Challenge Docker

v1.0.0 | 17/11/2024 | Auteur : Bauer Baptiste

Chapitre

Sommaire

Les challenges sont une série de petits exercices permettant de mettre en pratique l'utilisation de Docker dans des situations fictives, mais crédibles.

1. Challenge : Les volumes

1. Crée un conteneur basé sur l'image nginx.
2. Configure un volume pour que les fichiers du répertoire `/usr/share/nginx/html` dans le conteneur soient mappés à un répertoire local sur ta machine.
3. Modifie un fichier HTML dans le répertoire local et vérifie que les changements sont reflétés dans le conteneur.

2. Challenge : Mapping de ports

1. Lance un conteneur basé sur l'image `httpd` (Apache) en mappant le port `8080` de l'hôte au port 80 du conteneur.
2. Accède à la page Apache par ton navigateur via http://localhost:8080.
3. Arrête le conteneur et relance-le avec un autre port (exemple : 9090) pour vérifier que le mapping fonctionne.

3. Challenge : Création de conteneur

1. Télécharge l'image officielle de `redis`.
2. Lance un conteneur avec : Un nom personnalisé (ex : *my-redis*).
3. Une configuration pour qu'il redémarre automatiquement en cas de crash.
4. Un mapping du port `6379` de l'hôte au conteneur.
5. Vérifie que le conteneur fonctionne en utilisant `redis-cli`.

4. Challenge : Dockerisation d'une application

1. Prends une simple application `Node.js` (ou utilise l'exemple ci-dessous).
2. Écris un fichier **Dockerfile** pour cette application :
    a. Installe les dépendances.
    b. Expose le port utilisé par l'application.
    c. Construis une image Docker pour cette application.
    d. Lance un conteneur basé sur cette image et accède à l'application.

Application NodeJs simple

```
// app.js
const express = require('express');
const app = express();
app.get('/', (req, res) => res.send('Hello Docker!'));
app.listen(3000, () => console.log('App running on port 3000'));
```

5. Challenge : Utilisation de `bind-mount` et persistance des données

1. Lance un conteneur MySQL et mappe le répertoire local pour stocker les données MySQL persistantes.
2. Arrête et supprime le conteneur.
3. Relance un nouveau conteneur avec le même mapping et vérifie que les données sont toujours présentes.

6. Challenge : Multi-stage builds

1. Utilise l'application Go ci-dessous.
2. Écris un fichier **Dockerfile** pour construire l'application en deux étapes :
   a. **Étape 1** : Compile l'application dans un conteneur.
   b. **Étape 2** : Copie le binaire compilé dans une image plus légère (ex : alpine).
3. Lance un conteneur basé sur cette image pour tester l'application.

Application GO simple

```
// main.go
package main
import "fmt"
func main() {
    fmt.Println("Hello from Docker multi-stage build!")
}
```

7. Challenge : Simulation d’un site web en production

1. Lance un conteneur nginx pour servir une application front-end.
2. Lance un conteneur node pour le backend.
3. Configure le fichier `nginx.conf` pour rediriger les requêtes du front vers le backend.
4. Utilise des volumes pour permettre des mises à jour à chaud sur les fichiers de configuration.

8. TD : Comprendre le Multi stage

8.1. Objectif :

Créer une application minimaliste, la compiler avec un Dockerfile multi-stage pour réduire la taille de l'image finale.

8.2. Étape 1 : Préparer l'application

Nous allons utiliser une application simple en **Go** comme exemple. Cette méthode fonctionne également pour des applications Node.js, Python, etc.

Crée un fichier nommé **main.go** :

main.go

```
package main

import "fmt"

func main() {
    fmt.Println("Hello, Multi-Stage Docker!")
}
```

8.3. Étape 2 : Comprendre le multi-stage

Un Dockerfile multi-stage permet d'utiliser plusieurs étapes pour construire une application. Chaque étape utilise une image de base différente. Les fichiers nécessaires sont transférés entre ces étapes. Cela réduit la taille de l'image finale en excluant les outils de build ou les fichiers inutiles.

8.4. Étape 3 : Écrire le Dockerfile

.Dockerfile

```
# Étape 1 : Compilation
FROM golang:1.20 AS builder
# Définir le répertoire de travail
WORKDIR /app
# Copier les fichiers dans le conteneur
COPY main.go .
# Compiler l'application
RUN go build -o main .

# Étape 2 : Image minimale
FROM alpine:latest
# Définir le répertoire de travail
WORKDIR /root/
```

```
# Copier uniquement le binaire depuis le conteneur builder
COPY --from=builder /app/main .
# Spécifier la commande à exécuter
CMD ["./main"]
```

8.5. Étape 4 : Explication

Compilation

- `FROM golang:1.20 AS builder` : Utilise une image Go officielle avec un alias (builder).
- `WORKDIR /app` : Définit le répertoire de travail pour le conteneur.
- `COPY main.go .` : Copie le fichier source dans le conteneur.
- `RUN go build -o main .` : Compile l'application en un exécutable appelé main.

Image finale

- `FROM alpine:latest` : Utilise une image Alpine légère.
- `COPY --from=builder /app/main .` : Copie le fichier binaire depuis l'étape builder.
- `CMD ["./main"]` : Définit la commande par défaut.

8.6. Étape 5 : Construire et tester l'image

Dans le terminal, exécute la commande suivante pour construire l'image :

```
docker build -t multi-stage-demo .
```

Lance un conteneur basé sur l'image construite :

```
docker run --rm multi-stage-demo
```

Résultat attendu : Le terminal doit afficher ⇒

```
Hello, Multi-Stage Docker!
```

8.7. Étape 6 : Vérification de la taille de l'image

Compare la taille des images pour comprendre l'impact du multi-stage :

```
docker images
```

- L'image golang:1.20 utilisée pour la compilation est volumineuse.
- L'image finale basée sur alpine est beaucoup plus légère.

Résumé :

Un Dockerfile multi-stage :

- Réduit la taille de l'image.
- Sépare les étapes de compilation et d'exécution.
- Simplifie la maintenance des images.

Ce TD te permet de comprendre l'approche multi-stage tout en te donnant une base pour créer des images optimisées !