Conteneurisation avec Docker

Déploiement d'applications full-stack et orchestration

Dr. El Hadji Bassirou TOURE
Département de Mathématiques et Informatique
Faculté des Sciences et Techniques
Université Cheikh Anta Diop

2025

Résumé

Ce cours-lab vise à vous initier à Docker et à la conteneurisation d'applications modernes. Vous apprendrez les principes fondamentaux des conteneurs, comment créer des images Docker efficaces, déployer une application complète (frontend, backend et base de données), partager vos images sur Docker Hub, et orchestrer votre application avec Docker Compose. À travers une approche progressive combinant théorie et pratique, vous maîtriserez les outils et concepts essentiels pour moderniser vos méthodes de développement et de déploiement d'applications.

Ressources pour le cours-TP

Note importante : Pour faciliter la réalisation de ce TP, l'ensemble du code source de l'application de démonstration est disponible sur GitHub aux adresses suivantes :

- Code du backend : `https://github.com/elbachir67/tp-agl-backend-code.git`
- Code du frontend : `https://github.com/elbachir67/tp-agl-frontend-code.git`

Ces dépôts contiennent tous les éléments nécessaires pour suivre le TP. Une version live de l'application est également disponible ici : `https://permissible-road.surge.sh/` (`Username=user, Password=user`).

Table des matières

1 Introduction à Docker et aux conteneurs

Concept fondamental

Docker est une plateforme open-source qui automatise le déploiement d'applications à l'intérieur de conteneurs logiciels. Les conteneurs permettent d'empaqueter une application avec toutes ses dépendances dans une unité standardisée pour le développement logiciel. Contrairement aux machines virtuelles traditionnelles, les conteneurs partagent le même noyau du système d'exploitation et isolent les processus de l'application, ce qui les rend plus légers et plus efficaces.

1.1 Qu'est-ce qu'un conteneur ?

Intuition

Imaginez que vous déménagez dans une nouvelle maison. Au lieu de transporter tous vos meubles et objets en vrac dans un camion, vous les mettez dans des conteneurs d'expédition standardisés. Ces conteneurs sont faciles à transporter, à empiler et à déplacer d'un camion à un bateau, puis à un train. Ils protègent leur contenu et s'adaptent à n'importe quel moyen de transport.
De la même manière, les conteneurs Docker encapsulent une application et ses dépendances dans un "package" standardisé qui peut s'exécuter de manière cohérente sur n'importe quel environnement : du poste de développement aux serveurs de test, puis en production.

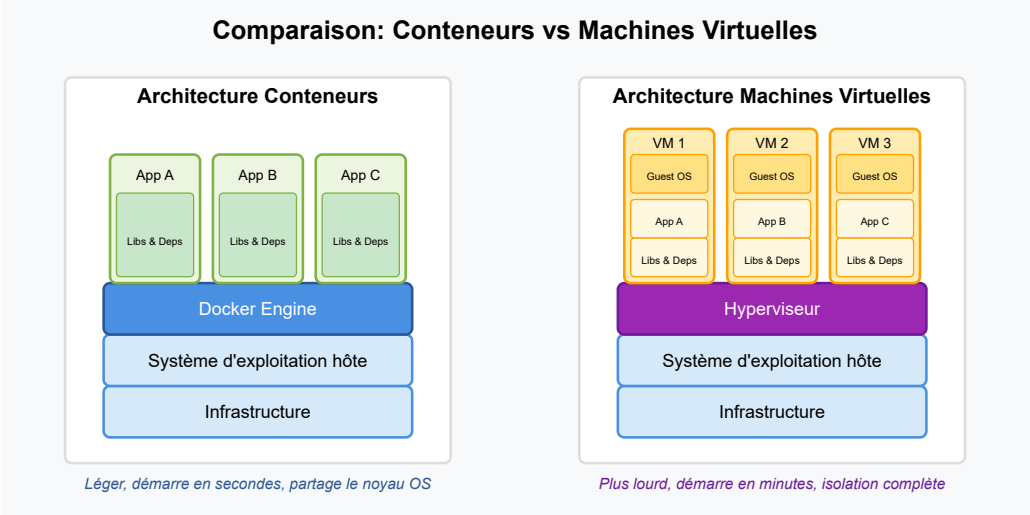

FIGURE 1 – Comparaison entre conteneurs et machines virtuelles

Point clé à retenir

Différences clés entre conteneurs et machines virtuelles (VM) :

- **Légèreté** : Les conteneurs partagent le noyau du système d'exploitation hôte et ne nécessitent pas de système d'exploitation complet pour chaque instance, ce qui les rend beaucoup plus légers (de l'ordre de quelques dizaines de MB) que les VM (de l'ordre de quelques GB).
- **Démarrage rapide** : Les conteneurs démarrent en quelques secondes, contre plusieurs minutes pour les VM.
- **Efficacité des ressources** : Les conteneurs consomment moins de ressources CPU et mémoire.
- **Portabilité** : Les conteneurs fonctionnent de manière identique dans tous les environnements qui disposent d'un moteur de conteneur.
- **Isolation** : Bien que l'isolation soit moins complète que celle des VM, elle est suffisante pour la plupart des cas d'utilisation et peut être renforcée si nécessaire.

1.2 L'architecture de Docker

Docker utilise une architecture client-serveur où :

- Le **client Docker** est l'interface principale avec laquelle les utilisateurs interagissent via des commandes.
- Le **démon Docker** (dockerd) est un processus persistant qui gère les objets Docker comme les images, les conteneurs, les réseaux et les volumes.
- Le **registre Docker** stocke les images Docker (Docker Hub est le registre public par défaut).

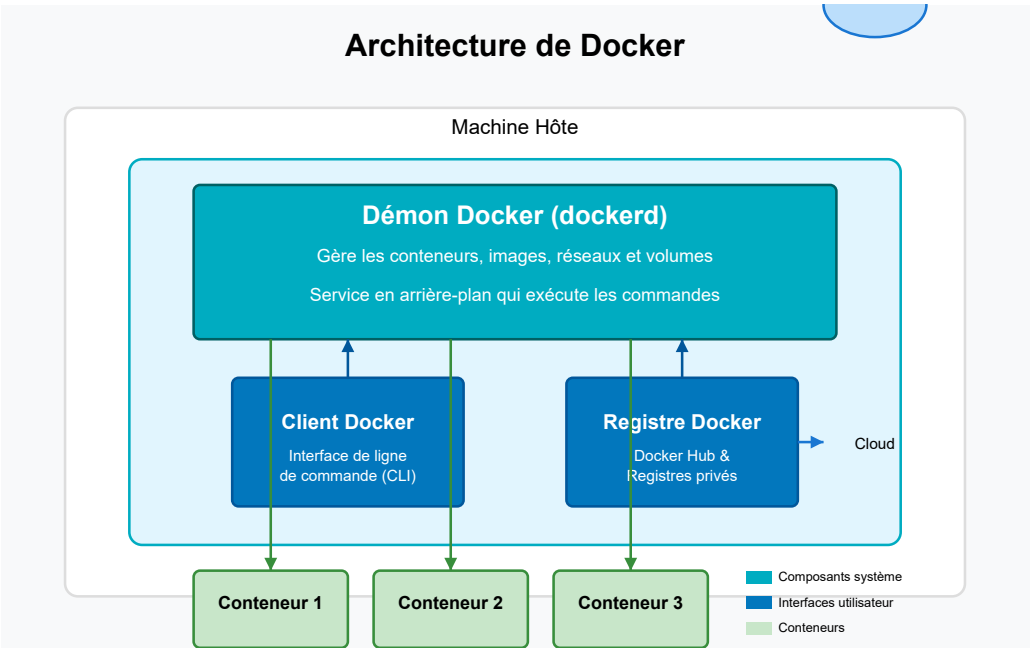

FIGURE 2 – Architecture de Docker

1.3 Concepts clés de Docker

Concept fondamental

Comprendre Docker nécessite de maîtriser trois concepts fondamentaux : les images, les conteneurs et les Dockerfiles.

— **Image Docker** : Un modèle en lecture seule contenant un ensemble de couches qui représentent une application et ses dépendances. Les images sont immuables et servent de base pour créer des conteneurs.

— **Conteneur Docker** : Une instance en cours d'exécution d'une image, avec son propre système de fichiers, son espace de nommage et ses ressources isolées.

— **Dockerfile** : Un script contenant des instructions pour construire automatiquement une image Docker, définissant l'environnement d'exécution de l'application.

— **Docker Compose** : Un outil permettant de définir et gérer des applications multi-conteneurs avec un fichier YAML.

— **Volume Docker** : Un mécanisme de persistance des données générées par les conteneurs, indépendant de leur cycle de vie.

Exemple

Analogie avec la programmation orientée objet :
Si nous faisons une analogie avec la POO :

— Une **image Docker** est comparable à une **classe** : un modèle défini.

— Un **conteneur** est comme une **instance d'objet** : une entité exécutable basée sur le modèle.

— Un **Dockerfile** est similaire au **code source** qui définit la classe.

— Un **volume Docker** serait comme une **variable statique** partagée entre instances.

2 Mise en place de l'environnement de travail

2.1 Installation de Docker

Pour suivre ce cours-lab, vous aurez besoin d'installer Docker sur votre système. Les instructions varient selon votre système d'exploitation.

Installation de Docker sur diffÃ¨rents systÃ¨mes d'exploitation

```
# Pour Ubuntu/Debian
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io

# Pour Windows
# Télécharger et installer Docker Desktop depuis https://www.docker.com/products/docker-desktop

# Pour macOS
# Télécharger et installer Docker Desktop depuis https://www.docker.com/products/docker-desktop

# Vérifier l'installation
docker --version
docker run hello-world
```

Point clé à retenir

Si vous utilisez Windows ou macOS, Docker Desktop inclut Docker Engine, Docker CLI, Docker Compose, et d'autres outils essentiels. Pour Linux, ces composants doivent être installés séparément.
Assurez-vous également d'avoir les droits d'administrateur ou d'être dans le groupe "docker" pour exécuter les commandes sans sudo.

2.2 Clonage des dépôts de code pour le TP

Pour réaliser les exercices pratiques, nous allons utiliser une application full-stack composée de :

— Un frontend en React

— Un backend en Spring Boot

— Une base de données MariaDB

Tâche à réaliser

Clonage des dépôts de code source :
Ouvrez un terminal et exécutez les commandes suivantes :

```
# Création d'un dossier pour le projet
mkdir -p docker-fullstack-app
cd docker-fullstack-app

# Clonage du backend
git clone https://github.com/elbachir67/tp-agl-backend-code.git backend

# Clonage du frontend
git clone https://github.com/elbachir67/tp-agl-frontend-code.git frontend
```

3 Les commandes Docker essentielles

Avant de conteneuriser notre application complète, familiarisons-nous avec les commandes Docker fondamentales.

3.1 Gestion des images Docker

Commandes de base pour la gestion des images

```
# Télécharger une image depuis Docker Hub
docker pull nginx:latest

# Lister les images disponibles localement
docker images

# Construire une image à partir d'un Dockerfile
docker build -t mon-app:1.0 .

# Supprimer une image
docker rmi nginx:latest

# Trouver des images sur Docker Hub
docker search ubuntu
```

3.2 Gestion des conteneurs

Commandes de base pour la gestion des conteneurs

```
# Créer et démarrer un conteneur
docker run --name mon-conteneur -p 8080:80 -d nginx:latest

# Lister les conteneurs en cours d'exécution
docker ps

# Lister tous les conteneurs (y compris ceux arrêtés)
docker ps -a

# Arrêter un conteneur
docker stop mon-conteneur

# Démarrer un conteneur existant
docker start mon-conteneur

# Supprimer un conteneur
docker rm mon-conteneur

# Exécuter une commande dans un conteneur en cours d'exécution
docker exec -it mon-conteneur bash

# Voir les logs d'un conteneur
docker logs mon-conteneur
```

Intuition

Dans Docker, la distinction entre démarrage et création est importante :

— `docker run` à la fois crée ET démarre un nouveau conteneur

— `docker start` redémarre un conteneur existant qui a été arrêté

Pensez à `docker run` comme à l'action de démarrer une voiture pour la première fois, tandis que `docker start` est comme redémarrer une voiture déjà utilisée.

3.3 Exercice pratique : Manipulation de base de Docker

Tâche à réaliser

Exercice 1 : Manipulations de base avec Docker
Suivez ces étapes pour vous familiariser avec les commandes Docker de base :

1. Téléchargez l'image officielle de Nginx :

   ```
   docker pull nginx:latest
   ```

2. Lancez un conteneur Nginx qui expose le port 80 sur le port 8080 de votre machine :

   ```
   docker run --name test-nginx -p 8080:80 -d nginx
   ```

3. Vérifiez que le conteneur est en cours d'exécution :

   ```
   docker ps
   ```

4. Ouvrez votre navigateur et accédez à http://localhost:8080 pour voir la page par défaut de Nginx.
5. Examinez les logs du conteneur :

   ```
   docker logs test-nginx
   ```

6. Modifiez la page d'accueil de Nginx en exécutant une commande dans le conteneur :

   ```
   docker exec -it test-nginx bash
   echo "<h1>Hello Docker World!</h1>" > /usr/share/nginx/html/index.html
   exit
   ```

7. Rafraîchissez votre navigateur pour voir les changements.
8. Arrêtez et supprimez le conteneur :

   ```
   docker stop test-nginx
   docker rm test-nginx
   ```

4 Création d'images Docker avec Dockerfile

Concept fondamental

Un Dockerfile est un script contenant une série d'instructions qui décrivent comment construire une image Docker. Chaque instruction crée une nouvelle couche dans l'image, permettant une construction efficace et incrémentale.

4.1 Structure d'un Dockerfile

Structure de base d'un Dockerfile

```
# Image de base
FROM node:14-alpine

# Métadonnées
LABEL maintainer="votre.email@exemple.com"
LABEL version="1.0"

# Variables d'environnement
ENV NODE_ENV=production

# Répertoire de travail
WORKDIR /app

# Copie des fichiers
COPY package*.json ./
COPY src/ ./src/

# Exécution de commandes
RUN npm install

# Exposition de ports
EXPOSE 3000

# Commande de démarrage
CMD ["npm", "start"]
```

Point clé à retenir

Instructions Dockerfile les plus courantes :

- **FROM** : Spécifie l'image de base (obligatoire et généralement première instruction)
- **WORKDIR** : Définit le répertoire de travail pour les instructions suivantes
- **COPY/ADD** : Copie des fichiers de l'hôte vers l'image
- **RUN** : Exécute des commandes pendant la construction de l'image
- **ENV** : Définit des variables d'environnement
- **EXPOSE** : Documente les ports sur lesquels le conteneur écoute
- **CMD/ENTRYPOINT** : Spécifie la commande à exécuter lorsque le conteneur démarre
- **VOLUME** : Déclare des points de montage pour les données persistantes

Point d'attention

Différences entre CMD et ENTRYPOINT :

- **CMD** : Définit la commande par défaut qui peut être remplacée lors de l'exécution du conteneur avec `docker run`
- **ENTRYPOINT** : Définit la commande principale qui ne peut pas être facilement remplacée

  Un cas d'usage courant est de combiner les deux : ENTRYPOINT définit l'exécutable et CMD fournit les arguments par défaut.

Exemple :

```
ENTRYPOINT ["java"]
CMD ["-jar", "app.jar"]
```

4.2 Bonnes pratiques pour la création de Dockerfiles

Point clé à retenir

Meilleures pratiques pour des images Docker efficaces :

- **Utiliser des images de base légères** : Privilégier les variantes Alpine ou Slim
- **Combiner les instructions RUN** : Réduire le nombre de couches en utilisant `&&` pour enchaîner les commandes
- **Supprimer les fichiers inutiles** dans la même instruction RUN
- **Utiliser .dockerignore** pour exclure les fichiers non nécessaires
- **Optimiser l'ordre des couches** : Placer les instructions qui changent peu au début
- **Utiliser des arguments de construction** (ARG) pour les valeurs variables
- **Spécifier des versions précises** des images de base plutôt que `latest`
- **Ne pas exécuter les conteneurs en tant que root** quand possible

4.3 Exercice pratique : Création d'un Dockerfile simple

Tâche à réaliser

Exercice 2 : Création d'un Dockerfile pour une application Node.js simple
Dans cet exercice, vous allez créer une image Docker pour une application Node.js simple.

1. Créez un nouveau dossier et les fichiers suivants :
   Dossier : simple-node-app
   Fichier : package.json

```
{
  "name": "simple-node-app",
  "version": "1.0.0",
  "description": "Une application Node.js simple pour Docker",
  "main": "server.js",
  "scripts": {
    "start": "node server.js"
  },
  "dependencies": {
    "express": "^4.17.1"
  }
}
```

   Fichier : server.js

```
const express = require('express');
const app = express();
const PORT = process.env.PORT || 3000;

app.get('/', (req, res) => {
  res.send('<h1>Bonjour depuis Docker!</h1>');
});

app.listen(PORT, () => {
  console.log(`Serveur démarré sur le port ${PORT}`);
});
```

   Fichier : .dockerignore

```
node_modules
npm-debug.log
```

2. Créez un Dockerfile avec le contenu suivant :

```
FROM node:14-alpine

WORKDIR /app

COPY package*.json ./

RUN npm install

COPY . .

EXPOSE 3000

CMD ["npm", "start"]
```

3. Construisez l'image Docker :

```
docker build -t simple-node-app .
```

4. Exécutez l'application dans un conteneur :

```
docker run --name node-app -p 3000:3000 -d simple-node-app
```

5. Testez l'application en ouvrant http://localhost:3000 dans votre navigateur.
6. Arrêtez et supprimez le conteneur lorsque vous avez terminé :

```
docker stop node-app
docker rm node-app
```

5 Dockerisation d'une application full-stack

Concept fondamental

Conteneuriser une application full-stack implique de créer des images Docker distinctes pour chaque composant (frontend, backend, base de données) et de les configurer pour qu'ils puissent communiquer entre eux. Cette approche modulaire est au cœur de l'architecture microservices et facilite le déploiement, la mise à l'échelle et la maintenance.

Dans cette section, nous allons dockeriser une application complète composée de trois éléments :

— Une base de données MariaDB

— Un backend Spring Boot

— Un frontend React

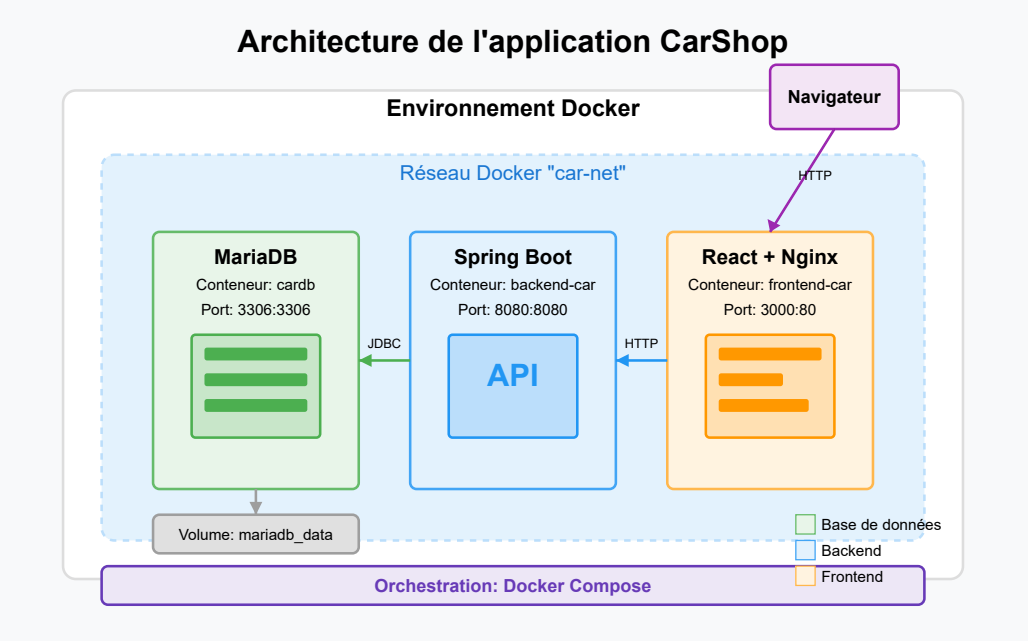

FIGURE 3 – Application carshop à déployer

5.1 Mise en place de la base de données MariaDB

Nous commencerons par déployer une instance MariaDB pour notre application.

CrÃĺation et dÃĺmarrage d'un conteneur MariaDB

```
# Téléchargement de l'image officielle MariaDB
docker pull mariadb:latest

# Vérification que l'image a bien été téléchargée
docker images

# Création et démarrage du conteneur MariaDB
docker run --name cardb -p 3306:3306 \
  -e MARIADB_ROOT_PASSWORD=root \
  -e MARIADB_DATABASE=cardb \
  -d mariadb

# Vérification que le conteneur est en cours d'exécution
docker ps
```

Point clé à retenir

Paramètres importants :

- `-name cardb` : Assigne un nom au conteneur pour y faire référence facilement
- `-p 3306:3306` : Mappe le port 3306 du conteneur sur le port 3306 de l'hôte
- `-e MARIADB_ROOT_PASSWORD=root` : Définit le mot de passe root
- `-e MARIADB_DATABASE=cardb` : Crée automatiquement une base de données
- `-d` : Exécute le conteneur en arrière-plan (mode détaché)

5.2 Configuration d'un réseau Docker

Pour permettre la communication entre les conteneurs, nous allons créer un réseau Docker dédié.

CrÃľation d'un rÃľseau Docker et connexion du conteneur MariaDB

```
# Création d'un réseau Docker nommé car-net
docker network create car-net

# Connexion du conteneur MariaDB au réseau
docker network connect car-net cardb

# Vérification de la configuration du réseau
docker network inspect car-net
```

Intuition

Les réseaux Docker sont comme des réseaux privés virtuels. Les conteneurs connectés au même réseau peuvent communiquer entre eux en utilisant simplement leur nom comme nom d'hôte (hostname), sans avoir besoin de connaître leur adresse IP spécifique. Par exemple, notre backend pourra se connecter à la base de données en utilisant simplement l'URL `jdbc:mysql://cardb:3306/cardb` au lieu d'une adresse IP.

5.3 Création de l'image pour le backend Spring Boot

Maintenant, nous allons créer un Dockerfile pour notre application backend Spring Boot.

Dockerfile du backend (Ãă crÃľer dans le dossier backend)

```
FROM openjdk:21

WORKDIR /app

ADD jar_file target/my-car-app.jar app.jar

EXPOSE 8080

ENTRYPOINT ["java", "-jar", "/app/app.jar"]
```

Tâche à réaliser

```
Construction de l'image du backend :
Naviguez dans le dossier du backend et exécutez la commande suivante :


cd backend
docker build -t carbackend .


Vérifiez que l'image a été correctement créée :


docker images
```

Point clé à retenir

Dans ce Dockerfile pour Spring Boot :

- Nous utilisons OpenJDK 21 comme image de base
- Nous copions le fichier JAR compilé de notre application (supposé disponible dans le dossier `target`)
- Nous exposons le port 8080 sur lequel notre application Spring Boot écoute
- Nous définissons la commande pour démarrer l'application

5.4 Exécution du backend

Maintenant que l'image du backend est prête, nous pouvons exécuter un conteneur à partir de cette image.

DÃ¯marrage du conteneur backend

```
docker run -p 8080:8080 --name backend-car --net car-net \
  -e MARIADB_HOST=cardb \
  -e MARIADB_USER=root \
  -e MARIADB_PASSWORD=root \
  -e MYSQL_PORT=3306 \
  carbackend
```

Point d'attention

Notez que nous passons plusieurs variables d'environnement pour configurer la connexion à la base de données :

- `MARIADB_HOST=cardb` : Utilise le nom du conteneur MariaDB comme hôte
- `MARIADB_USER` et `MARIADB_PASSWORD` : Informations d'authentification
- `MYSQL_PORT` : Port sur lequel MariaDB écoute

Cela suppose que l'application Spring Boot est configurée pour utiliser ces variables d'environnement. Dans un contexte de production, ces informations sensibles devraient être gérées de manière plus sécurisée, par exemple avec Docker Secrets.

5.5 Création de l'image pour le frontend React

Ensuite, nous allons créer un Dockerfile pour notre application frontend React.

Dockerfile du frontend (Ãă crÃ¯er dans le dossier frontend)

```
FROM node:alpine AS builder
WORKDIR /app
COPY ./package.json ./
RUN npm install
COPY . .
RUN npm run build

FROM nginx
WORKDIR /app
COPY --from=builder /app/build /usr/share/nginx/html
COPY nginx.conf /etc/nginx/conf.d/default.conf
```

nginx.conf (Ãă crÃľer dans le dossier frontend)

```
server {
    listen       80;
    server_name  localhost;

    location / {
        root   /usr/share/nginx/html;
        index  index.html index.htm;
        try_files $uri $uri/ /index.html;
    }

    # Redirection des API calls vers le backend
    location /api {
        proxy_pass http://backend-car:8080;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }
}
```

Tâche à réaliser

Construction de l'image du frontend :
Naviguez dans le dossier du frontend et exécutez la commande suivante :

```
cd ../frontend
docker build -t carfrontend .
```

Vérifiez que l'image a été correctement créée :

```
docker images
```

Point clé à retenir

Ce Dockerfile utilise une approche multi-stage :

- Dans la première étape, nous utilisons Node.js pour construire notre application React
- Dans la seconde étape, nous copions uniquement les fichiers de build dans une image Nginx légère
- Cette technique permet de réduire considérablement la taille de l'image finale en excluant tous les outils de développement

La configuration Nginx ajoute un proxy pour rediriger les appels API vers le backend, ce qui permet de contourner les problèmes de CORS.

5.6 Exécution du frontend

Maintenant que l'image du frontend est prête, nous pouvons exécuter un conteneur à partir de cette image.

DÃľmarrage du conteneur frontend

```
docker run -d --name frontend-car -p 3000:80 --net car-net carfrontend
```

5.7 Test de l'application complète

Une fois tous les conteneurs démarrés, nous pouvons tester notre application full-stack.

Tâche à réaliser

Test de l'application :

1. Vérifiez que tous les conteneurs sont en cours d'exécution :

   ```
   docker ps
   ```

2. Ouvrez votre navigateur et accédez à http://localhost:3000
3. Vous devriez voir l'interface utilisateur de notre application de gestion de voitures
4. Essayez de vous connecter avec les identifiants user/user
5. Testez les fonctionnalités CRUD (Création, Lecture, Mise à jour, Suppression) des voitures

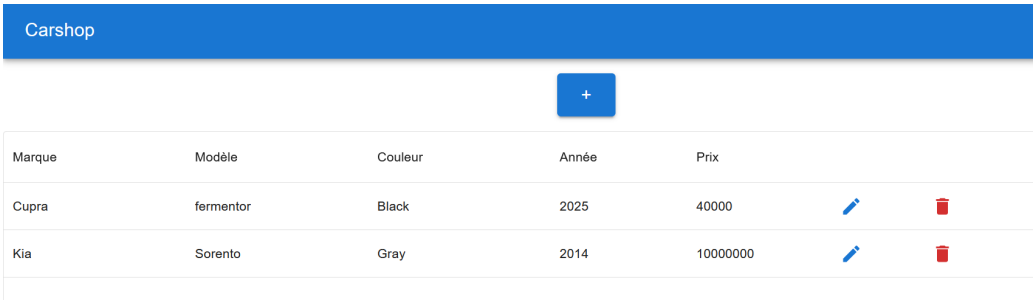

FIGURE 4 – Application full stack

Point d'attention

Si vous rencontrez des problèmes de connexion entre les conteneurs, vérifiez les points suivants :

- Tous les conteneurs sont-ils connectés au même réseau Docker (car-net) ?
- Les variables d'environnement sont-elles correctement définies ?
- Les ports sont-ils correctement exposés et mappés ?
- Vérifiez les logs de chaque conteneur pour identifier d'éventuelles erreurs :

```
docker logs cardb
docker logs backend-car
docker logs frontend-car
```

6 Persistance des données avec les volumes Docker

Concept fondamental

Les volumes Docker sont le mécanisme privilégié pour persister les données générées par les conteneurs et les utiliser entre les redémarrages. Contrairement aux montages de liaison (bind mounts), les volumes sont entièrement gérés par Docker et fonctionnent sur tous les systèmes d'exploitation.

6.1 Types de persistance dans Docker

Docker propose plusieurs mécanismes pour la persistance des données :

- **Volumes** : Gérés par Docker dans l'espace réservé du système de fichiers hôte
- **Bind mounts** : Montage direct d'un répertoire de l'hôte dans le conteneur
- **tmpfs mounts** : Stockage en mémoire (RAM) uniquement, pour les données temporaires

Intuition

Imaginez un conteneur comme une location de vacances temporaire :

- Sans volume, c'est comme partir sans rien emporter : toutes vos affaires disparaissent à la fin du séjour
- Un volume est comme une consigne où vous stockez vos affaires importantes : même si vous changez de logement (conteneur), vos affaires restent accessibles au même endroit
- Un bind mount est comme avoir un accès à votre propre maison depuis votre location : vous accédez aux mêmes objets depuis deux endroits différents

6.2 Utilisation de volumes pour la base de données

Recréons notre conteneur de base de données avec un volume pour persister les données.

CrÃ©ation d'un volume et dÃ©marrage de MariaDB avec persistance

```
# Création d'un volume Docker
docker volume create mariadb_data

# Démarrage de MariaDB avec le volume
docker run --name cardb -p 3306:3306 \
  -e MARIADB_ROOT_PASSWORD=root \
  -e MARIADB_DATABASE=cardb \
  -v mariadb_data:/var/lib/mysql \
  -d mariadb

# Connexion au réseau
docker network connect car-net cardb
```

Point clé à retenir

Avantages de l'utilisation des volumes Docker :

- **Persistance** : Les données survivent à la suppression des conteneurs
- **Portabilité** : Fonctionne de manière cohérente sur tous les OS
- **Sécurité** : Meilleure isolation que les bind mounts
- **Performance** : Optimisés pour le stockage des conteneurs
- **Facilité de sauvegarde** : Les volumes peuvent être sauvegardés et restaurés facilement

Tâche à réaliser

Exercice 3 : Test de la persistance des données
Testez la persistance des données avec les étapes suivantes :

1. Assurez-vous que votre application complète fonctionne avec les trois conteneurs
2. Ajoutez quelques voitures dans l'application via l'interface web
3. Arrêtez et supprimez le conteneur de base de données :

```
docker stop cardb
docker rm cardb
```

4. Recréez le conteneur en utilisant le même volume :

```
docker run --name cardb -p 3306:3306 \
  -e MARIADB_ROOT_PASSWORD=root \
  -e MARIADB_DATABASE=cardb \
  -v mariadb_data:/var/lib/mysql \
  -d mariadb
```

5. Reconnectez-le au réseau :

```
docker network connect car-net cardb
```

6. Redémarrez le backend (qui se connectera à la base de données) :

```
docker restart backend-car
```

7. Vérifiez dans l'interface web que les voitures ajoutées précédemment sont toujours présentes

7 Partage d'images avec Docker Hub

Concept fondamental

Docker Hub est un service d'hébergement d'images Docker, similaire à GitHub pour le code source. Il permet de stocker, partager et distribuer vos images Docker, facilitant le déploiement sur différents environnements et la collaboration avec d'autres développeurs.

7.1 Création d'un compte Docker Hub

Pour partager vos images, vous devez d'abord créer un compte sur Docker Hub.

Tâche à réaliser

Création d'un compte Docker Hub :

1. Accédez à https://hub.docker.com/
2. Cliquez sur "Sign Up" et suivez les instructions pour créer un compte
3. Une fois votre compte créé, connectez-vous depuis votre terminal :

```
docker login
```

4. Entrez vos identifiants Docker Hub lorsque vous y êtes invité

7.2 Préparation et push des images

Avant de pouvoir pousser (push) une image vers Docker Hub, vous devez la tagger avec votre nom d'utilisateur Docker Hub.

PrÃĺparation et push des images vers Docker Hub

```
# Tagger l'image de la base de données
# (Nous allons d'abord créer une image à partir du conteneur modifié)
docker commit cardb votre-username/db_car:latest

# Tagger l'image du backend
docker tag carbackend votre-username/backend_car:latest

# Tagger l'image du frontend
docker tag carfrontend votre-username/frontend_car:latest

# Pousser les images vers Docker Hub
docker push votre-username/db_car:latest
docker push votre-username/backend_car:latest
docker push votre-username/frontend_car:latest
```

Point clé à retenir

Lors de la création de tags pour vos images, suivez ces bonnes pratiques :

- Utilisez des noms significatifs qui décrivent clairement le contenu
- Incluez des numéros de version spécifiques plutôt que d'utiliser uniquement "latest"
- Considérez l'utilisation de tags pour différentes variantes (ex : production, development)
- Documentez clairement le contenu et l'utilisation de chaque image

7.3 Création d'un token d'accès pour l'authentification

Pour une meilleure sécurité, vous pouvez utiliser un token d'accès au lieu de votre mot de passe pour vous connecter à Docker Hub.

CrÃľation et utilisation d'un token d'accÃĺs

```
# Les tokens sont créés via l'interface web de Docker Hub
# 1. Accédez à votre compte Docker Hub
# 2. Allez dans Account Settings -> Security
# 3. Cliquez sur "New Access Token"
# 4. Donnez un nom à votre token et configurez les permissions
# 5. Copiez le token généré (il ne sera plus accessible après)

# Utilisation du token pour l'authentification
docker login -u votre-username
# Collez le token quand le mot de passe est demandé
```

7.4 Téléchargement et utilisation des images partagées

Une fois vos images publiées sur Docker Hub, n'importe qui peut les télécharger et les utiliser.

Utilisation des images depuis Docker Hub

```
# Téléchargement des images
docker pull votre-username/db_car:latest
docker pull votre-username/backend_car:latest
docker pull votre-username/frontend_car:latest

# Démarrage des conteneurs à partir des images téléchargées
docker run -d --name cardb -p 3306:3306 \
  -e MARIADB_ROOT_PASSWORD=root \
  -e MARIADB_DATABASE=cardb \
  votre-username/db_car:latest

docker network create car-net
docker network connect car-net cardb

docker run -p 8080:8080 --name backend-car --net car-net \
  -e MARIADB_HOST=cardb \
  -e MARIADB_USER=root \
  -e MARIADB_PASSWORD=root \
  -e MYSQL_PORT=3306 \
  -d votre-username/backend_car:latest

docker run -d --name frontend-car -p 3000:80 --net car-net \
  votre-username/frontend_car:latest
```

Tâche à réaliser

Exercice 4 : Partage d'images sur Docker Hub
Suivez ces étapes pour partager vos images sur Docker Hub :

1. Créez un compte sur Docker Hub si vous n'en avez pas déjà un
2. Connectez-vous depuis le terminal :

```
docker login
```

3. Taguez vos images avec votre nom d'utilisateur (remplacez "votre-username") :

```
docker commit cardb votre-username/db_car:latest
docker tag carbackend votre-username/backend_car:latest
docker tag carfrontend votre-username/frontend_car:latest
```

4. Poussez vos images vers Docker Hub :

```
docker push votre-username/db_car:latest
docker push votre-username/backend_car:latest
docker push votre-username/frontend_car:latest
```

5. Vérifiez que vos images sont disponibles sur votre compte Docker Hub via l'interface web
6. Simulez l'utilisation sur une autre machine en supprimant d'abord les images locales :

```
docker rm -f cardb backend-car frontend-car  # Supprime les conteneurs
docker rmi votre-username/db_car:latest votre-username/backend_car:latest votre-username/frontend_car:latest  # Supprime les images
```

7. Téléchargez et utilisez vos images depuis Docker Hub :

```
docker pull votre-username/db_car:latest
docker pull votre-username/backend_car:latest
docker pull votre-username/frontend_car:latest
```

8. Redémarrez l'application en utilisant ces images

8 Orchestration avec Docker Compose

Concept fondamental

Docker Compose est un outil qui permet de définir et gérer des applications multi-conteneurs avec un simple fichier YAML. Au lieu d'exécuter plusieurs commandes `docker run` avec de nombreux paramètres, vous décrivez l'ensemble de votre stack d'application dans un fichier et la démarrez avec une seule commande.

8.1 Structure d'un fichier Docker Compose

Structure de base d'un fichier docker-compose.yml

```
version: '3'

services:
  service1:
    image: image1
    ports:
      - "8080:8080"
    environment:
      - VAR1=value1
    volumes:
      - volume1:/path/in/container
    depends_on:
      - service2

  service2:
    build: ./path/to/dockerfile
    networks:
      - my-network

volumes:
  volume1:

networks:
  my-network:
```

Point clé à retenir

Éléments essentiels d'un fichier Docker Compose :

- **version** : Spécifie la version du format Docker Compose
- **services** : Définit les conteneurs à créer
- **image/build** : Spécifie l'image à utiliser ou le chemin vers le Dockerfile
- **ports** : Mappage des ports entre l'hôte et le conteneur
- **environment** : Variables d'environnement
- **volumes** : Montages de volumes pour la persistance
- **depends_on** : Dépendances entre services (ordre de démarrage)
- **networks** : Réseaux auxquels connecter les conteneurs

8.2 Création d'un fichier Docker Compose pour notre application

Créons maintenant un fichier Docker Compose pour orchestrer notre application complète.

docker-compose.yml (Ãă crÃľer Ãă la racine du projet)

```
version: '3'

services:
  # Service de base de données
  db:
    image: mariadb:latest
    container_name: cardb
    ports:
      - "3306:3306"
    environment:
      - MARIADB_ROOT_PASSWORD=root
      - MARIADB_DATABASE=cardb
    volumes:
      - mariadb_data:/var/lib/mysql
    networks:
      - car-network
    healthcheck:
      test: ["CMD", "mysqladmin", "ping", "-h", "localhost"]
      interval: 10s
      timeout: 5s
      retries: 5

  # Service backend
  backend:
    image: carbackend
    container_name: backend-car
    build: ./backend
    ports:
      - "8080:8080"
    environment:
      - MARIADB_HOST=db
      - MARIADB_USER=root
      - MARIADB_PASSWORD=root
      - MYSQL_PORT=3306
    networks:
      - car-network
    depends_on:
      - db

  # Service frontend
  frontend:
    image: carfrontend
    container_name: frontend-car
    build: ./frontend
    ports:
      - "3000:80"
    networks:
      - car-network
    depends_on:
      - backend

volumes:
  mariadb_data:

networks:
  car-network:
    driver: bridge
```

Point d'attention

Points importants à noter dans ce fichier Docker Compose :

— Remarquez que nous faisons référence au service de base de données par le nom `db` plutôt que `cardb` dans les variables d'environnement du backend. Docker Compose configure automatiquement le DNS pour résoudre les noms de services.

— L'utilisation de `depends_on` garantit que les services démarrent dans le bon ordre, mais ne garantit pas que le service est prêt à recevoir des connexions. Pour une meilleure fiabilité, vous pouvez utiliser des healthchecks comme nous l'avons fait pour la base de données.

— Pour les services backend et frontend, nous avons inclus à la fois `image` et `build`. Cela permet à Docker Compose de construire l'image si elle n'existe pas encore, mais d'utiliser l'image existante si elle est déjà construite.

8.3 Utilisation de Docker Compose

Commandes de base de Docker Compose

```
# Démarrer tous les services définis dans docker-compose.yml
docker-compose up -d

# Voir les logs de tous les services
docker-compose logs

# Voir les logs d'un service spécifique
docker-compose logs backend

# Arrêter tous les services
docker-compose down

# Arrêter les services et supprimer les volumes
docker-compose down -v

# Reconstruire les images et redémarrer les services
docker-compose up -d --build
```

Tâche à réaliser

Exercice 5 : Orchestration avec Docker Compose

1. Assurez-vous que tous les conteneurs existants sont arrêtés :

   ```
   docker stop $(docker ps -a -q)
   ```

2. Créez un fichier docker-compose.yml à la racine de votre projet avec le contenu indiqué précédemment
3. Lancez l'application complète avec Docker Compose :

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

4. Vérifiez que tous les services sont démarrés :

   ```
   docker-compose ps
   ```

5. Testez l'application en accédant à http://localhost:3000
6. Consultez les logs pour déboguer d'éventuels problèmes :

   ```
   docker-compose logs
   ```

7. Modifiez l'un des fichiers source (par exemple, changez la couleur d'un élément dans le frontend)
8. Reconstruisez et redémarrez uniquement le service modifié :

   ```
   docker-compose up -d --build frontend
   ```

9. Vérifiez que vos modifications sont prises en compte
10. Arrêtez tous les services lorsque vous avez terminé :

    ```
    docker-compose down
    ```

Point clé à retenir

Avantages de Docker Compose par rapport aux commandes Docker individuelles :

- **Configuration déclarative** : Définissez l'ensemble de votre stack dans un fichier versionnable
- **Reproductibilité** : Garantit que l'application est déployée de manière cohérente sur tous les environnements
- **Simplicité** : Une seule commande pour démarrer/arrêter toute l'application
- **Isolation** : Crée automatiquement un réseau dédié pour les services
- **Gestion des dépendances** : Contrôle l'ordre de démarrage des services
- **Variables d'environnement** : Support pour les fichiers .env et la substitution de variables

9 Bonnes pratiques et optimisations

9.1 Optimisation des images Docker

Point clé à retenir

Techniques d'optimisation de la taille et des performances des images Docker :

- **Utiliser des images de base légères** comme Alpine Linux
- **Constructions multi-stage** pour éliminer les outils de build des images finales
- **Minimiser le nombre de couches** en combinant les commandes RUN
- **Nettoyage des caches** des gestionnaires de packages (`apt-get clean`, `npm cache clean`)
- **Utiliser .dockerignore** pour exclure les fichiers inutiles
- **Installer uniquement les packages nécessaires**
- **Attention à l'ordre des instructions** : placer les moins fréquemment modifiées en premier

Exemple d'optimisation d'un Dockerfile Node.js

```
# Avant optimisation
FROM node:14
WORKDIR /app
COPY . .
RUN npm install
EXPOSE 3000
CMD ["npm", "start"]

# Après optimisation
FROM node:14-alpine AS builder
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

FROM nginx:alpine
COPY --from=builder /app/build /usr/share/nginx/html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

9.2 Sécurisation des conteneurs Docker

Point d'attention

Pratiques essentielles pour la sécurité des conteneurs Docker :

- **Ne pas exécuter les conteneurs en tant que root** : Utiliser l'instruction USER pour définir un utilisateur non privilégié
- **Scanner les images pour les vulnérabilités** avec des outils comme Trivy, Clair ou Docker Scout
- **Utiliser des images officielles ou vérifiées** sur Docker Hub
- **Maintenir les images à jour** avec les correctifs de sécurité
- **Limiter les capacités et les privilèges** des conteneurs
- **Activer la signature et la vérification des images**
- **Utiliser des secrets pour les informations sensibles** plutôt que des variables d'environnement
- **Appliquer le principe du moindre privilège** dans la configuration des conteneurs

Exemple

Exemple d'utilisation d'un utilisateur non privilégié :

```
FROM node:14-alpine

# Création d'un utilisateur et groupe dédiés
RUN addgroup -S appgroup && adduser -S appuser -G appgroup

WORKDIR /app

COPY package*.json ./
RUN npm install

COPY . .

# Changement des permissions
RUN chown -R appuser:appgroup /app

# Passage à l'utilisateur non-root
USER appuser

EXPOSE 3000
CMD ["node", "app.js"]
```

9.3 Gestion des logs et monitoring

9.4 Gestion des logs et monitoring

Point clé à retenir

Recommandations pour la gestion des logs et le monitoring des conteneurs Docker :

- **Configurer les pilotes de logs** appropriés (json-file, syslog, fluentd, etc.)
- **Limiter la taille des logs** pour éviter de remplir le disque
- **Centraliser la collecte des logs** avec des outils comme ELK Stack ou Graylog
- **Implémenter le monitoring** avec Prometheus et Grafana
- **Utiliser des healthchecks** pour vérifier l'état des services
- **Configurer des alertes** pour les problèmes critiques
- **Suivre l'utilisation des ressources** avec des outils comme cAdvisor

Configuration des options de logging dans docker-compose.yml

```
services:
  backend:
    image: carbackend
    logging:
      driver: "json-file"
      options:
        max-size: "10m"
        max-file: "3"
```

9.5 Mise en place de healthchecks

Ajout de healthchecks dans docker-compose.yml

```
services:
  db:
    image: mariadb:latest
    healthcheck:
      test: ["CMD", "mysqladmin", "ping", "-h", "localhost"]
      interval: 10s
      timeout: 5s
      retries: 5

  backend:
    image: carbackend
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8080/actuator/health"]
      interval: 30s
      timeout: 10s
      retries: 3
      start_period: 40s
```

10 Conclusion et perspectives

10.1 Récapitulatif des concepts clés

Point clé à retenir

Concepts essentiels de Docker abordés dans ce cours-lab :

- **Conteneurisation** : Isolation des applications et leurs dépendances
- **Images Docker** : Templates immuables pour créer des conteneurs
- **Conteneurs** : Instances en cours d'exécution des images
- **Dockerfile** : Script déclaratif pour construire des images
- **Volumes** : Persistance des données entre les cycles de vie des conteneurs
- **Réseaux Docker** : Communication entre conteneurs
- **Docker Compose** : Orchestration de multi-conteneurs
- **Docker Hub** : Partage et distribution d'images

10.2 Bonnes pratiques à retenir

Point clé à retenir

Bonnes pratiques pour l'utilisation de Docker en production :

- **Images légères** : Utiliser des images de base minimalistes et des builds multi-stage
- **Sécurité** : Ne pas exécuter les conteneurs en tant que root et scanner les vulnérabilités
- **Immutabilité** : Traiter les conteneurs comme des entités jetables et sans état
- **Configuration** : Externaliser la configuration via des variables d'environnement
- **Logs** : Centraliser et limiter la taille des logs
- **Monitoring** : Implémenter des healthchecks et surveiller les performances
- **CI/CD** : Automatiser les builds et les tests des images Docker
- **Versions** : Tagger précisément les images et éviter l'utilisation du tag "latest" en production

10.3 Perspectives d'évolution

Pour aller au-delà de ce cours-lab, voici quelques pistes à explorer :

- **Orchestration avancée** avec Kubernetes pour la gestion de clusters de conteneurs
- **Service mesh** comme Istio ou Linkerd pour gérer les communications entre services
- **Serverless containers** avec AWS Fargate ou Azure Container Instances
- **Infrastructure as code** avec Terraform pour provisionner l'infrastructure Docker
- **GitOps** avec ArgoCD ou Flux pour le déploiement continu basé sur Git
- **Sécurité avancée** avec des outils comme Aqua Security ou Sysdig Secure
- **Stockage distribué** avec des solutions comme Rook ou Portworx

Bénéfice

Docker a révolutionné la façon dont nous développons, partageons et déployons des applications. En maîtrisant les concepts et techniques présentés dans ce cours-lab, vous disposez maintenant des compétences fondamentales pour conteneuriser vos applications, les partager efficacement et les déployer de manière cohérente dans différents environnements.
La conteneurisation n'est pas seulement un outil technique, c'est une approche qui favorise les bonnes pratiques DevOps : reproductibilité, isolation, portabilité et automatisation. Ces principes vous accompagneront quelle que soit l'évolution future des technologies de conteneurisation.

11 Références et ressources complémentaires

- **Documentation officielle de Docker** : `https://docs.docker.com/`
- **Docker Hub** : `https://hub.docker.com/`
- **Docker Compose reference** : `https://docs.docker.com/compose/compose-file/`
- **Docker security best practices** : `https://docs.docker.com/develop/security-best-practices/`
- **Dockerfile best practices** : `https://docs.docker.com/develop/develop-images/dockerfile_best-practices/`
- **Docker Curriculum** : `https://docker-curriculum.com/`
- **Play with Docker** (lab en ligne) : `https://labs.play-with-docker.com/`

Point clé à retenir

Pour continuer votre apprentissage, considérez les ressources suivantes :

- **Livres** : "Docker : Up & Running" (O'Reilly), "Docker in Practice" (Manning)
- **Cours en ligne** : Cours Docker sur Udemy, Pluralsight ou LinkedIn Learning
- **Certifications** : Docker Certified Associate (DCA)
- **Communautés** : Forums Docker, Stack Overflow, Reddit r/docker
- **Événements** : DockerCon, meetups locaux sur les conteneurs et DevOps