BENHAMMADA Ahmed Amine INGINFOA2 12107981

TP2 - Docker - K8S

💡 Commandes Docker Clés et Explications

Les commandes suivantes ont été essentielles pour la gestion des images, des volumes et des conteneurs tout au long du TP.

Commande Docker	Rôle dans le TP
docker volume create [NOM]	**Crée un volume** sur l'hôte, garantissant la persistance des données.
docker run -d --name [NOM] -v [VOLUME]:[CHEMIN] [IMAGE]	**Démarre un conteneur** en arrière-plan (-d), lui donne un nom, et attache le volume Docker (-v) à un chemin interne (ici, /var/lib/mysql).
docker exec -it [CONTENEUR] [COMMANDE]	**Exécute une commande** (ici mysql -u root -p) dans un conteneur en cours d'exécution.
docker stop [CONTENEUR] / docker rm [CONTENEUR]	**Arrête** puis **supprime** un conteneur spécifique.
docker build -t [NOM] .	**Construit une image** à partir du Dockerfile dans le répertoire actuel (.) et lui attribue un tag/nom (-t).
docker images	**Liste** toutes les images Docker locales téléchargées ou construites.
docker search [NOM]	**Recherche** des images sur Docker Hub (utile pour trouver des images comme redis).
docker ps -a	**Liste** tous les conteneurs (actifs et arrêtés : -a).
docker system prune	**Nettoie** le système Docker en supprimant les conteneurs arrêtés, les réseaux inutilisés et le cache de construction.
docker volume ls	**Liste** tous les volumes Docker locaux existants.
docker volume inspect [NOM]	**Affiche les métadonnées** et le chemin physique (Mountpoint) d'un volume.

1. Phase 1 : Persistance Manuelle des Données

Cette phase a démontré la persistance des données en utilisant un volume externe au conteneur.

A. Démarrage du Conteneur Initial et Création du Volume

```
PS C:\Users\schwa> docker volume create mysql_data
mysql_data
PS C:\Users\schwa> docker run -d --name mysql -e MYSQL_ROOT_PASSWORD=pass -v
mysql_data:/var/lib/mysql mysql:8
# ... Logs de téléchargement/démarrage ...
603d783ddf68b0066395c954ce2afc1e1256d42f5d34bae7584bc87570f3a7ed
```

B. Création de la Base de Données et Insertion Manuelle des Données
Connexion au conteneur mysql (Mdp : pass) et exécution des commandes SQL :

```
PS C:\Users\schwa> docker exec -it mysql mysql -u root -p
Enter password:
[...]
mysql> CREATE DATABASE persistance_test;
mysql> USE persistance_test;
mysql> CREATE TABLE produits (...);
mysql> INSERT INTO produits (nom, prix) VALUES ('Clavier', 59.99);
mysql> INSERT INTO produits (nom, prix) VALUES ('Souris', 25.50);
mysql> EXIT;
Bye
```

C. Vérification de la Persistance
Arrêt et suppression du conteneur initial, puis redémarrage d'un nouveau conteneur (mysql_check) sur le même volume mysql_data.

```
PS C:\Users\schwa> docker stop mysql
mysql
PS C:\Users\schwa> docker rm mysql
mysql
PS C:\Users\schwa> docker run -d --name mysql_check -e MYSQL_ROOT_PASSWORD=pass -v
mysql_data:/var/lib/mysql mysql:8
b46c80fc8f1845a209368e8d6b8fdcc088a571296001f10939bd1ba533673fa9
```

La connexion au conteneur mysql_check a validé que les données étaient toujours présentes :

```
PS C:\Users\schwa> docker exec -it mysql_check mysql -u root -p
Enter password:
[...]
+----+---------+-------+
```

```
| id | nom     | prix  |
+----+---------+-------+
|  1 | Clavier | 59.99 |
|  2 | Souris  | 25.50 |
+----+---------+-------+
2 rows in set (0.00 sec)
mysql> EXIT;
Bye
```

---

2. Phase 2 : Industrialisation via Dockerfile

Cette phase a utilisé un **Dockerfile** pour automatiser l'initialisation de la base de données.

A. Définitions des Fichiers Source

1. Dockerfile (mon-mysql)

Dockerfile

```
# Dockerfile MySQL Simple et Complet
FROM mysql:8
ENV MYSQL_DATABASE=persistance_test
COPY init.sql /docker-entrypoint-initdb.d/
EXPOSE 3306
VOLUME /var/lib/mysql
```

2. Script d'Initialisation (init.sql)

Le script crée la BDD, la table produits et insère cinq entrées.

```
-- init.sql
-- Crée la base de données
CREATE DATABASE IF NOT EXISTS persistance_test;
USE persistance_test;
-- ... Création de la table ...
-- Insère des données initiales
INSERT INTO produits (nom, prix) VALUES
    ('Clavier', 59.99),
    ('Souris', 25.50),
    ('Écran', 199.99),
    ('Casque', 89.99),
    ('Webcam', 49.99);
```

B. Construction et Lancement du Conteneur Automatisé

```
schwa@KUSH MINGW64 ~/OneDrive/Bureau/ING/ING2/CODE TO
DEPLOYEMENT/TP/code-to-deployment-course/TP2 (main)
$ docker build -t mon-mysql .
```

```
# ... Logs de construction réussie ...
schwa@KUSH MINGW64 ~/OneDrive/Bureau/ING/ING2/CODE TO
DEPLOYEMENT/TP/code-to-deployment-course/TP2 (main)
$ docker run -d --name mysql-image -e MYSQL_ROOT_PASSWORD=pass -v mysql_data:/var/lib/mysql
-p 3307:3306 mon-mysql
f93f5873edcb9a27a2f670c1c0f7bb11b83f53ee1d81a22e5c5b06c925b17972
```

C. Vérification de l'Initialisation Automatique

La requête finale a confirmé que les cinq lignes ont été correctement initialisées par l'image personnalisée :

```
schwa@KUSH MINGW64 ~/OneDrive/Bureau/ING/ING2/CODE TO
DEPLOYEMENT/TP/code-to-deployment-course/TP2 (main)
$ sleep 20 && docker exec mysql-image mysql -u root -ppass persistance_test -e "SELECT *
FROM produits;"
mysql: [Warning] Using a password on the command line interface can be insecure.
id      nom     prix
1       Clavier 59.99
2       Souris  25.50
3       Écran   199.99
4       Casque  89.99
5       Webcam  49.99
```

3. Étude de Cas : Isolation et Réseautage Personnalisé avec Docker

🎯 Objectif de la Section

Cette étude de cas visait à comprendre l'**isolation réseau** par défaut des conteneurs Docker et à établir une communication entre deux conteneurs (site et client) via un **réseau défini par l'utilisateur**.

A. Test d'Isolation du Réseau

Le premier test a utilisé l'option --network none pour empêcher délibérément le conteneur busybox d'accéder à l'extérieur.

Commande	Rôle	Résultat et Conclusion
docker run -it --rm --network none busybox sh	Lance un conteneur sans aucune interface réseau.	ping: bad address 'google.com'. Confirme l'**isolation totale** du conteneur du monde extérieur.
ping google.com (depuis l'hôte)	Vérification de la connectivité de la machine hôte.	**Succès (0% loss)**. Confirme que le problème d'accès n'est pas lié à l'hôte.

B. Création du Réseau Personnalisé

Pour permettre la communication interne, un réseau de type bridge a été créé, nommé par erreur initialement (monreseaau).

```
PS C:\Users\schwa> docker network create monreseaau
0b8bdafa4fe9944c7190ebebb61bd7edafee0440b443df883eaec395f591ac4f

PS C:\Users\schwa> docker network ls
NETWORK ID      NAME        DRIVER    SCOPE
[...]
0b8bdafa4fe9    monreseaau  bridge    local
[...]
```

C. Déploiement des Conteneurs sur le Réseau

Deux conteneurs, un serveur **site (Nginx)** et un **client (Alpine)**, ont été déployés sur le réseau monreseaau.

1. **Déploiement du Serveur Nginx (site) :**

```
PS C:\Users\schwa> docker run -d --name site --network monreseaau nginx
5eb968e80d0c22bf460d4905077f14f7480fb1e1e9d0a8f18f4c0dbef7c8515c
```

- **Déploiement du Client Alpine (client) :**

```
PS C:\Users\schwa> docker run -it --name client --network monreseaau alpine sh
/ # ping google.com
PING google.com (142.250.178.142): 56 data bytes
64 bytes from 142.250.178.142: seq=0 ttl=63 time=12.316 ms
[...]
--- google.com ping statistics ---
15 packets transmitted, 15 packets received, 0% packet loss
round-trip min/avg/max = 5.143/38.120/123.963 ms
/ #
```

Conclusion de l'Étude de Cas : Le conteneur client, une fois attaché au réseau bridge créé par l'utilisateur (monreseaau), a retrouvé un accès réseau complet (y compris la résolution DNS pour google.com), prouvant que les réseaux personnalisés permettent l'accès externe **et** l'isolement interne par défaut (si d'autres conteneurs ne sont pas attachés).

4. 🌐 Section : Exposition et Modification du Contenu Web via Bind Mount

🎯 Objectif de la Section

Démontrer l'exposition d'un service web Nginx à la machine hôte et modifier dynamiquement le contenu du site en utilisant la fonctionnalité de **Bind Mount** de Docker.

A. Exposition du Port à l'Hôte

Le conteneur Nginx initial a été lancé en mappant le port interne **80** (le port d'écoute de Nginx) au port **8080** de la machine hôte.

Commande	Explication
docker run -d -p 8080:80 nginx	Lance Nginx en arrière-plan (-d) et mappe le port 8080 de l'hôte au port 80 du conteneur.
docker stop 937f3412064c / docker rm 937f3412064c	Arrêt et suppression du premier conteneur pour préparer le Bind Mount.

B. Préparation du Contenu Personnalisé (Hôte)

Un nouveau dossier (mon-site) a été créé sur l'hôte (C:\Users\schwa) et un fichier index.html personnalisé y a été placé, remplaçant la page d'accueil par défaut de Nginx.

```
PS C:\Users\schwa> mkdir mon-site
PS C:\Users\schwa\mon-site>echo "<!DOCTYPE html>
<html>
<head>
    <title>Mon Site Docker</title>
    <style>
        body { font-family: sans-serif; background-color: #f0f8ff; text-align: center;
padding-top: 50px; }
        h1 { color: #336699; }
    </style>
</head>
<body>
    <h1>🎉 Succès ! Le Contenu a été Modifié par Docker Bind Mount!</h1>
    <p>Cette page n'est plus la page par défaut de Nginx.</p>
</body>
</html>" > index.html
```

C. Lancement du Conteneur avec Bind Mount

Le conteneur Nginx a été relancé en utilisant l'option -v pour lier le répertoire hôte (/c/Users/schwa/mon-site) au répertoire de service web interne de Nginx (/usr/share/nginx/html).

```
PS C:\Users\schwa\mon-site> docker run -d --name site_modifie -p 8080:80 -v
/c/Users/schwa/mon-site:/usr/share/nginx/html nginx
26897e172bfdd98a93b4481621e066860465c2e16c175d624b59c8019ffe1472
```

D. Validation du Succès

L'accès via le navigateur à http://localhost:8080 a confirmé la réussite de l'opération : le contenu affiché est le fichier index.html personnalisé, prouvant que le **Bind Mount est actif** et permet une modification dynamique du site web depuis l'hôte.

5. 💻 Conteneurisation d'Applications

A. Exemple 1 : Application Monolithe Simple (Script Shell)

Cet exemple montre comment conteneuriser une commande simple en utilisant une image de base extrêmement légère.

1. Fichiers Source

Fichier	Contenu	Rôle

Dockerfile	`FROM alpine:latest`
	
	`CMD ["echo", "Bonjour Polytech Nancy"]`
		Utilise l'image **Alpine** (très légère) et définit la commande de démarrage.

2. Processus et Résultat

L'image a été construite et nommée mon-app. L'exécution du conteneur a déclenché la commande CMD définie dans le Dockerfile.

```
schwa@KUSH MINGW64 [...] $ docker build -t mon-app .
# ... Logs de construction réussie ...
schwa@KUSH MINGW64 [...] $ docker run mon-app
Bonjour Polytech Nancy
```

Analyse : Le conteneur s'est exécuté, a affiché le message prédéfini, et s'est arrêté immédiatement (car le processus echo s'est terminé). Ceci valide la construction et l'exécution de base d'une image.

B. Exemple 2 : Application Java (Build Multistage)

Cet exemple est plus avancé et utilise une technique essentielle pour les applications compilées : le **Build Multistage** (construction multi-étapes).

1. Fichiers Source

Fichier	Contenu	Rôle
Hello.java	public class Hello { public static void main(String[] args) { System.out.println("Bonjour Polytech Nancy !"); } }	Code source de l'application Java.

Dockerfile	(Voir ci-dessous)	Définit deux étapes : une pour la compilation et une pour l'exécution.

2. Dockerfile (Build Multistage)

Ce Dockerfile utilise deux étapes (builder et stage-1) pour optimiser la taille finale de l'image.

```
FROM eclipse-temurin:21-jdk AS builder

WORKDIR /app

COPY Hello.java .

RUN javac Hello.java  # Étape 1 : Compilation (nécessite le JDK)

FROM eclipse-temurin:21-jre # Étape 2 : Runtime (ne nécessite que le JRE, plus petit)

WORKDIR /app

COPY --from=builder /app/Hello.class . # Copie uniquement le fichier compilé


CMD ["java", "Hello"] # Exécution de l'application
```

3. Processus et Résultat

L'image a été construite et nommée mon-app-java. L'exécution du conteneur a démarré la machine virtuelle Java et exécuté la classe compilée.

```
schwa@KUSH MINGW64 [...] $ docker build -t mon-app-java .
# ... Logs de construction, incluant les deux étapes (builder et runtime) ...
schwa@KUSH MINGW64 [...] $ docker run mon-app-java
Bonjour Polytech Nancy !
```

Analyse : Le Build Multistage a permis de compiler l'application Java dans la première étape (utilisant la lourde image **JDK**) et de ne transférer que le fichier compilé (Hello.class) dans la deuxième étape, plus légère (**JRE**). Le résultat est une image finale optimisée pour l'exécution.

C. Exemple 3 : Conteneurisation de l'API Flask

🎯 Objectif de la Section

Valider le cycle de vie **CRUD** de l'API Flask conteneurisée, confirmant l'accès externe après la correction de la configuration du host.

A. Fichiers Source et Correction Critique

La correction cruciale pour assurer l'accessibilité à travers Docker a été l'ajout de host='0.0.0.0' dans le fichier app.py.

1. Dockerfile

Ce fichier a été utilisé pour encapsuler l'application Python et ses dépendances :

```
# Image Python
FROM python:3.12-slim

# Rép de travail
WORKDIR /app

# installation flask
RUN pip install Flask

# on copie app.py dans l'image
COPY app.py .

# On expose le port 5000
EXPOSE 5000

# On définit la commande qui démarre l'app
CMD ["python", "app.py"]
```

2. Lancement Corrigé

La commande de lancement a utilisé le mapping de port 4900 vers 5000 :

```
# Exemple de lancement après reconstruction (avec host='0.0.0.0' dans app.py)
docker run -d --name flask_rest_app -p 4900:5000 flask-api-rest
```

B. Validation des Fonctionnalités (Réussite)

Après la reconstruction et le lancement du conteneur avec la correction host='0.0.0.0', tous les endpoints de l'API REST (GET, POST, PUT, DELETE) sont devenus accessibles et fonctionnels via l'adresse http://localhost:4900.

2. **Résultat :** Le conteneur flask_rest_app a permis une interaction complète avec l'API, démontrant que l'application est correctement conteneurisée et exposée.

C. Conclusion de la Conteneurisation

L'application Flask a été conteneurisée avec succès et toutes ses fonctionnalités REST (CRUD) sont opérationnelles via le port exposé de Docker. Le dépannage a mis en évidence l'importance vitale de configurer les applications web conteneurisées pour écouter sur **0.0.0.0** afin d'assurer l'accessibilité à travers le réseau bridge de Docker.

6. Mise en Œuvre d'une Architecture Multi-Conteneurs (Microservice) avec Docker Compose

🎯 Objectif de la Section

Déployer deux services interdépendants—une API Python (product-service) et une application PHP/Apache (website)—sur un réseau commun défini par **Docker Compose**, et valider leur communication.

A. Configuration Docker Compose (docker-compose.yml)

La configuration établit deux services sur un réseau **bridge** par défaut. Le service website dépend de product-service pour assurer l'ordre de démarrage.

```
services:
  product-service:
    build: ./Product
    volumes:
      - ./Product:/usr/src/app
    ports:
      - "5001:80"  # Expose le service REST sur le port hôte 5001

  website:
    image: php:apache
    volumes:
      - ./Website:/var/www/html
    ports:
      - "5002:80"  # Expose le site web sur le port hôte 5002
    depends_on:
      - product-service
```

B. Accessibilité et Endpoints

Les deux services sont accessibles depuis votre machine hôte aux adresses suivantes :

Service	Rôle	Adresse d'Accès Hôte	Port Interne Conteneur
Product API	Microservice Flask/REST (Fournisseur de données)	http://localhost:5001	80
Website	Application Client (PHP/Apache)	http://localhost:5002	80

C. Déploiement et Logs d'Exécution

Le déploiement a été effectué avec succès en utilisant docker compose up.

Service	Observation des Logs
Démarrage	product-service-1 démarre sur http://172.19.0.2:80/. website-1 démarre sur 172.19.0.3.
Communication	Le log product-service-1 montre : 172.19.0.3 - - [...] "GET / HTTP/1.1" 200 -. Ceci est la **requête PHP** du conteneur website-1 vers l'API (product-service), confirmant que la **communication inter-conteneurs via le nom de service** (http://product-service/) a réussi.

D. Validation des Résultats

La réussite de la requête interne prouve que :

3. Le réseau Docker Compose est fonctionnel.
4. Le conteneur **website** a pu résoudre le nom **product-service** en son adresse IP interne.
5. Le script **index.php** a pu récupérer la liste des produits depuis l'API.

7. Déploiement et Orchestration avec Minikube (Kubernetes)

🎯 Objectif de la Section

Déployer un service conteneurisé (Nginx) sur un cluster Kubernetes local en utilisant **Minikube**, puis gérer le cycle de vie du *Deployment*, l'échelle (scaling), et l'exposition du service.

A. Initialisation du Cluster

Vous avez vérifié la version de **Minikube** (v1.37.0) et démarré le cluster local en utilisant le pilote Docker.

Commande	Rôle	Résultat Clé
minikube version	Vérifie l'outil local.	minikube version: v1.37.0
minikube start	**Démarre le cluster** Minikube (créant une machine virtuelle ou un conteneur pour le plan de contrôle).	Automatically selected the docker driver. Done! kubectl is now configured...
kubectl get nodes	Vérifie que le cluster est opérationnel.	Le nœud minikube a le statut **Ready**.

B. Création, Déploiement, et Mise à l'Échelle (Scaling)

Après quelques erreurs de syntaxe (par exemple, deployementt et le flag --image), le déploiement de Nginx a été créé avec succès.

Commande	Rôle	État Initial (Pods)
kubectl create deployment hello-nginx --image=nginx	Crée un **Deployment** nommé hello-nginx à partir de l'image Nginx, qui crée initialement 1 Pod.	hello-nginx-fbcf4796f-hrmv7 (1/1 Ready, Running)
kubectl scale deployment hello-nginx --replicas=3	**Mise à l'échelle.** Demande à Kubernetes de maintenir 3 copies (Pod) du conteneur Nginx en cours d'exécution.	
kubectl get pods	Vérifie la mise à l'échelle.	**3 Pods** Nginx en état Running sont affichés (par exemple, ...-rqdvq, ...-sqtw7).

C. Exposition du Service

Pour rendre l'application accessible depuis l'extérieur du cluster K8s, le Deployment a été exposé via un service de type **NodePort**.

Commande	Rôle	Résultat
kubectl expose deployment hello-nginx --type=NodePort --port 80	Crée un **Service** qui expose le port 80 des Pods sur un port aléatoire (NodePort) de la machine hôte.	service/hello-nginx exposed
minikube service hello-nginx --url	Récupère l'URL et le port externe attribué par Minikube.	http://127.0.0.1:56387 (Le port est dynamique, ici 56387).

D. Nettoyage

Les ressources Kubernetes ont été supprimées et le cluster Minikube a été éteint.

```
kubectl delete service hello-nginx
kubectl delete deployment hello-nginx
minikube stop
```