network

  backend:
    build: ./backend
    environment:
      - DB_HOST=db
      - DB_USER=root
      - DB_PASS=password
    networks:
      - app-network

  db:
    image: mysql:5.7
    environment:
      - MYSQL_ROOT_PASSWORD=password
      - MYSQL_DATABASE=appdb
    volumes:
      - db-data:/var/lib/mysql
    networks:
      - app-network

networks:
  app-network:
    driver: bridge

volumes:
  db-data:
```

Explication :

- Le service **web** utilise une image Nginx et expose le port 80 à l'hôte via le port 8080. Il monte le répertoire local `./html` dans le conteneur pour servir des fichiers HTML.
- Le service **backend** est un service personnalisé qui sera construit à partir du répertoire `./backend`. Il utilise des variables d'environnement pour se connecter à la base de données **db**.
- Le service **db** utilise l'image **mysql:5.7**, et une base de données MySQL est initialisée avec un mot de passe root et une base de données nommée **appdb**.

4.6 Conclusion du Module

Docker Compose est un outil puissant qui permet de gérer des applications complexes composées de plusieurs conteneurs. Grâce à sa capacité à définir, configurer et orchestrer des services dans un environnement multi-conteneurs, Docker Compose simplifie le développement, les tests et le déploiement d'applications. Son fichier de configuration **docker-compose.yml** permet de décrire l'ensemble de l'application de manière déclarative, facilitant ainsi la portabilité et la reproductibilité de l'environnement.

Module 5 : Docker Compose : Étude de cas

Dans ce module, nous allons explorer un cas d'utilisation concret de **Docker Compose**. Cette étude de cas vous permettra de comprendre comment configurer et déployer une application multi-conteneurs en utilisant **Docker Compose**. Nous allons créer un environnement pour une application web avec une interface front-end, un serveur back-end et une base de données, le tout orchestré avec Docker Compose.

5.1 Objectifs de l'étude de cas

L'objectif de cette étude de cas est de vous guider dans la mise en place d'une application à trois niveaux (front-end, back-end, base de données) à l'aide de Docker Compose. Les services seront connectés entre eux par des réseaux Docker et persisteront leurs données dans des volumes Docker pour garantir la résilience des données. À la fin de cette étude, vous serez en mesure de :

- Créer des services Docker pour une application web complète.
- Utiliser Docker Compose pour gérer et interconnecter plusieurs conteneurs.
- Déployer une application multi-conteneurs avec une base de données persistante et un environnement front-end/back-end.

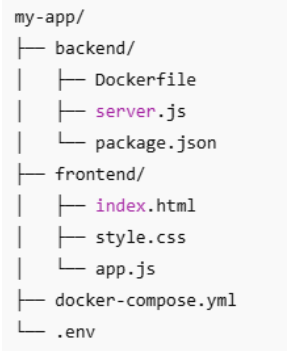
5.2 Architecture de l'application

L'architecture de l'application que nous allons créer est simple mais représente une architecture courante pour une application web :

- **Front-end** : Une application statique HTML/CSS/JavaScript servant de page web.
- **Back-end** : Un serveur API (ici, un serveur Node.js) qui gère la logique métier et les communications avec la base de données.
- **Base de données** : Une base de données MySQL pour stocker les informations des utilisateurs et autres données de l'application.

5.3 Création de l'application

5.3.1 Structure du projet

Voici la structure du projet que nous allons créer pour cette étude de cas	

- **backend/** : Contient les fichiers du serveur Node.js.
- **frontend/** : Contient les fichiers statiques du front-end (HTML, CSS, JS).
- **docker-compose.yml** : Le fichier Docker Compose pour définir les services et leurs interactions.
- **.env** : Un fichier pour stocker les variables d'environnement comme les informations de connexion à la base de données.

5.3.2 Définir le back-end (Node.js)

Dans le répertoire **backend/**, nous avons un serveur Node.js simple. Voici le contenu de chaque fichier dans ce répertoire :

- **package.json** :

```
{
  "name": "backend",
  "version": "1.0.0",
  "main": "server.js",
  "dependencies": {
    "express": "^4.17.1",
    "mysql2": "^2.3.3"
  },
  "scripts": {
    "start": "node server.js"
  }
}
```

- **server.js** :

```
const express = require('express');
const mysql = require('mysql2');
const app = express();
const port = 3000;

// Connexion à la base de données MySQL
const db = mysql.createConnection({
  host: process.env.DB_HOST,
  user: process.env.DB_USER,
  password: process.env.DB_PASS,
  database: process.env.DB_NAME
});

db.connect((err) => {
  if (err) throw err;
  console.log('Connected to the database!');
});

// Exemple d'API pour récupérer des utilisateurs
app.get('/users', (req, res) => {
  db.query('SELECT * FROM users', (err, results) => {
    if (err) throw err;
    res.json(results);
  });
});

// Démarrer le serveur
app.listen(port, () => {
  console.log(`Back-end server running on http://localhost:${port}`);
});
```

Dans ce serveur, nous avons un point d'API `/users` qui interroge la base de données MySQL et retourne les utilisateurs enregistrés.

5.3.3 Définir le front-end (HTML + JavaScript)

Le répertoire **frontend/** contient les fichiers pour le front-end. Le front-end est une simple page HTML qui communique avec l'API back-end.

- **index.html** :

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>My App</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <h1>Users</h1>
  <div id="users-list"></div>

  <script src="app.js"></script>
</body>
</html>
```

- **app.js** :

```javascript
window.onload = () => {
  fetch('http://localhost:3000/users')
    .then((response) => response.json())
    .then((data) => {
      const usersList = document.getElementById('users-list');
      data.forEach(user => {
        const userElement = document.createElement('p');
        userElement.textContent = `ID: ${user.id}, Name: ${user.name}`;
        usersList.appendChild(userElement);
      });
    });
};
```

Dans ce fichier JavaScript, une requête est effectuée pour récupérer les utilisateurs depuis l'API back-end. Les données sont ensuite affichées dans le navigateur.

5.4 Docker Compose : Définir les services

Maintenant, nous allons créer le fichier **docker-compose.yml** pour orchestrer l'ensemble des services. Ce fichier décrit les trois services : front-end, back-end et base de données, ainsi que la manière dont ils interagissent.

Voici le contenu du fichier **docker-compose.yml** :

```yaml
version: "3.8"

services:
  frontend:
    build: ./frontend
    ports:
      - "80:80"
    networks:
      - app-network

  backend:
    build: ./backend
    environment:
      DB_HOST: db
      DB_USER: root
      DB_PASS: example
      DB_NAME: appdb
    ports:
      - "3000:3000"
    depends_on:
      - db
    networks:
      - app-network

  db:
    image: mysql:5.7
    environment:
      MYSQL_ROOT_PASSWORD: example
      MYSQL_DATABASE: appdb
    volumes:
      - db-data:/var/lib/mysql
    networks:
      - app-network

networks:
  app-network:
    driver: bridge

volumes:
  db-data:
```

Explication :

- **frontend** : Utilise un répertoire local **./frontend** pour construire l'image de l'application front-end. Il expose le port 80 et le connecte au réseau `app-network`.
- **backend** : Utilise **./backend** pour construire l'image du serveur Node.js. Il se connecte à la base de données via les variables d'environnement et expose le port 3000. Le service **backend** dépend du service **db**, garantissant que le serveur back-end ne démarrera qu'après que la base de données soit prête.
- **db** : Utilise l'image officielle de MySQL 5.7. Les variables d'environnement définissent les informations de connexion à la base de données. Un volume **db-data** est monté pour persister les données.

5.5 Démarrer l'application avec Docker Compose

Une fois que tous les fichiers sont prêts, vous pouvez démarrer l'application multi-conteneurs avec la commande suivante :

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

Cela va :

- Construire les images Docker pour le front-end et le back-end.
- Lancer les services et les conteneurs dans l'ordre défini dans le fichier **docker-compose.yml**.
- Créer les réseaux et volumes nécessaires.

Une fois l'application lancée, vous pouvez accéder à l'interface front-end en ouvrant un navigateur et en allant sur `http://localhost`. Vous devriez voir la liste des utilisateurs récupérée depuis l'API back-end.

Module 6 : L'automatisation de la création des containers avec un outil de type Docker

Dans ce module, nous allons explorer comment automatiser la création, le déploiement et la gestion des conteneurs Docker à l'aide d'outils d'automatisation. Bien que **Docker** lui-même offre une puissance de gestion manuelle via des commandes, il existe des outils supplémentaires qui permettent d'automatiser le processus de création et de gestion des conteneurs, ce qui est essentiel pour les environnements de production et pour améliorer l'efficacité dans des configurations complexes.

6.1 Pourquoi automatiser la création des conteneurs ?

L'automatisation de la création des conteneurs est cruciale pour plusieurs raisons :

1. **Cohérence** : Assurer que chaque conteneur est configuré de manière identique sur toutes les machines. Cela réduit les risques d'erreurs humaines et garantit que l'application fonctionne de la même manière partout.
2. **Gain de temps** : En automatisant le processus de création et de gestion des conteneurs, vous pouvez déployer des environnements complexes de manière rapide et reproductible.
3. **Scalabilité** : L'automatisation permet de facilement gérer des environnements à grande échelle avec de nombreux conteneurs, réduisant ainsi la complexité de l'administration système.
4. **Maintenance et mise à jour** : Automatiser les mises à jour et la maintenance des conteneurs permet d'éviter les erreurs humaines lors de l'application de changements dans des environnements complexes.

6.2 Outils d'automatisation pour Docker

Il existe plusieurs outils pour automatiser la gestion des conteneurs Docker. Nous allons nous concentrer sur deux des plus populaires : **Docker Compose** et **Docker Swarm**.

6.2.1 Docker Compose pour l'automatisation

Docker Compose est un outil qui permet de définir et d'exécuter des applications Docker multi-conteneurs avec un simple fichier de configuration (**docker-compose.yml**). En plus de l'orchestration de plusieurs conteneurs, Docker Compose peut également être utilisé pour automatiser le processus de création de conteneurs.

Exemple de Docker Compose automatisant le déploiement : Si vous avez une application avec plusieurs services, tels que front-end, back-end et base de données, vous pouvez automatiser leur création et exécution en définissant un fichier `docker-compose.yml` :

```yaml
version: '3.8'

services:
  web:
    image: nginx:latest
    ports:
      - "8080:80"
    volumes:
      - ./html:/usr/share/nginx/html
    networks:
      - app-network

  backend:
    build: ./backend
    environment:
      - DB_HOST=db
      - DB_USER=root
      - DB_PASS=password
    ports:
      - "3000:3000"
    depends_on:
      - db
    networks:
      - app-network

  db:
    image: mysql:5.7
    environment:
      MYSQL_ROOT_PASSWORD: password
      MYSQL_DATABASE: appdb
    volumes:
      - db-data:/var/lib/mysql
    networks:
      - app-network

networks:
  app-network:
    driver: bridge

volumes:
  db-data:
```

L'exécution d'une commande comme `docker-compose up --build` lancera tous les services, construira les images nécessaires et démarrera les conteneurs associés de manière automatique.

6.2.2 Docker Swarm pour l'automatisation et l'orchestration à grande échelle

Docker Swarm est une solution d'orchestration native de Docker, qui permet de gérer un cluster de machines exécutant Docker. Il s'agit d'un mode de fonctionnement de Docker qui facilite la gestion et le déploiement des conteneurs à grande échelle, ce qui est essentiel pour les environnements de production.

Swarm Mode vous permet de :

- Créer un cluster de machines (nœuds) qui partagent les ressources pour exécuter des services Docker.
- Déployer et mettre à l'échelle des services (conteneurs) sur plusieurs hôtes avec un contrôle centralisé.
- Automatiser le processus de répartition des conteneurs et de gestion des états souhaités (par exemple, maintenir un certain nombre d'instances d'un service actif).

Exemple de création d'un service avec Docker Swarm :

1. **Initialiser Docker Swarm** : Pour démarrer un cluster Swarm, vous devez d'abord initialiser Swarm sur une machine :

   ```
   docker swarm init
   ```

   Cette commande transforme votre machine en un "manager" dans un cluster Swarm.

2. **Déployer un service Docker dans Swarm** : Vous pouvez ensuite déployer un service dans Swarm en spécifiant le nombre d'instances du service que vous souhaitez :

   ```
   docker service create --name my-web-service --replicas 3 -p 8080:80 nginx
   ```

   Ici, un service appelé **my-web-service** est créé avec **3 réplicas** (instances de conteneurs). Ce service expose le port 8080 sur l'hôte, mappé au port 80 du conteneur.

3. **Mettre à l'échelle le service** : Pour mettre à l'échelle un service dans Docker Swarm (augmenter ou réduire le nombre de réplicas), vous pouvez utiliser la commande suivante :

   ```
   docker service scale my-web-service=5
   ```

   Cela fera passer le nombre d'instances de **my-web-service** à **5**.

4. **Mettre à jour un service dans Swarm** : Si vous souhaitez mettre à jour l'image d'un service déployé sur Swarm, vous pouvez utiliser la commande suivante, qui mettra à jour le service avec une nouvelle version de l'image :

   ```
   docker service update --image nginx:latest my-web-service
   ```

   Docker Swarm gère automatiquement la mise à jour et le déploiement des nouveaux conteneurs sans interrompre les services.

6.2.3 Outils complémentaires pour l'automatisation

En plus de Docker Compose et Docker Swarm, il existe des outils tiers et des plateformes d'automatisation qui peuvent être utilisés pour gérer les conteneurs Docker à grande échelle ou pour des déploiements continus. Ces outils incluent :

- **Kubernetes** : Un système d'orchestration de conteneurs plus avancé que Docker Swarm. Il permet de gérer des clusters de machines, de déployer des applications, et de maintenir l'état souhaité des services à grande échelle. Kubernetes est particulièrement adapté aux environnements complexes et aux déploiements dans le cloud.
- **Ansible** : Un outil d'automatisation qui permet de gérer des configurations, déployer des applications et orchestrer des conteneurs Docker.
- **Jenkins** : Un outil d'intégration continue qui peut être utilisé pour automatiser le processus de construction, de test et de déploiement de conteneurs Docker dans des pipelines CI/CD.

6.3 Automatisation du déploiement des applications avec CI/CD et Docker

Une autre façon d'automatiser la gestion des conteneurs Docker est d'utiliser un pipeline d'intégration continue (CI) et de déploiement continu (CD). En intégrant Docker avec des outils comme **Jenkins**, **GitLab CI**, ou **GitHub Actions**, vous pouvez automatiser la création, la construction et le déploiement de conteneurs à chaque mise à jour de votre code source.

Exemple avec Jenkins :

1. **Définir un fichier Dockerfile** pour l'application.
2. **Créer un pipeline Jenkins** pour surveiller votre dépôt de code source.
3. **Automatiser la construction de l'image Docker** à chaque commit et pousser l'image dans un registre Docker.
4. **Déployer l'image sur un cluster Docker Swarm** ou Kubernetes automatiquement.

Cela permet d'implémenter un processus totalement automatisé où le code est continuellement intégré, testé, puis déployé dans des conteneurs Docker sur des environnements de production.

6.4 Avantages de l'automatisation de la création des conteneurs

1. **Reproductibilité** : Chaque déploiement de conteneur peut être fait de manière identique, avec les mêmes configurations et le même environnement, ce qui réduit le risque de divergence entre les environnements de développement, de test et de production.
2. **Gain de temps** : L'automatisation accélère le processus de création, de déploiement et de mise à l'échelle des conteneurs, ce qui permet de se concentrer sur des tâches à plus forte valeur ajoutée.
3. **Maintenance simplifiée** : Vous pouvez facilement mettre à jour, redémarrer ou modifier des services de manière transparente, ce qui rend la gestion des environnements plus efficace.
4. **Scalabilité** : Les systèmes comme Docker Swarm et Kubernetes permettent de gérer un grand nombre de conteneurs sans intervention manuelle, en répondant automatiquement aux besoins de mise à l'échelle.

Module 7 : L'utilisation des conteneurs pour gérer les mises à jour applicatives

Dans ce module, nous allons examiner comment les conteneurs Docker peuvent être utilisés pour gérer efficacement les mises à jour applicatives, un aspect crucial pour assurer une livraison continue et éviter les interruptions lors des mises à jour des applications en production.

Les conteneurs sont idéaux pour gérer les mises à jour de manière transparente, rapide et fiable grâce à leur capacité à encapsuler l'application et toutes ses dépendances dans un environnement isolé. Cela permet non seulement de simplifier le processus de mise à jour, mais aussi de réduire les risques associés aux changements dans l'environnement.

7.1 Pourquoi utiliser les conteneurs pour gérer les mises à jour applicatives ?

Les mises à jour d'applications dans un environnement de production peuvent entraîner des interruptions de service et des risques de régression si elles ne sont pas correctement gérées. Les conteneurs Docker, cependant, offrent plusieurs avantages pour automatiser et sécuriser ce processus :

- **Isolation** : Chaque conteneur contient son application et toutes ses dépendances, ce qui permet d'éviter les conflits entre les mises à jour de différentes applications.
- **Portabilité** : Les conteneurs peuvent être déployés et mis à jour de manière cohérente sur différents environnements (local, test, production), garantissant une compatibilité parfaite.
- **Rollback rapide** : Si une mise à jour échoue, il est possible de revenir rapidement à une version précédente en redéployant un conteneur avec l'ancienne version de l'application.
- **Scalabilité** : Les conteneurs facilitent la mise à l'échelle des applications et la gestion des mises à jour de manière automatisée sans affecter l'ensemble de l'application.

7.2 Stratégies de mise à jour des applications avec Docker

Dans le contexte des conteneurs Docker, plusieurs stratégies peuvent être utilisées pour gérer les mises à jour des applications. Ces stratégies permettent de garantir la continuité du service, même lors des déploiements ou des mises à jour.

7.2.1 Mise à jour par remplacement d'image

La stratégie la plus courante pour mettre à jour une application dans un conteneur est de créer une nouvelle image Docker, de déployer un nouveau conteneur avec cette image et de supprimer l'ancien conteneur.

Étapes pour la mise à jour par remplacement d'image :

1. **Mettre à jour le code source de l'application** dans le dépôt.
2. **Construire une nouvelle image Docker** à partir du Dockerfile mis à jour.

   ```
   docker build -t my-app:v2 .
   ```

3. **Arrêter et supprimer l'ancien conteneur** :

   ```
   docker stop my-app-container
   docker rm my-app-container
   ```

4. **Démarrer un nouveau conteneur** avec la nouvelle image.

   ```
   docker run -d --name my-app-container my-app:v2
   ```

Cette méthode garantit que l'application sera mise à jour avec la nouvelle version sans interférer avec les versions en cours. Toutefois, elle nécessite un temps d'arrêt pendant le remplacement des conteneurs.

7.2.2 Stratégie de mise à jour en continu (Rolling Updates)

Une autre approche plus avancée est la mise à jour continue ou **rolling update**. Cette stratégie consiste à mettre à jour progressivement les instances de l'application tout en maintenant les autres instances en fonctionnement pour ne pas interrompre le service.

Exemple avec Docker Swarm : Docker Swarm permet de déployer des mises à jour continues (rolling updates) de manière automatique pour vos services. Voici comment procéder :

1. **Créer un service avec plusieurs réplicas** dans Docker Swarm :

   ```
   docker service create --name my-app --replicas 3 my-app:v1
   ```

   Cela crée un service avec 3 réplicas de votre application.

2. **Mettre à jour le service avec une nouvelle image** : Vous pouvez maintenant mettre à jour votre service de manière transparente en utilisant la commande suivante :

   ```
   docker service update --image my-app:v2 my-app
   ```

   Docker Swarm déploiera progressivement la nouvelle version de l'application, en remplaçant les anciens réplicas avec la nouvelle image tout en maintenant une partie de l'application en ligne pendant la mise à jour.

3. **Surveiller le processus de mise à jour** : Docker Swarm gère la mise à jour de manière à garantir qu'un minimum d'instances de l'application est toujours en fonctionnement, ce qui réduit les risques de panne.

7.2.3 Blue/Green Deployment

La méthode **Blue/Green Deployment** est une stratégie de mise à jour très populaire, qui permet de minimiser les risques en préparant une nouvelle version de l'application (la version **green**) tout en maintenant l'ancienne version (la version **blue**) en production. Une fois la nouvelle version prête, vous basculez tout le trafic vers la nouvelle version.

Étapes du Blue/Green Deployment :

1. **Déployer l'ancienne version** (Blue) : L'application en cours d'exécution représente la version stable de l'application.

   ```
   docker run -d --name my-app-blue my-app:v1
   ```

2. **Déployer la nouvelle version** (Green) en parallèle de l'ancienne version : Une nouvelle version de l'application (avec un nouveau tag d'image) est déployée dans un conteneur différent.

   ```
   docker run -d --name my-app-green my-app:v2
   ```

3. **Basculez le trafic vers la version Green** : Une fois que la nouvelle version est prête, vous pouvez rediriger le trafic réseau vers la version **green**.

   ```
   docker stop my-app-blue
   docker rename my-app-green my-app-blue
   ```

   Cette approche garantit que si un problème survient avec la nouvelle version, vous pouvez revenir rapidement à la version stable (Blue) sans interruption de service.

7.2.4 Canary Deployment

Le **Canary Deployment** est une approche similaire au Blue/Green Deployment, mais au lieu de déployer la nouvelle version à 100%, vous déployez d'abord une petite portion du trafic (un "canari") vers la nouvelle version et surveillez son comportement. Si la version fonctionne correctement, vous augmentez progressivement le pourcentage du trafic dirigé vers cette nouvelle version.

Exemple de Canary Deployment avec Docker Swarm :

1. Déployer la nouvelle version avec une réplique réduite :

   ```
   docker service update --image my-app:v2 --replicas 1 my-app
   ```

2. Surveiller les logs et les performances de l'application. Si la version se comporte bien, augmenter progressivement le nombre de réplicas.
3. Une fois que vous êtes sûr de la stabilité de la nouvelle version, mettez à jour tous les réplicas pour pointer vers la version la plus récente :

   ```
   docker service update --replicas 3 --image my-app:v2 my-app
   ```

7.3 Automatisation des mises à jour avec CI/CD

Les outils d'intégration continue (CI) et de déploiement continu (CD) comme **Jenkins**, **GitLab CI**, ou **GitHub Actions** peuvent être utilisés pour automatiser le processus de mise à jour des conteneurs. Cela vous permet d'automatiser la construction de nouvelles images Docker, les tests et le déploiement sur des environnements de production.

Voici comment cela pourrait fonctionner dans un pipeline CI/CD :

1. **Commit de code** : Vous poussez des modifications dans votre dépôt Git.
2. **Build Docker** : Le pipeline CI commence à construire une nouvelle image Docker à partir du Dockerfile mis à jour.
3. **Tests automatisés** : Des tests sont exécutés sur la nouvelle image Docker pour garantir que l'application fonctionne correctement.

4. **Push à un registre Docker** : Si les tests réussissent, l'image Docker mise à jour est poussée vers un registre (par exemple Docker Hub ou un registre privé).
5. **Déploiement en production** : L'image mise à jour est déployée sur votre infrastructure, selon l'une des stratégies de mise à jour que nous avons couvertes (Blue/Green, Canary, Rolling Update, etc.).

7.4 Gestion des versions et des rétrogradations

L'un des avantages clés des conteneurs Docker est la gestion facile des versions et la possibilité de revenir à une version antérieure si une mise à jour échoue. Grâce à Docker, vous pouvez :

- **Conserver plusieurs versions** d'une application en utilisant des tags différents pour les images Docker.
- **Revenir à une version précédente** en redéployant une ancienne image.

Par exemple, si la mise à jour vers la version `v2` échoue, il suffit de revenir à la version `v1` :

```
docker service update --image my-app:v1 my-app
```

Module 8 : Le paquetage d'une application Python sous forme de conteneur

Dans ce module, nous allons apprendre à empaqueter une application Python dans un conteneur Docker, en suivant les meilleures pratiques. L'objectif est de créer un environnement d'exécution portable et reproductible pour votre application Python, ce qui permet de déployer l'application sur n'importe quel environnement Docker compatible sans se soucier des configurations spécifiques de la machine hôte.

8.1 Pourquoi empaqueter une application Python dans un conteneur ?

Le paquetage d'une application Python sous forme de conteneur présente plusieurs avantages :

1. **Portabilité** : Un conteneur Docker permet de garantir que l'application fonctionne de manière cohérente sur tous les environnements, que ce soit en développement, en test, ou en production.
2. **Isolation** : Docker isole votre application et ses dépendances du système d'exploitation hôte, ce qui réduit les risques de conflits entre versions ou configurations différentes.
3. **Facilité de déploiement** : Vous pouvez déployer votre application Python dans n'importe quel environnement Docker compatible avec un simple fichier de configuration (Dockerfile).
4. **Scalabilité et gestion des dépendances** : Docker vous permet de gérer les dépendances de votre application facilement et de mettre à l'échelle des instances de votre application Python sans problème.

8.2 Prérequis avant de commencer

Avant de commencer à empaqueter une application Python dans un conteneur Docker, vous devez avoir :

- **Docker installé** sur votre machine.
- Une **application Python** prête à être conteneurisée.
- Un fichier **requirements.txt** qui contient toutes les dépendances nécessaires pour votre application Python.

8.3 Créer un Dockerfile pour une application Python

Le Dockerfile est un fichier de script qui contient une série d'instructions permettant de créer une image Docker. Nous allons examiner comment rédiger un Dockerfile pour une application Python simple.

8.3.1 Structure d'un Dockerfile pour une application Python

Voici un exemple de Dockerfile pour une application Python :

```
# Utiliser une image de base Python officielle
FROM python:3.9-slim

# Définir le répertoire de travail dans le conteneur
WORKDIR /app

# Copier le fichier requirements.txt dans le conteneur
COPY requirements.txt .

# Installer les dépendances à partir de requirements.txt
RUN pip install --no-cache-dir -r requirements.txt

# Copier tout le code source de l'application dans le conteneur
COPY . .

# Exposer le port que l'application va utiliser
EXPOSE 5000

# Définir la commande à exécuter pour démarrer l'application
CMD ["python", "app.py"]
```

Explication des instructions :

1. **FROM python:3.9-slim** : Cette ligne spécifie l'image de base à utiliser pour construire l'image Docker. Ici, nous utilisons l'image officielle Python avec la version 3.9 et une version "slim" qui est plus légère.
2. **WORKDIR /app** : Cette ligne définit le répertoire de travail dans le conteneur. Cela signifie que toutes les commandes suivantes seront exécutées dans le répertoire `/app`.
3. **COPY requirements.txt .** : Cela copie le fichier `requirements.txt` de votre machine locale dans le répertoire de travail du conteneur.
4. **RUN pip install --no-cache-dir -r requirements.txt** : Cette instruction installe toutes les dépendances Python nécessaires à partir du fichier `requirements.txt`. L'option `--no-cache-dir` permet de ne pas conserver les fichiers de cache de pip, ce qui rend l'image plus légère.
5. **COPY . .** : Cette instruction copie tous les fichiers de votre répertoire local dans le répertoire de travail du conteneur (c'est-à-dire `/app`).
6. **EXPOSE 5000** : L'application écoute le port 5000, c'est donc ce port que nous exposons pour que le conteneur puisse être accessible à partir de l'extérieur. Ce port doit correspondre à celui que votre application utilise pour communiquer.
7. **CMD ["python", "app.py"]** : La commande qui sera exécutée lorsque le conteneur démarre. Ici, nous lançons le fichier `app.py` avec Python.

8.3.2 Créer un fichier requirements.txt

Le fichier `requirements.txt` contient la liste des dépendances Python de votre application. Voici un exemple de ce fichier :

```
txt
Flask==2.0.1
requests==2.25.1
```

Lorsque vous exécutez `pip freeze > requirements.txt`, vous obtenez la liste de toutes les dépendances nécessaires à votre projet Python.

8.4 Construire et exécuter l'image Docker

Une fois que vous avez écrit votre Dockerfile, vous pouvez créer l'image Docker et exécuter le conteneur. Voici les étapes pour cela :

1. **Construire l'image Docker** :

   Dans le répertoire où se trouve votre Dockerfile et votre fichier `requirements.txt`, exécutez la commande suivante pour construire l'image Docker :

   ```
   docker build -t my-python-app .
   ```

   Cette commande va analyser le Dockerfile et exécuter les instructions pour construire l'image Docker. L'option `-t` permet de donner un nom à l'image (`my-python-app` dans cet exemple).

2. **Exécuter le conteneur** :

   Une fois l'image construite, vous pouvez exécuter votre application Python dans un conteneur Docker avec la commande suivante :

   ```
   docker run -p 5000:5000 my-python-app
   ```

   Ici, nous utilisons l'option `-p 5000:5000` pour mapper le port 5000 de l'hôte au port 5000 du conteneur, permettant ainsi l'accès à l'application via `http://localhost:5000`.

8.5 Optimiser l'image Docker pour Python

Pour garantir que l'image Docker est aussi légère et rapide que possible, voici quelques bonnes pratiques à suivre :

- **Utiliser une image de base légère** : Au lieu d'utiliser une image `python:3.9` complète, vous pouvez utiliser une version "slim" (par exemple, `python:3.9-slim`) pour réduire la taille de l'image.
- **Minimiser les couches Docker** : Chaque instruction dans un Dockerfile crée une nouvelle couche dans l'image. Essayez de regrouper plusieurs instructions, si possible, pour minimiser le nombre de couches.

Par exemple, vous pouvez combiner la copie du fichier `requirements.txt` et l'installation des dépendances dans une seule étape pour optimiser l'image.

dockerfile

```
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
EXPOSE 5000
CMD ["python", "app.py"]
```

8.6 Gestion des environnements dans Docker

Une bonne pratique dans les environnements de production est de séparer les variables de configuration, comme les clés d'API ou les informations de base de données, en utilisant des fichiers d'environnement ou des variables d'environnement.

Exemple de Dockerfile avec variables d'environnement :

```
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .

# Définir une variable d'environnement
ENV FLASK_APP=app.py

EXPOSE 5000
CMD ["python", "app.py"]
```

Vous pouvez également passer des variables d'environnement lors de l'exécution du conteneur :

```
docker run -p 5000:5000 -e FLASK_ENV=production my-python-app
```

Module 9 : La connaissance de l'architecture applicative de microservices

Dans ce module, nous allons explorer l'architecture applicative des microservices, une approche moderne pour concevoir, développer et déployer des applications. Les microservices ont gagné en popularité en raison de leur capacité à diviser des applications monolithiques complexes en services indépendants et autonomes, permettant ainsi une plus grande flexibilité, évolutivité et résilience dans le développement des applications.

9.1 Introduction aux Microservices

Les microservices sont une approche architecturale où une application est divisée en plusieurs petits services indépendants, chacun étant responsable d'une fonction ou d'une tâche spécifique. Ces services communiquent généralement entre eux via des API (Application Programming Interfaces) et peuvent être déployés, mis à jour, et évolués indépendamment.

9.1.1 Principaux avantages des microservices :

- **Scalabilité** : Chaque microservice peut être mis à l'échelle indépendamment en fonction de sa charge.
- **Développement et déploiement indépendants** : Les équipes peuvent travailler sur des microservices spécifiques sans interférer avec les autres parties de l'application.
- **Resilience** : Si un microservice échoue, il n'affecte pas nécessairement l'ensemble du système, ce qui améliore la tolérance aux pannes.
- **Flexibilité technologique** : Chaque microservice peut être développé en utilisant la technologie la plus appropriée pour ses besoins spécifiques.
- **Maintenance simplifiée** : Les microservices sont plus faciles à maintenir et à mettre à jour car ils sont plus petits et mieux définis.

9.2 Composants d'une architecture de microservices

Une architecture de microservices est généralement composée de plusieurs éléments clés qui permettent de gérer l'interaction, le déploiement et la communication entre les différents services.

9.2.1 Services indépendants

Chaque microservice représente un domaine fonctionnel de l'application (par exemple, un service de paiement, un service de gestion des utilisateurs, etc.). Chaque service gère son propre état, sa logique métier et ses données. Ils sont généralement responsables d'une tâche spécifique, comme la gestion des commandes ou l'authentification des utilisateurs.

9.2.2 API Gateway

Une **API Gateway** agit comme un point d'entrée centralisé pour l'ensemble des microservices. Elle redirige les demandes des utilisateurs vers les services appropriés, gère la sécurité, l'authentification, et peut effectuer des transformations de requêtes et de réponses. L'API Gateway peut également gérer la limitation du débit, la mise en cache et l'agrégation de réponses.

9.2.3 Base de données par service

Dans une architecture de microservices, chaque microservice a souvent sa propre base de données ou son propre mécanisme de stockage des données. Cela garantit une indépendance complète entre les services et permet à chaque service de gérer ses données de manière autonome.

9.2.4 Communication entre services

Les microservices communiquent généralement via des API REST ou des messages asynchrones via des systèmes de gestion de messages comme **RabbitMQ**, **Apache Kafka**, ou **Amazon SQS**. La communication peut être synchrone (requêtes HTTP) ou asynchrone (file d'attente de messages).

9.2.5 Gestion des événements et des notifications

L'architecture de microservices repose souvent sur des systèmes d'événements et de notifications pour permettre aux différents services de se notifier et de réagir aux changements. Cela peut être réalisé via des mécanismes tels que les **event-driven architectures** (EDA) où les services réagissent à des événements.

9.2.6 Déploiement et gestion des conteneurs

Les microservices sont généralement déployés dans des conteneurs Docker pour garantir la portabilité et l'isolation entre les services. Les plateformes d'orchestration comme **Kubernetes** ou **Docker Swarm** sont souvent utilisées pour déployer, gérer et scaler les microservices automatiquement.

9.3 Principes de conception des microservices

L'architecture des microservices nécessite une approche de conception spécifique, qui tient compte de l'indépendance des services, de leur résilience et de leur communication. Voici quelques principes fondamentaux dans la conception de microservices :

9.3.1 Décomposition en services indépendants

L'un des aspects fondamentaux des microservices est leur décomposition en petites unités indépendantes. Chaque microservice doit être responsable d'un seul domaine fonctionnel ou métier, ce qui le rend autonome, facile à développer, à tester et à déployer.

9.3.2 Autonomie des données

Chaque microservice doit être responsable de ses propres données, et il est déconseillé d'avoir un stockage de données centralisé. Chaque service gère son propre schéma de base de données pour éviter les dépendances entre services.

9.3.3 Communications claires et bien définies

Les microservices doivent avoir des interfaces claires pour interagir les uns avec les autres. Les API doivent être bien documentées, sécurisées et évolutives. Cela permet aux équipes de travailler sur des microservices indépendants sans affecter les autres parties de l'application.

9.3.4 Gestion des échecs

Une des caractéristiques essentielles des microservices est la gestion des échecs. Comme les services sont indépendants, un échec dans un service ne doit pas affecter l'ensemble du système. Des mécanismes comme les **circuit breakers**, la **répétition des tentatives** (retry logic) et le **délai d'expiration** doivent être mis en place pour éviter les défaillances en cascade.

9.3.5 Évolutivité et mise à l'échelle

Les microservices permettent de mettre à l'échelle chaque service indépendamment. Cela permet de répondre plus efficacement aux demandes croissantes d'un service spécifique sans affecter les autres services de l'application.

9.4 Technologies associées à l'architecture de microservices

9.4.1 Containers et Orchestration

Les conteneurs, souvent gérés par des outils comme **Docker** et orchestrés avec **Kubernetes** ou **Docker Swarm**, sont utilisés pour déployer et gérer les microservices de manière fiable et scalable. Ces technologies offrent des environnements d'exécution légers et isolés pour chaque microservice, permettant un déploiement et une mise à l'échelle rapide.

9.4.2 Systèmes de gestion des API

Des systèmes comme **API Gateway** (par exemple, **Kong**, **AWS API Gateway**, **Netflix Zuul**) sont utilisés pour centraliser l'accès aux microservices, gérer la sécurité et fournir des fonctionnalités comme l'agrégation de réponses et le routage des requêtes.

9.4.3 Communication entre microservices

Les microservices communiquent principalement via des **API RESTful** ou des protocoles de messagerie asynchrone tels que **Apache Kafka** ou **RabbitMQ**. Ces systèmes permettent de gérer la communication entre services de manière efficace et flexible.

9.4.4 Suivi et gestion des logs

Les outils comme **Prometheus**, **Grafana**, **ELK Stack (Elasticsearch, Logstash, Kibana)** ou **Jaeger** sont souvent utilisés pour le suivi, la gestion des logs et le traçage des requêtes entre les différents microservices. Cela permet de diagnostiquer rapidement les problèmes et d'assurer une surveillance en temps réel.

9.4.5 Base de données distribuée

Dans une architecture de microservices, chaque service peut avoir sa propre base de données ou magasin de données, permettant une gestion décentralisée des données. Des technologies comme **MongoDB**, **Cassandra**, **PostgreSQL**, et **MySQL** peuvent être utilisées pour les bases de données de microservices.

9.5 Défis de l'architecture des microservices

Malgré les nombreux avantages, les microservices comportent également des défis qui doivent être pris en compte :

- **Complexité accrue** : Gérer un grand nombre de microservices indépendants peut être complexe, en particulier en ce qui concerne la gestion des communications entre services, les transactions distribuées et les erreurs.
- **Surveillance et gestion des logs** : Comme les microservices sont distribués, il devient plus difficile de suivre les performances et de diagnostiquer les erreurs. Il est essentiel de mettre en place un système robuste de suivi des logs et de monitoring.
- **Gestion des versions** : Les versions des microservices doivent être gérées de manière cohérente pour éviter des conflits ou des problèmes d'incompatibilité lors des mises à jour.
- **Sécurité** : Chaque microservice est une cible potentielle pour les attaques. La sécurisation des communications entre services, l'authentification des utilisateurs et l'intégrité des données sont des enjeux importants.