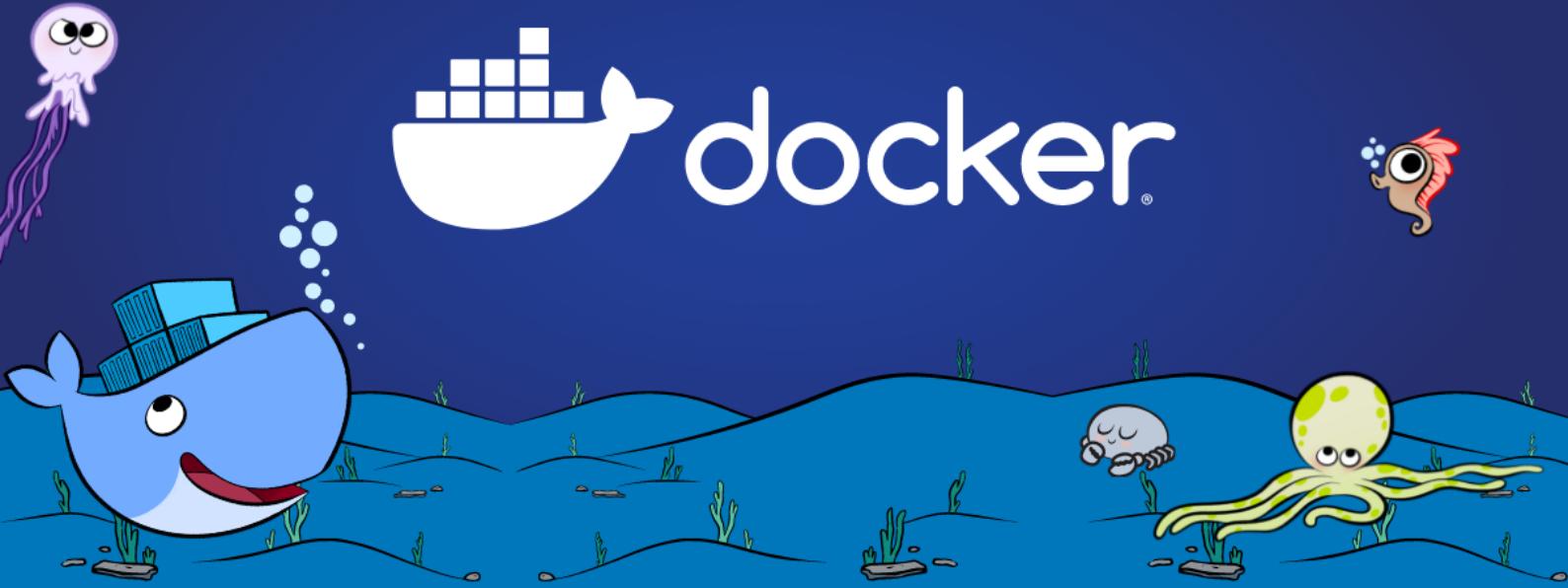


Docker - Guide pratique

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

Livre



Sommaire

1. Introduction	1
1.1. Préambule	1
1.2. Docker? C'est quoi ?	1
1.3. Pourquoi Docker et les conteneurs ?	6
1.4. Des environnements de développement et de production identiques	6
1.4.1. Exemple : une application NodeJs	7
1.5. Des environnements identiques	9
1.6. Des environnements conflictuels entre différents projets	9
1.7. Machines Virtuelles vs Conteneurs Docker	11
1.8. Un ordinateur dans l'ordinateur	11
1.9. Les machines virtuelles sont lourdes	13
1.9.1. Calcul des ressources restantes	14
1.9.1.1. Calcul des ressources restantes	14
1.9.1.2. Réflexion	15
1.9.1.3. Conclusion	15
1.10. Les conteneurs sont légers	17
2. Installation de Docker	21
2.1. Vue d'ensemble	21
2.2. Installation de Docker sur Windows	23
2.2.1. Le nouveau Terminal pour Windows	23
2.2.2. Téléchargement	25
2.2.3. Virtualisation d'un noyau Linux	26
2.2.3.1. Avec WSL	26
2.2.3.1.1. Prérequis : version du système d'exploitation	26
2.2.3.1.2. Prérequis : Configuration matérielle	27
2.2.3.1.3. Prérequis : Activer WSL	27
2.2.3.1.4. Mise à jour de WSL	28
2.2.3.1.5. Installation	28
2.2.3.1.6. Vérification de la version de WSL	28
2.2.3.2. Avec Hyper-V	29
2.2.4. Installation de Docker Desktop	30
3. Les images Docker	31
3.1. Introduction aux images Docker	31
3.2. Images et Conteneurs : Quelle différence ?	31
3.3. Les images Docker pré-construites	33
3.3.1. Préambule	33
3.3.2. Utiliser une image existante	34
3.4. Les images Docker personnalisée	37

3.4.1. Préambule	37
3.4.2. TD : Créer une image pour une application Node.js	38
3.4.2.1. Objectif	38
3.4.2.2. Préparation	38
3.4.2.3. Présentation des fichiers	38
3.4.2.4. Lancez l'application Node.js en Local	40
3.4.2.5. Dockerfile : Création de notre propre image Docker	42
3.4.2.5.1. Instruction : FROM	42
3.4.2.5.2. Instruction : COPY	43
3.4.2.5.3. Instruction : RUN	43
3.4.2.5.4. Instruction : WORKDIR	43
3.4.2.5.5. Instruction : CMD	44
3.4.2.5.6. Instruction : EXPOSE	45
3.4.2.5.7. Construction de l'image	46
3.4.2.5.8. Mapping de ports	48
3.4.3. Précisions sur les images Docker	51
3.4.3.1. Lecture seule	51
3.4.3.2. Comprendre les Couches d'images	52
3.5. Manager les images et les conteneurs	54
3.6. Stopper et redémarrer les conteneurs	54
3.7. Comprendre les modes attaché et détaché	54
3.8. Entrer dans un conteneur en marche	54
3.9. Le mode interactif	54
3.10. Suppression des images et des conteneurs	54
3.11. Supprimer un conteneur arrêté automatiquement	54
3.12. Inspecter un conteneur	54
3.13. Copier des fichiers dans et depuis un conteneur	54
3.14. Nommer et tagger des conteneurs et images	54
3.15. Mise en pratique	54
3.16. Partager des images	55
3.17. Push d'images sur Dockerhub	55
3.18. Pull et utilisation d'images partagées	55
3.19. Résumé	55
4. Gestion des images et des conteneurs	56
4.1. Lister les conteneurs	56
4.2. Démarrer, arrêter et redémarrer un conteneur	57
4.3. Comprendre le mode détaché et le mode attaché	58
4.4. Le mode interactif	60
4.5. Supprimer des images et des conteneurs	60
4.6. Supprimer automatiquement des conteneurs arrêtés	60
4.7. Inspecter des images	60

4.8. Copier des fichiers vers/depuis un conteneur	60
4.9. Nommer & Tagger des images et des conteneurs	61
4.10. Partager des images	61
4.11. Pusher des images vers un registre	61
5. Data et Volumes	62
5.1. Introduction	62
5.2. Différents types de données	62
5.3. Mise en pratique : Manipulation de données temporaires et permanentes	63
5.3.1. Analyse d'une application de démonstration	63
5.3.2. Comprendre la problématique	65
5.3.3. Introduction aux <i>Volumes</i>	72
5.3.4. Un premier essai raté	73
5.3.5. Les volumes à la rescousse !	77
5.3.6. Les Liaisons de répertoires (bind mounts)	81
5.3.7. Combiner et fusionner différents volumes	83
5.3.7.1. (NODEJS) Utiliser Nodemon dans un conteneur	88
5.3.7.2. (NODEJS) Windows, WSL2 et Bond Mounts	90
5.3.7.2.1. Solution efficace, simple, mais sale	90
5.4. Volumes et "Bind Mounts" : Résumé	90
5.5. Zoom sur les volumes en lecture seul	90
5.6. Gestion des volumes Docker	91
5.7. Utilisation de "COPY" vs "BIND MOUNTS"	91
5.8. Le fichier ".dockerignore"	91
5.9. Variables d'environnement et sécurité	91
5.10. Utiliser les arguments (ARG) lors d'un Build	91
5.11. Résumé	91
6. Networking : La communication entre conteneurs	92
6.1. Introduction	92
6.2. Présentation de l'application de démonstration	93
6.3. Communication	96
6.3.1. Préambule	96
6.3.2. Situation 1 : Du conteneur vers un serveur distant (HTTP)	97
6.3.3. Situation 2 : Du conteneur vers la machine hôte locale	97
6.3.4. Situation 3 : Du conteneur vers un autre conteneur	98
6.4. Mise en pratique : Crédit du conteneur et communication avec le serveur API REST	99
6.5. Mise en pratique : Faire communiquer le conteneur avec le localhost	103
6.6. Mise en pratique : Communication entre conteneurs [Solution basique]	106
6.7. Mise en pratique : Les Networks Docker [Solution élégante]	108
6.8. Les pilotes (drivers) de Docker Network	112
7. Construire des applications multi-conteneurs avec Docker	114
8. Docker Compose : Applications Multi-conteneurs qui ont de la classe !	115

9. Travailler avec des conteneurs utilitaires	116
10. Projet Laravel complexe et Docker	117
10.1. Introduction	117
10.2. La configuration cible	118
10.3. Préparation	120
10.4. Un conteneur Nginx(Serveur Web)	121
10.4.1. Son Image	121
10.4.2. Exposition de son port	122
10.4.3. Ajout d'un montage Bind Mount	122
10.5. Un conteneur PHP	123
10.5.1. Son Image	123
10.5.2. Référencement de l'image dans le docker-compose	125
10.5.3. Création du dossier SRC	126
10.5.4. Le port	127
10.6. Un conteneur MySQL	128
10.6.1. L'image	128
10.6.2. Les variables d'environnement ENV	129
10.7. Un conteneur Composer	131
10.7.1. Configuration du build	131
10.8. Créer une application Laravel avec le conteneur utilitaire	133
10.9. Lancer des services Docker Compose à la carte	136
10.10. Conteneurs et leurs dépendances	139
10.11. Le conteneur Artisan	140
10.12. Le conteneur NPM	141
10.13. Test des conteneurs outils	142
10.14. Conclusion	143
10.15. Docker Compose AVEC et SANS Dockerfiles	143
10.16. Bind Mount ou COPY : Quand ?	143
11. Déploiement de conteneurs Docker	146
11.1. Introduction	146
11.1.1. Objectif du chapitre	146
11.1.2. Conclusion	147
11.2. Du Développement au Déploiement	147
11.2.1. Les Conteneurs : Une Solution aux Problèmes de Déploiement	147
11.2.2. Déploiement de Conteneurs : Points Clés à Considérer	149
11.3. Processus de déploiement et fournisseurs	150
11.3.1. Introduction avec un Exemple Simple	150
11.3.1.1. Choix d'un Fournisseur de Serveurs Distants	151
11.3.1.2. Conditions Préalables et Options Gratuites	152
11.3.1.3. Lancement d'une Instance EC2	153
11.4. Déploiement d'une Application Node Basique sur AWS EC2	153

11.4.1. Étapes Principales pour Déployer une Application conteneurisée sur EC2	154
11.4.2. Préparation de l'Application conteneurisée	155
11.5. Différence entre le Développement et la Production avec Docker	156
11.5.1. Mode Développement	156
11.5.2. Mode Production	157
11.5.3. Flexibilité entre Développement et Production	157
11.5.4. Conclusion	157
11.6. Déploiement d'une Application de Démonstration sur un Serveur Web avec AWS EC2 ..	158
11.6.1. Mise en place	158
11.6.2. Lancement d'une Instance EC2 et Connexion via SSH	160
11.6.3. Installation de Docker sur l'Instance EC2	168
11.6.4. Déploiement de l'Image Docker sur une Machine Distante	170
11.6.5. Conclusion	173
11.7. Exécution de l'Image Docker Poussée sur la Machine Distante (Instance EC2)	173
11.8. Gestion et mise à jour du conteneur/image	176
11.8.1. Mise à jour de l'image et de l'application	176
11.8.2. Arrêter et Supprimer l'Instance EC2	178
11.8.3. Conclusion	180
11.9. Inconvénients de l'approche actuelle	180
11.10. Du déploiement manuel aux services managés	182
11.11. IMPORTANT : Avertissement concernant l'utilisation de AWS ECS (Elastic Container Service)	185
11.12. Déploiement d'une Application avec AWS ECS	186
11.12.1. Avertissement	186
11.12.2. Configuration de ECS	186
11.12.3. Déploiement de Conteneurs	187
11.12.3.1. Choix du Conteneur	187
11.12.3.2. Configuration du Conteneur	187
11.12.3.3. Options Avancées	187
11.12.3.4. Configuration des Tâches	187
11.12.4. Configuration du Service et du Cluster	187
11.12.5. Finaliser le Déploiement	187
11.12.6. Mise à Jour des Conteneurs	188
11.12.7. Conclusion	188



1. Introduction

Version 1.0.0 | Dernière mise à jour : 17/11/2024 | Auteur : Bauer Baptiste

1.1. Préambule

Le cours que vous allez lire va aborder deux technologies qui facilitent le déploiement et le développement d'applications complexes :

- Docker
- Kubernetes

Certainement, qu'en ce moment même, vous n'avez absolument aucune idée de ce qu'est Docker et à quoi cela sert concrètement.

Docker est lié à ce que l'on appelle des **containers**.

Tout au long de ce cours, vous recevrez des commandes à tester avec leur explication.

Ainsi, vous saurez comment :

- Installer et utiliser Docker localement sur votre machine.
- Utiliser Docker dans votre projet simple comme dans les plus complexes.
- Déployez ces projets et des applications.
- Apprendre ce qu'est Kubernetes et son lien avec Docker.

Vous rencontrerez beaucoup de petits exemples pour vous plonger dans des cas pratiques et concrets.

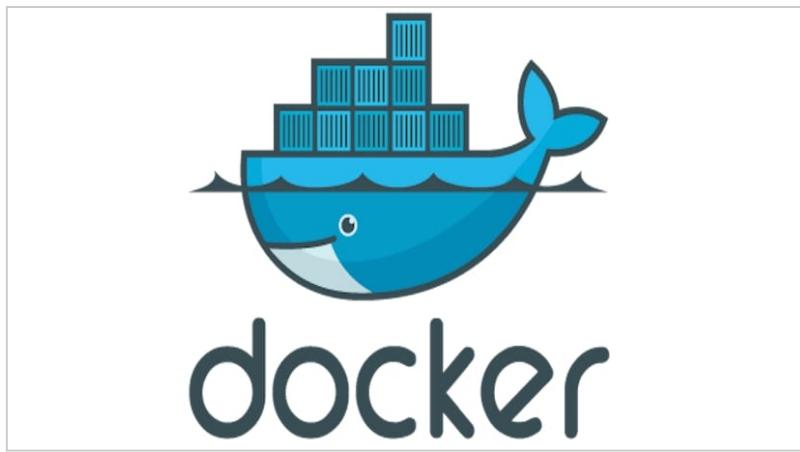
Nous allons commencer de zéro.

Pré-requis pour suivre ce cours



- Aucune connaissance préalable de **Docker** n'est requise.
- Aucune connaissance préalable de **Kubernetes** n'est requise non plus !

1.2. Docker? C'est quoi ?



Voici une définition académique de Docker :

Docker est une technologie de conteneurisation : un outil pour créer et gérer des conteneurs.

Mais finalement, nous ne sommes pas plus avancés !

Qu'est-ce qu'un **conteneur** ? et pourquoi aurions-nous besoin, à notre niveau, de les utiliser, puisque jusqu'à présent, nous vivions très bien sans ?

Vous étudiez le développement d'applications n'est-ce pas ? Dans ce cas, vous connaissez au moins les bases d'un langage de développement comme le **Javascript**, le **PHP**, **Java** ou **C#**.

Donc, vous n'êtes pas sans savoir qu'une application est le produit d'un long processus d'écriture de lignes de codes, sur une machine de développement fonctionnant avec des librairies. L'ensemble tournant dans un environnement particulier.

Par exemple, imaginons que nous développons une application de gestion pour le football. Cette application permet d'ajouter des joueurs et leur palmarès dans des équipes et de télécharger la composition de ces équipes en format PDF.

Au lieu de programmer de zéro comment convertir nos données en PDF, nous allons utiliser des morceaux de code déjà écrits par d'autres développeurs. Ces morceaux de code sont regroupés dans ce qu'on appelle une 'librairie'. En utilisant une librairie, nous gagnons du temps et nous assurons que la fonctionnalité est déjà bien testée et fonctionnelle.

Attention toutefois, notre application est développée en PHP 7.2, une version un peu ancienne. Il faudra donc choisir une librairie qui est compatible avec cette version de PHP. Si nous ne faisons pas attention à cette compatibilité, nous risquons de rencontrer des problèmes ou des 'bugs' lors de l'utilisation de notre application. C'est un peu comme s'assurer que les pièces d'un puzzle s'emboîtent bien ensemble.

Jusqu'à maintenant, nous ne voyons pas vraiment de problèmes !

Sur notre machine de développement, nous devrons seulement installer PHP 7.2 et la librairie compatible. Idem sur le serveur de production.

Machine de développement : Windows 11

Environnement PHP 7.2

index.php

equipes.php

joueurs.php

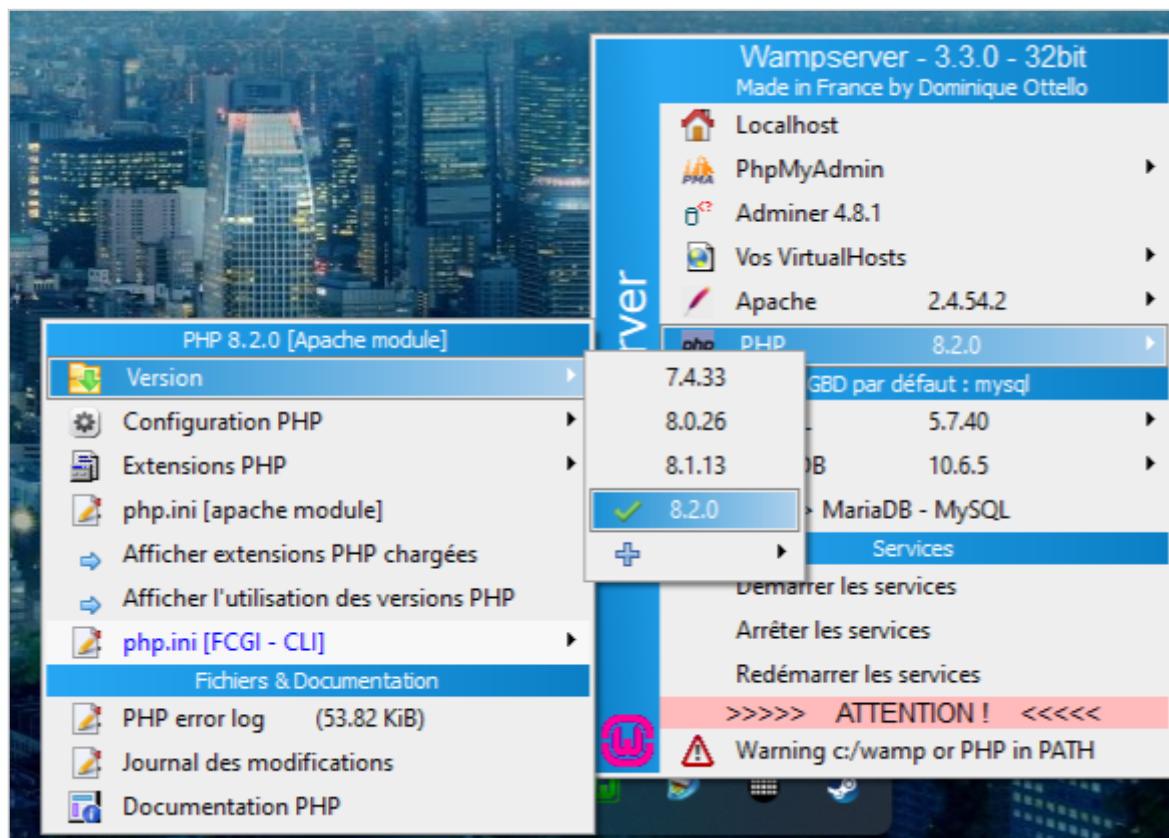
Fichier3.php

Librairie_export_pdf.php

Exemple : Application de gestion d'équipes de Football

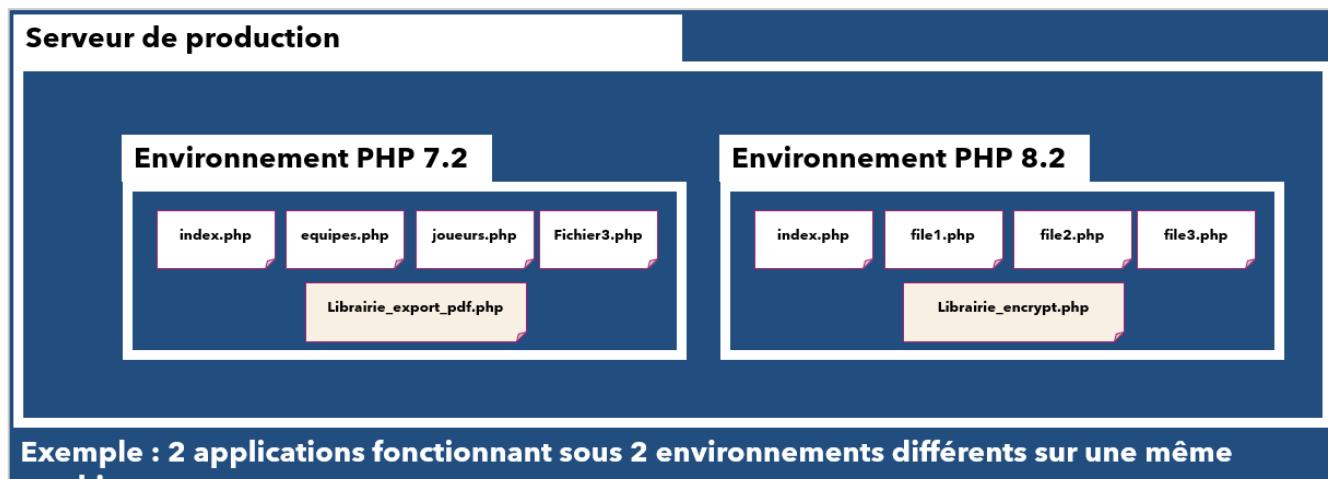
Maintenant, imaginons qu'une autre mission consistera à développer une nouvelle application sous PHP 8.2, et qui aura besoin entre autres, d'une librairie d'encryptage de mot de passe par exemple.

Sur votre machine de développement, vous utilisez sûrement **WampServer** qui permet en quelques clics de changer de version de PHP en cours :



Les choses se compliqueront durant la mise en production :

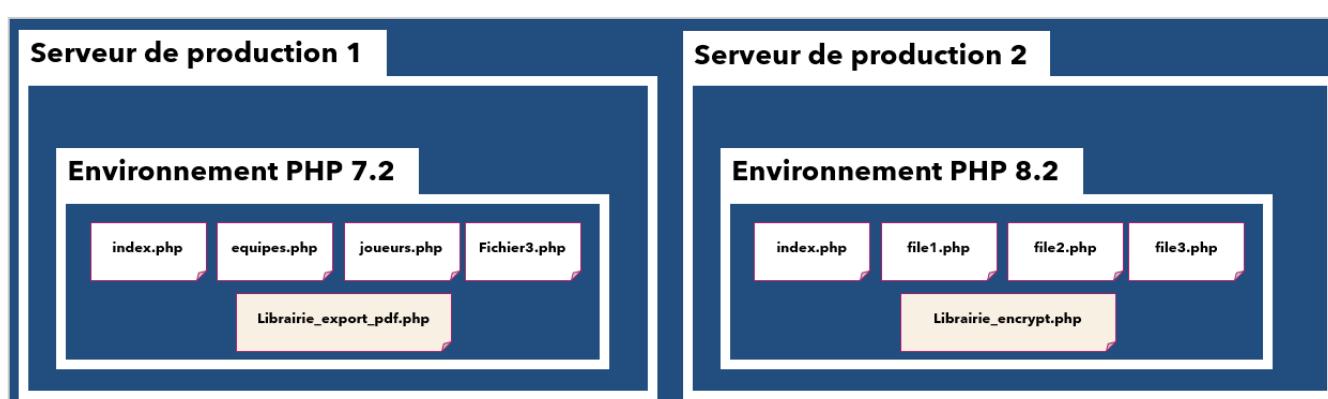
Comment faire tourner sur votre serveur les deux applications en même temps ?



La première solution qui nous vient à l'esprit est aussi la plus coûteuse :

- Créer et maintenir 2 serveurs différents pour faire tourner nos applications.

On pourrait imaginer tout un tas de solutions, mais qui demanderaient de gros efforts de configuration rendant la maintenance des systèmes difficiles et sûrement instables !



Fermez les yeux, imaginez maintenant un outil miraculeux, qui, en quelques clics, vous permettrait de créer autant d'environnement que nécessaire ! indépendants les uns des autres, isolés, mais ouverts au dialogue , c'est là que **Docker** entre en scène !

Vous ne voyez toujours pas l'intérêt ?



Partons faire un pique-nique avec notre beau panier en osier !

Dans ce panier, il y a des couverts, du pain, de la nourriture, une nappe ! tout ce qui est nécessaire pour faire un pique-nique.

Chaque fois que vous prenez ce panier, vous retrouverez à chaque fois : les mêmes couverts, la même nappe, la même vaisselle. (*Surement pas la même nourriture, surtout si le panier n'a pas servi depuis longtemps, mais vous comprenez l'idée !*).

Sachez alors qu'un **conteneur**, est comme ce panier, chaque fois que vous en aurez besoin, vous pourrez le prendre partout où vous le désirez, et vous aurez la certitude d'avoir exactement les mêmes outils !

De plus un **conteneur** peut se répliquer à l'infini ou presque.



Le mot **conteneur**, évoque sûrement dans votre esprit cette image :



Et vous aurez raison !

Car un conteneur est une boîte, dans laquelle sont stockées des marchandises isolées du reste du monde. Les marchandises d'un conteneur ne sont pas mélangées avec celles d'un autre conteneur.

De plus, prenons l'exemple d'une marchandise qui nécessite d'être conservée au frais. Dans ce cas, le conteneur doit être équipé d'un système de refroidissement autonome. Pour éviter que la marchandise ne se renverse durant le transport, il est aussi essentiel de prévoir des systèmes de fixation à l'intérieur du conteneur. Grâce à ces équipements, le conteneur peut être chargé sur n'importe quel navire, stocké dans n'importe quel entrepôt ou port, tout en assurant que les marchandises sont maintenues à la bonne température et bien fixées.

Les conteneurs Docker suivent cette même philosophie. À l'intérieur d'un conteneur Docker, nous pouvons intégrer tout ce qui est nécessaire pour faire fonctionner notre application :

- Un environnement spécifique pour des applications web, comme PHP entre autres.
- Des bibliothèques dans les versions requises.

- La version spécifique de notre application.
- Tous les services essentiels, qu'il s'agisse de la gestion des e-mails, de la persistance des données, et plus encore.

Tant qu'une machine est équipée de Docker, nous pourrons y exécuter notre conteneur et utiliser notre application sans nous préoccuper des prérequis nécessaires à son fonctionnement. En effet, tout ce dont nous avons besoin se trouve déjà à l'intérieur du conteneur !

Docker n'est qu'un outil qui sert à construire des conteneurs !

A retenir



- Un même conteneur, s'il est dupliqué contiendra la même application et le même contexte d'exécution, peu importe la machine qui l'exécute, tant qu'elle contient Docker !
- Docker est utilisable sur tous les systèmes d'exploitations modernes.
- Docker simplifie la création et la gestion des conteneurs.

1.3. Pourquoi Docker et les conteneurs ?

Dans le chapitre précédent, au travers d'illustrations telles que le panier de pique-nique et le conteneur de marchandises, nous avons illustré le concept de conteneur.

Vous avez certainement compris que cette technologie facilite la création d'environnements distincts pour l'exécution d'applications.

Toutefois, Est-ce qu'utiliser des conteneurs est aussi utile qu'important ? Quels sont les avantages de l'utilisation de conteneurs ?

Pourquoi voudrions-nous des "boîtes" indépendantes et standardisées pour exécuter nos applications ?

Explorons les raisons pour lesquelles un développeur pourrait choisir d'utiliser des conteneurs dans le développement et le déploiement d'applications.

Bien que nous ayons déjà pris l'exemple d'une application de gestion d'équipe de football, approfondissons la question avec un cas d'utilisation légèrement plus technique.

1.4. Des environnements de développement et de production identiques

Lorsque nous développons une application, nous avons besoin d'un environnement de développement pour exécuter notre code et tester notre application. Ensuite, lorsque le moment est venu de déployer notre application, nous avons besoin d'un environnement de production pour exécuter notre application.

Dans un monde idéal, ces environnements **devraient être identiques**. Cependant, dans la réalité, il est difficile de garantir que les environnements de développement et de production sont

strictement identiques.

Par exemple, les développeurs peuvent utiliser des versions différentes de logiciels, des configurations différentes, des systèmes d'exploitation différents, etc.

Cela peut entraîner des problèmes de compatibilité et des bugs qui ne sont pas détectés avant le déploiement de l'application.

Les conteneurs peuvent nous aider à résoudre ce problème en nous permettant de créer des environnements de développement et de production identiques.

Nous pouvons créer un conteneur pour notre environnement de développement et un autre pour notre environnement de production.

Ces conteneurs peuvent être créés à partir de la même image de conteneur, ce qui garantit que les environnements sont identiques.

1.4.1. Exemple : une application NodeJs

Prenons l'exemple d'une application NodeJs. Vous ne connaissez pas NodeJs ? Pas de panique ! Pour résumer, NodeJs est un environnement d'exécution JavaScript côté serveur qui permet d'exécuter du code JavaScript en dehors d'un navigateur.

Car à l'origine, JavaScript était un langage de programmation utilisé uniquement dans les navigateurs web.

Aujourd'hui, NodeJs est utilisé pour créer des applications web, des applications mobiles, des applications de bureau, des outils de ligne de commande, etc.

Pour exécuter une application NodeJs, nous avons besoin d'un environnement NodeJs.

Cet environnement est composé de NodeJs, de notre code source et de toutes les dépendances de notre application.

Pour créer un environnement de développement, nous pouvons installer NodeJs sur notre ordinateur et y placer notre code source.

Le langage de programmation JavaScript est évolué rapidement et de nouvelles versions sont publiées régulièrement. Par conséquent, NodeJs est également mis à jour régulièrement pour prendre en compte les nouvelles fonctionnalités Javascript.

Voici un exemple de code source d'une application NodeJs simple:

```
1 import axios from 'axios';
2 const {data} = await axios.get('https://jsonplaceholder.typicode.com/users');
3 console.log(data);
```

Il s'agit d'une application NodeJs simple qui utilise la bibliothèque Axios pour interroger une adresse web distante, un service, qui renvoie en réponse une liste d'utilisateurs sous forme d'un tableau. On appelle ce genre de service Web une **API** (Application Programming Interface).

```
const { data } = await axios.get('https://jsonplaceholder.typicode.com/users');
```

Et ce que j'ai encadré en rouge, c'est un mot clé spécifique au langage JavaScript `await`. Utilisable de la sorte, c'est-à-dire à l'extérieur d'une fonction `async` dans un module, depuis la version **14.3** de **NodeJs**.

En d'autres mots si vous voulez exécuter ce code, il vous faudra une version de **NodeJs** supérieure ou égale à la **14.3**.

Nous pourrions avoir cette version installée sur notre machine locale, dans notre environnement de développement. Si nous souhaitons que notre application soit déployée sur un serveur distant afin qu'elle soit accessible à tous, nous devons également installer NodeJs sur ce serveur. Mais comment pouvons-nous nous assurer que la version de NodeJs installée sur notre serveur est la même que celle installée sur notre machine locale (version ≥ 14.3) ?

Si le serveur distant utilise une version plus ancienne de NodeJs, notre application ne fonctionnera pas tout simplement pas !

Nous aurons le message d'erreur suivant :

```
(node:3936) ExperimentalWarning: The ESM module loader is experimental.  
file:///app.mjs:4  
  await axios.get('https://jsonplaceholder.typicode.com/users')  
^^^^^  
  
SyntaxError: await is only valid in async function
```

Nous ne pourrons jamais avoir la certitude absolue que les environnements de développement et de production sont identiques.

JAMAIS ! C'est impossible.

Un **environnement de développement** est toujours différent d'un environnement de production. L'environnement est constitué de plusieurs composants, tels que le système d'exploitation, les versions des logiciels, les configurations, etc.

Cependant, nous pouvons essayer de rendre ces environnements aussi similaires que possible.

Et c'est là que les conteneurs entrent en jeu.

Nous pouvons créer un conteneur pour notre environnement de développement et un autre pour notre environnement de production.

Ces conteneurs peuvent être créés à partir de la même image de conteneur, ce qui garantit que les environnements sont identiques.



Cela doit être facile de **partager un environnement de développement commun** entre les membres de l'équipe !

1.5. Des environnements identiques

De la même manière que nous voulons que nos environnements de développement et de production soient identiques, nous voulons également que les environnements de développement de notre équipe ou de notre organisation soient identiques.

Cela permet de s'assurer que tous les développeurs travaillent dans le même environnement sur le même code source.

Cela permet également de s'assurer que les problèmes de compatibilité et les bugs sont détectés le plus tôt possible.

Si l'on reprend l'exemple de notre application NodeJs précédente. Imaginez que vous n'avez pas utilisé NodeJs depuis très longtemps et que vous avez une version 10 sur votre machine locale. Vous ne pourrez pas modifier le code source de l'application et l'exécuter sur votre machine sans quelques difficultés.

Encore une fois, vous vous dites que cela n'est pas un problème, vous pouvez mettre à jour NodeJs facilement. Et vous avez raison. Mais cela n'est qu'un exemple pédagogique, la réalité est bien plus complexe. Et ce n'est pas une ou deux librairies dont vous aurez à gérer les versions, mais des dizaines, voir des centaines.

Et si vous travaillez en équipe, vous voulez que chaque membre de l'équipe utilise la même version de NodeJs, la même version des librairies, la même version des outils de développement, etc. Mais, vous ne pouvez pas vous permettre de perdre du temps à mettre à jour les versions des librairies de chacun des membres de l'équipe.

Et c'est là que les conteneurs entrent en jeu ...



Les divergences entre les versions d'environnement peuvent souvent être à l'origine de conflits lorsqu'il s'agit de travailler sur différents projets.

1.6. Des environnements conflictuels entre différents projets

Lorsque nous travaillons sur plusieurs projets, nous pouvons avoir des environnements conflictuels entre eux.

Par exemple, nous pouvons avoir un projet qui utilise une version spécifique d'une librairie, tandis

qu'un autre projet utilise une version différente de la même librairie.

Parfois même certaines versions récentes de librairies ne sont pas compatibles avec d'autres entraînant des erreurs qui pourraient être difficiles à traiter.

Entre deux projets, il faudra alors reparamétrer les versions des librairies, des outils de développement, etc.

Cela peut être très chronophage, source d'erreurs et absolument pas passionnant.

Et c'est là que les conteneurs entrent en jeu ...



Portables :
Les conteneurs Docker peuvent être exécutés sur n'importe quel système d'exploitation qui prend en charge la technologie Docker, ce qui facilite le transfert et le déploiement d'applications entre différents environnements.

Modulaires :
Les conteneurs Docker sont conçus pour encapsuler des parties spécifiques d'une application, ce qui les rend interchangeables et réutilisables.

Isolés :
Chaque conteneur Docker fonctionne de manière isolée, garantissant que l'application à l'intérieur du conteneur reste stable et cohérente indépendamment des autres conteneurs ou des changements dans l'environnement hôte.

Avec Docker, nous pouvons créer un conteneur pour chaque projet. Nous n'avons plus besoin d'installer les librairies, les outils de développement, etc. sur notre machine locale à chaque fois que nous travaillons sur un nouveau projet, car tout est déjà installé dans le conteneur et pas sur notre machine locale.

Changer de projet devient aussi simple que de changer de conteneur.

Durant la suite de ce cours, vous allez apprendre tout ce qu'il y a de nécessaire pour créer vos propres conteneurs et résoudre les problèmes que nous venons d'évoquer.

Vous êtes maintenant déterminé à apprendre Docker ? Alors, allons-y !



Nous voulons avoir **EXACTEMENT** le même environnement sur la machine de développement que sur celle de production !

Pourquoi ? Parce que nous voulons avoir l'assurance que l'application fonctionne **EXACTEMENT** comme durant les phases de tests !!

1.7. Machines Virtuelles vs Conteneurs Docker

Nous savons maintenant ce que sont Docker et les conteneurs et pourquoi les utiliser.

Les problèmes résolus par **Docker** et les **conteneurs** ont du sens et nous les avons vus précédemment.

Nous comprenons alors facilement pourquoi Docker est si populaire !!



Mais si vous avez aussi compris les principes de la virtualisation, vous n'êtes pas sans savoir que nous pouvons installer tous les environnements dont nous avons besoin sur une machine virtuelle et les utiliser comme nous le souhaitons :

- Nous pouvons installer un système d'exploitation sur une machine virtuelle et l'utiliser comme un ordinateur normal.
- Nous avons des terminaux indépendants de la machine hôte et exécuter les commandes que nous voulons pour installer ou configurer des logiciels.

Nous pouvons même créer des machines virtuelles à la volée et les détruire quand nous n'en avons plus besoin.

Dans ce cas, vous vous demandez peut-être pourquoi utiliser Docker alors que nous avons déjà des machines virtuelles qui peuvent faire la même chose.

C'est une question légitime et nous allons y répondre dans cette section.

1.8. Un ordinateur dans l'ordinateur



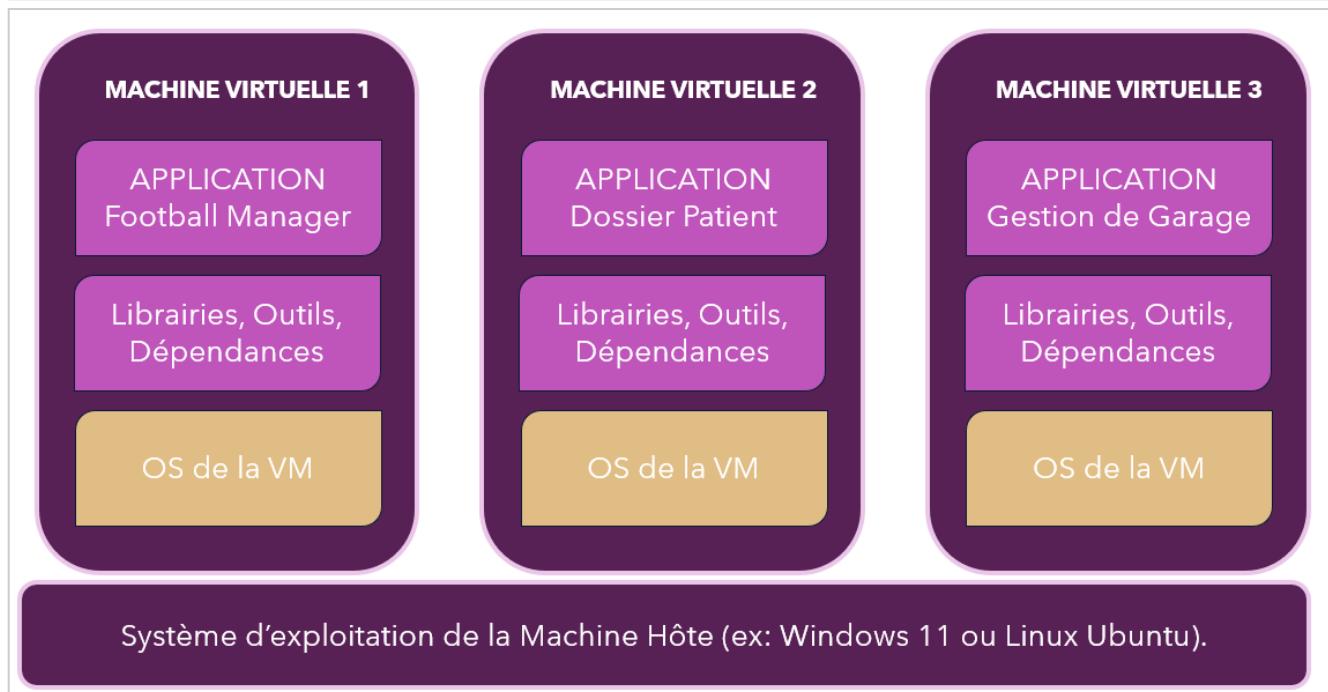
Pour virtualiser un système d'exploitation, nous avons besoin d'un logiciel nommé l'**hyperviseur** qui va créer une **machine virtuelle** et émuler le matériel pour que le système d'exploitation puisse s'exécuter.

Autrement dit : nous créons un ordinateur dans l'ordinateur.

Il existe plusieurs hyperviseurs, mais les plus connus sont **VirtualBox**, **VMWare** ou **HyperV**. Ce logiciel s'installe sur une première machine physique que l'on appelle la **machine hôte**, car c'est lui l'hôte, celui qui accueille **les machines virtuelles**.

La machine hôte possède son propre système d'exploitation et ses propres ressources matérielles (mémoire, processeur, disque dur...) Elle va donc partager ses ressources avec les **machines virtuelles** qu'elle va créer.

Donnant ainsi **aux machines virtuelles une totale indépendance**. Elles possèdent leur propre système d'exploitation et leurs propres ressources matérielles. Elles peuvent donc être utilisées comme un ordinateur normal.



Comme le représente le schéma ci-dessus, vous voyez que les machines virtuelles sont complètement autonomes les unes des autres, isolées dans leur propre coquille bien qu'elles soient toutes sur la même machine hôte.

1.9. Les machines virtuelles sont lourdes.



Les avantages que nous avons étudiés précédemment sur **Docker**, sont aussi valables pour les machines virtuelles.

- Mais alors pourquoi ne pas utiliser les machines virtuelles et devoir apprendre une nouvelle technologie ?

Pour répondre à la question, je vous propose le petit exercice de calcul suivant : calculons les ressources systèmes restant sur une machine hôte après la réaction d'une VM et menons une réflexion.

1.9.1. Calcul des ressources restantes



Voici les caractéristiques d'une machine hôte et d'une machine virtuelle installée :

Rappels

- Les ressources d'une machine virtuelle sont définies lors de son installation.
- Les ressources attribuées à une machine virtuelle ne peuvent plus être attribuées à une autre machine virtuelle ou à la machine hôte.



Retenez bien que : si vous allouez 1To de disque dur à une machine virtuelle, alors 1To de disque dur ne pourra plus être utilisé par une autre machine virtuelle ou par la machine hôte, et cela, **même si la machine virtuelle n'utilise pas tout le disque dur alloué**.

Machine	RAM	CPU	Disque dur
Machine hôte	8 Go	4 cœurs	1 To
Machine virtuelle	2 Go	2 cœurs	20 Go

1.9.1.1. Calcul des ressources restantes

Question 1 : Combien reste-t-il de ressources sur la machine hôte ?

Lors de la création d'une VM, les ressources allouées à la VM sont déduites des ressources de la machine hôte.

Pour calculer les ressources restantes, il faut soustraire les ressources de la VM aux ressources de la machine hôte.

- RAM : 8 Go - 2 Go = 6 Go
- CPU : 4 coeurs - 2 coeurs = 2 coeurs
- Disque dur : 1 To - 20 Go = 980 Go

Question 2 : Combien de machines virtuelles de ce type peut-on installer sur la machine hôte ?

Pour répondre à cette question, il suffit de diviser les ressources restantes par les ressources de la VM.

Question 3 : Puis-je installer une troisième machine virtuelle de ce type sur la machine hôte, et ne la démarrer que si les deux autres machines virtuelles sont arrêtées ?

Non, car les ressources allouées à une machine virtuelle ne peuvent pas être utilisées par une autre machine virtuelle ou par la machine hôte.

Nous sommes donc limités à deux machines virtuelles de ce type sur la machine hôte.

1.9.1.2. Réflexion

Voici quelques remarques qui vont nourrir notre réflexion :

- Sur 2 VMs, nous sommes obligés d'installer deux systèmes d'exploitation.

Par exemple : si ma machine hôte est sous Windows et que je veux installer une VM sous Windows, alors je dois installer intégralement Windows sur la nouvelle VM. De même, si je veux installer une VM sous Linux, je dois installer intégralement Linux sur la VM.

Cela prend beaucoup de place sur le disque dur de la machine hôte et beaucoup de ressources systèmes pour installer au final 2, 3 ou 4 fois les fichiers systèmes de Windows ou de Linux.

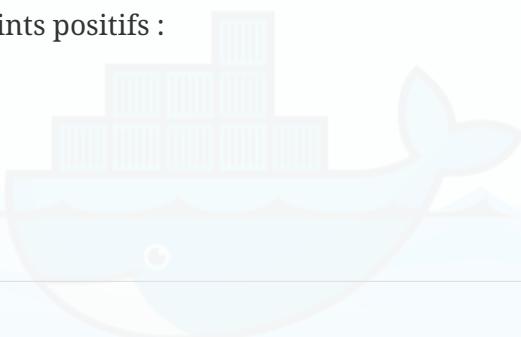
Plus il y a de VMs, plus la machine est ralentie, car ses ressources sont consommées.

1.9.1.3. Conclusion

Les machines virtuelles sont lourdes ! Mais toujours utiles et indispensables dans certains cas.

Vous ne pourrez jamais mettre par exemple Windows Server 2022 dans un conteneur, mais vous pourrez le faire sur une VM par exemple.

Voici un tableau illustrant les points positifs :



Avantages et inconvénients des VMs

POINTS POSITIFS

Environnements séparés et indépendants

Environnements spécifiques

La configuration peut être partagée et reproduite

POINTS NEGATIFS

Perte d'espace, duplication redondante

La performance est ralentie

Reproduire une VM sur une autre machine hôte: difficile et hasardeux

Notez le point négatif suivant :

"Reproduire une VM sur une autre machine hôte est parfois difficile et hasardeux".

Cela est dû au fait que les VMS installent des pilotes spécifiques à la configuration de la machine hôte.

- La carte réseau de la VM dépendra de la carte réseau de la machine hôte.
- La carte graphique de la VM dépendra de la carte graphique de la machine hôte.
- Le gestionnaire des processus qui tourne sur votre machine virtuelle dépendra de son OS bien entendu, mais aussi du processeur de la machine hôte.

Si vous changez de machine hôte, vous devrez réinstaller les pilotes de la VM pour qu'elle fonctionne correctement.

Cela peut être très chronophage et source d'erreurs.

Alors, toujours partant de créer une VM pour seulement avoir un environnement de développement en PYTHON ?

Je crois que vous avez compris l'idée !

Nous allons donc appeler sur le ring le prochain challenger : Docker !



1.10. Les conteneurs sont légers.

Sur une machine hôte sur laquelle nous voulons installer des conteneurs, il nous faudra toujours un système d'exploitation : Windows, Linux ou MacOs.

Mais nous n'installerons pas un ensemble de "machines dans la machine" via un hyperviseur.

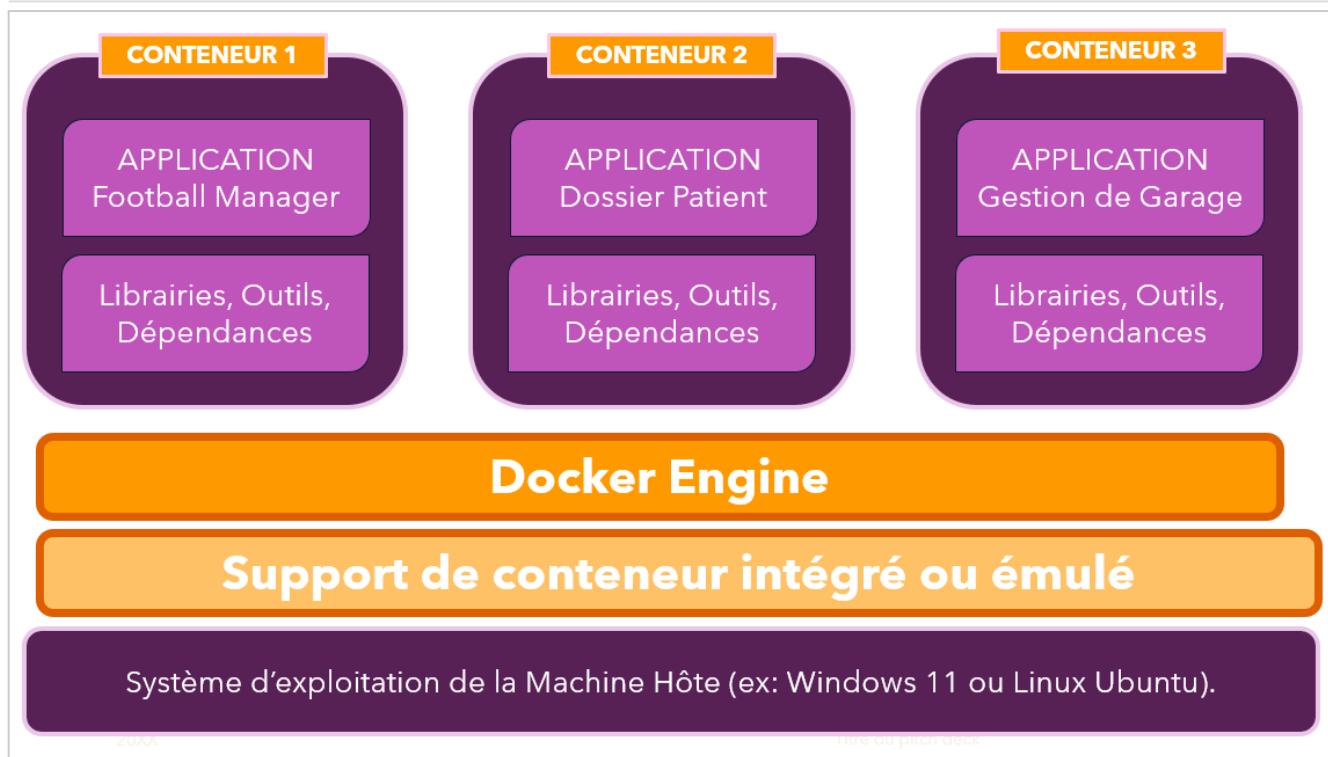
À la place, nous allons avoir besoin d'un support de conteneur intégré (ou ***built-in container support*** en anglais) au système d'exploitation de la machine hôte ou une émulation de ce support.

Et c'est là que Docker entre en jeu ! Lors de son installation, Docker fournit un tout **petit outil et léger** nommé : "**Docker Engine**".

Cet outil va nous permettre de créer des conteneurs et de les exécuter sur notre machine hôte !!

Voyons un schéma qui illustre la place des conteneurs dans l'architecture de votre ordinateur :





Et regardez les conteneurs :

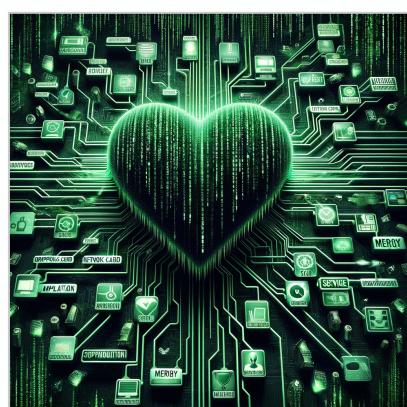
Ils sont beaucoup moins lourd que les VMs car il n'y a pas de système d'exploitation.

Nos applications seront donc plus rapides à démarrer et à exécuter. Sur le conteneur 1, il y a seulement notre application de Football ainsi que les librairies nécessaires à son exécution. Rien de plus !!

Vraiment? bon ... j'ai menti ... il y a bien un système d'exploitation dans le conteneur, mais vraiment très léger !!

La machine hôte va seulement partager **son noyau** avec les conteneurs MAIS pas l'intégralité de son système d'exploitation. C'est pour cela que le conteneur doit embarquer des petits bouts de système d'exploitation qui lui sont nécessaires pour fonctionner et rien de plus.

J'imagine que vous avez bloqué sur le mot "**noyau**" ?



Le noyau est un terme que l'on rencontre souvent en informatique, son nom est "kernel" en anglais. C'est le cœur du système d'exploitation. C'est la partie qui permet au monde matériel de votre machine : carte graphique, carte réseau, processeur, mémoire, disque dur, etc. de communiquer

avec le monde logiciel : les applications, les bibliothèques, les services, etc.

Le noyau est donc la partie la plus importante du système d'exploitation.

A contrario des VMs, les conteneurs n'ont donc pas à se soucier de la compatibilité avec le matériel de la machine hôte, car ils partagent le même noyau.

Ainsi, les conteneurs peuvent se partager d'une machine à une autre, et sont beaucoup plus légers et rapide à mettre en place !

Un autre point non négligeable : La gestion des ressources systèmes est beaucoup plus fine avec les conteneurs qu'avec les VMs.

Un conteneur ne consomme que ce qu'il a besoin, et pas plus ! Si pendant 10 min, il a besoin de 4 Giga de Ram ... il va les prendre, mais les rendra aussitôt après !

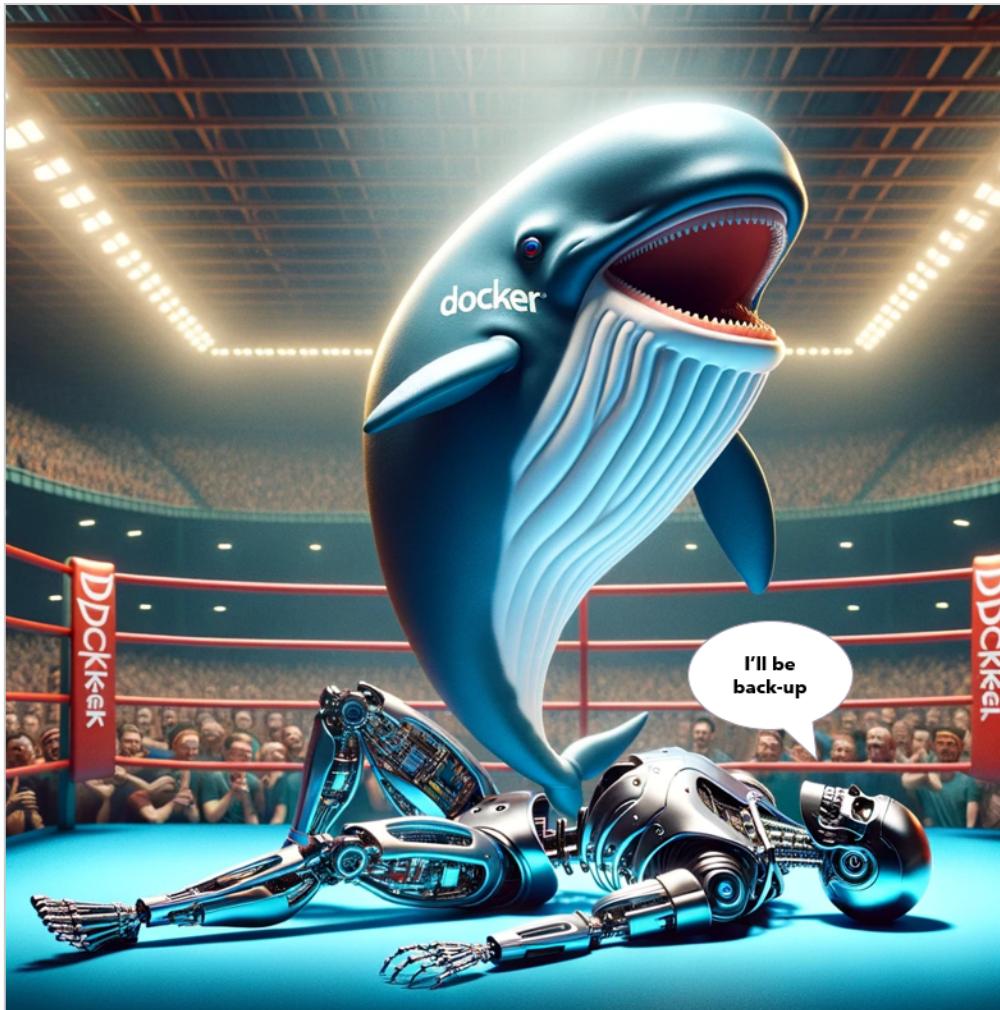
L'usage de l'espace disque est dynamique : il augmente ou diminue en fonction des besoins du conteneur.

Même principe avec les coeurs du processeur.

Que dire de plus

Fin du match !

Docker a gagné sur le Ring : "Déploiement d'environnement spécifique pour vos applications" mais n'enterrez pas les VMs trop vite, elles ont encore de beaux jours devant elles !



2. Installation de Docker

Version 1.0.0 | Dernière mise à jour : 17/11/2024 | Auteur : Bauer Baptiste

2.1. Vue d'ensemble

Nous avons maintenant une bonne idée de ce que sont Docker et les Conteneurs. Nous allons maintenant passer à l'installation de Docker sur notre machine.

Les étapes d'installation de Docker dépendent du système d'exploitation que vous utilisez.



Allez sur le site officiel de Docker pour télécharger la version de Docker qui correspond à votre système d'exploitation.

- En cliquant sur le lien suivant : <http://www.docker.com>

The screenshot shows the official Docker website. The top navigation bar includes links for Docs, Get support, Contact sales, Products, Developers, Pricing, Support, Blog, and Company. On the right side of the header are a search icon, a 'Sign In' button, and a 'Get Started' button. The main content area features a dark blue background with white text. It announces the general availability of Docker Build Cloud with the headline 'Docker Builds: Now Lightning Fast' and the subtext 'Announcing Docker Build Cloud general availability'. A prominent button labeled 'Discover Docker Build Cloud' is centered below the announcement. At the bottom, there's a large call-to-action button with the text 'Accelerate how you build,'. A decorative graphic of stacked shipping containers is visible at the bottom of the page.

Puis cliquez sur le menu "Développeurs" et sélectionnez "Documentation".

The screenshot shows the top navigation bar of the Docker website. The 'Developers' menu item is highlighted with a red box. A dropdown menu is open under 'Developers', containing the following items: 'Documentation' (highlighted with a red box), 'Getting Started', 'Resources', and 'Trainings'. The 'buildcloud' logo is visible on the right side of the header.

Ensuite, cliquez sur "Get Docker" :

The screenshot shows the 'Get Docker' section of the Docker website. It features a purple play button icon followed by the text 'Get Docker'. Below this, a subtext reads 'Learn how to install Docker for Mac, Windows, or Linux and explore our developer tools.' A red box highlights the 'Get Docker' button, and a red arrow points to it from the left.

Et sur la page, vous trouverez les instructions pour installer Docker sur votre système d'exploitation.

Comme ici, nous voyons le lien pour installer Docker sur Windows :

The screenshot shows the 'Get Docker' section of the Docker website with three options listed: 'Docker Desktop for Mac', 'Docker Desktop for Windows', and 'Docker Desktop for Linux'. Each option has a small icon and a brief description. The 'Docker Desktop for Windows' option is highlighted with a red box, and a red arrow points to it from the left.

Sur la page, vous trouverez les prérequis pour installer Docker sur Windows, Mac ou Linux.

Sur votre Système d'exploitation Windows ou Mac, vous pouvez installer **Docker Desktop** ou **Docker Toolbox**.

Docker Desktop et **Docker Toolbox** sont des outils qui donnent vie à Docker sur les systèmes d'exploitation Windows et Mac.

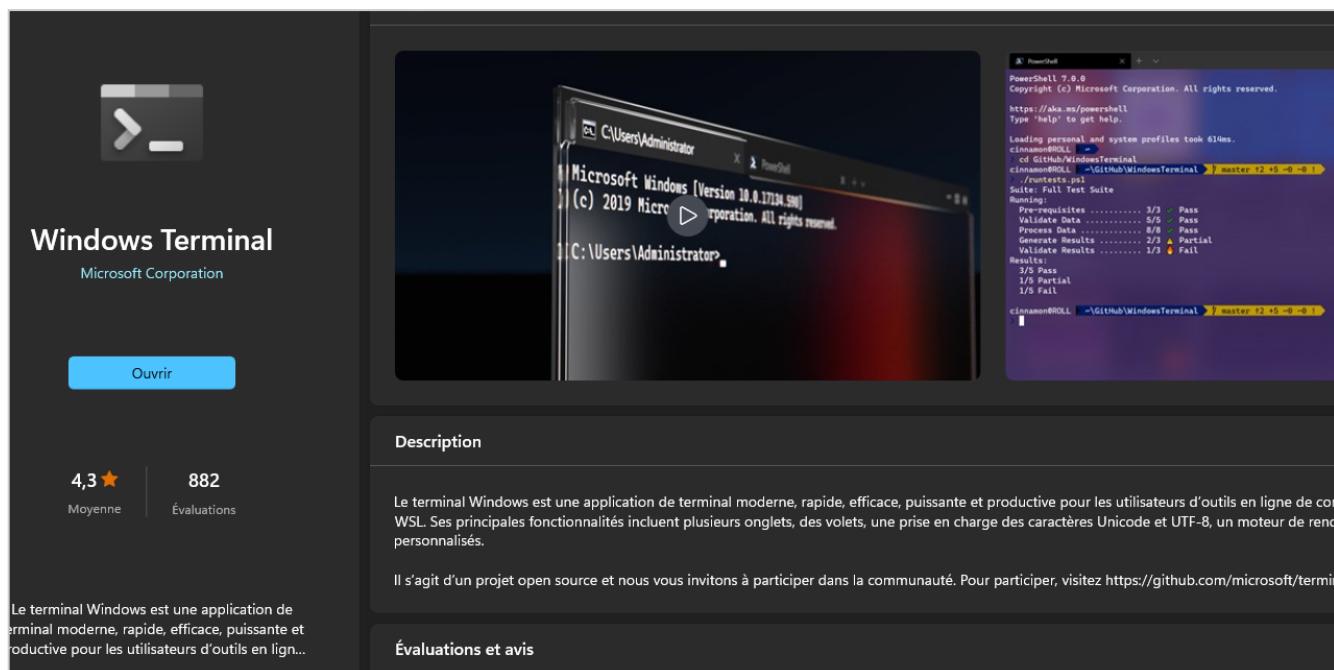
Si vous avez une machine sous Linux, vous pouvez installer **Docker Engine** directement sans avoir besoin de **Docker Desktop** ou **Docker Toolbox**. Car les **OS Linux** prennent en charge nativement les conteneurs et la technologie que Docker utilise.

2.2. Installation de Docker sur Windows

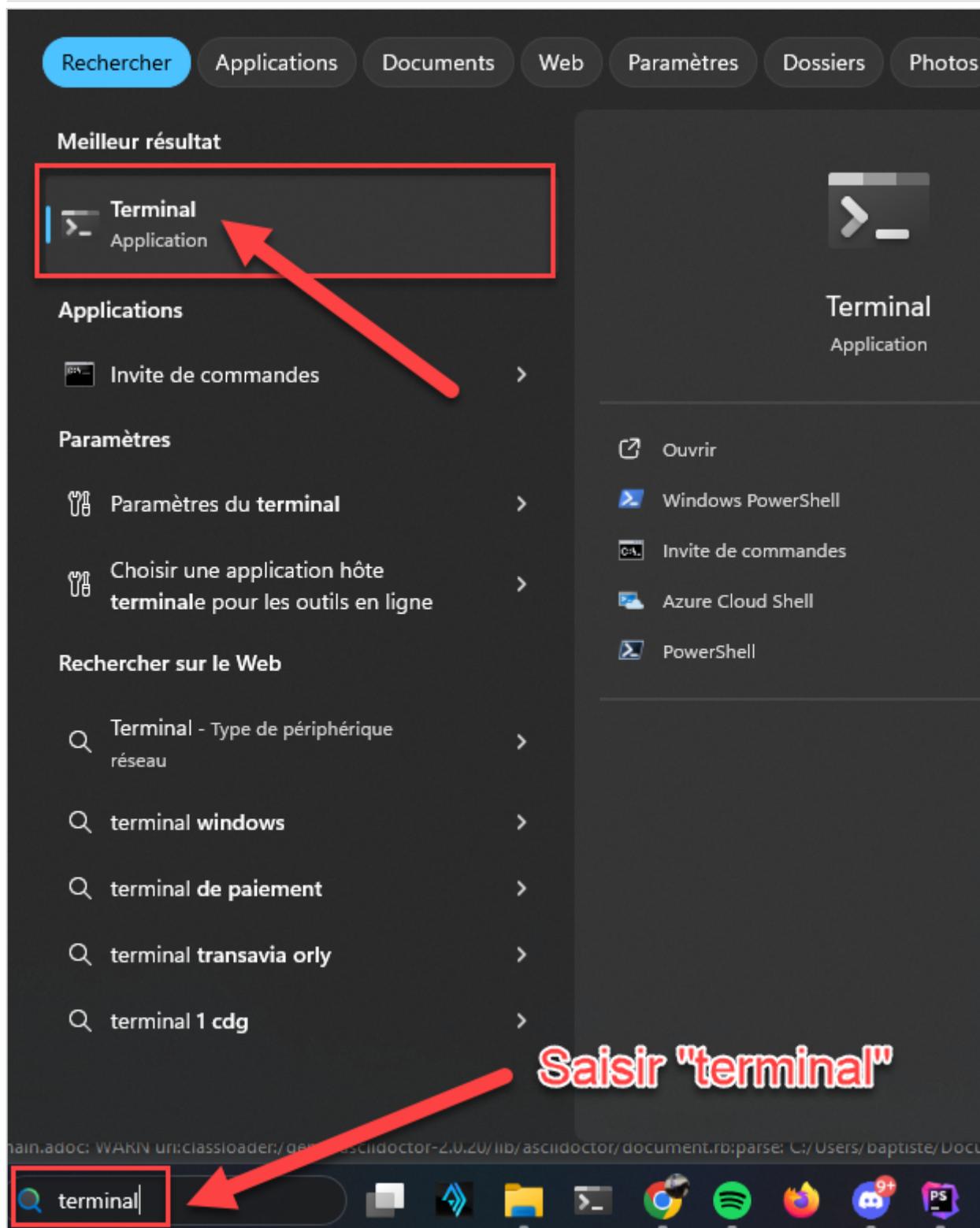
2.2.1. Le nouveau Terminal pour Windows

Durant tout ce cours, nous allons utiliser de nombreuses lignes de commande. Pour cela, nous vous recommandons d'utiliser le nouveau terminal de Windows. Qui est un outil très puissant, personnalisable, qui supporte les onglets, les thèmes, les raccourcis clavier, et bien plus encore.

Pour installer le nouveau terminal de Windows, vous pouvez le télécharger depuis le **Microsoft Store**.

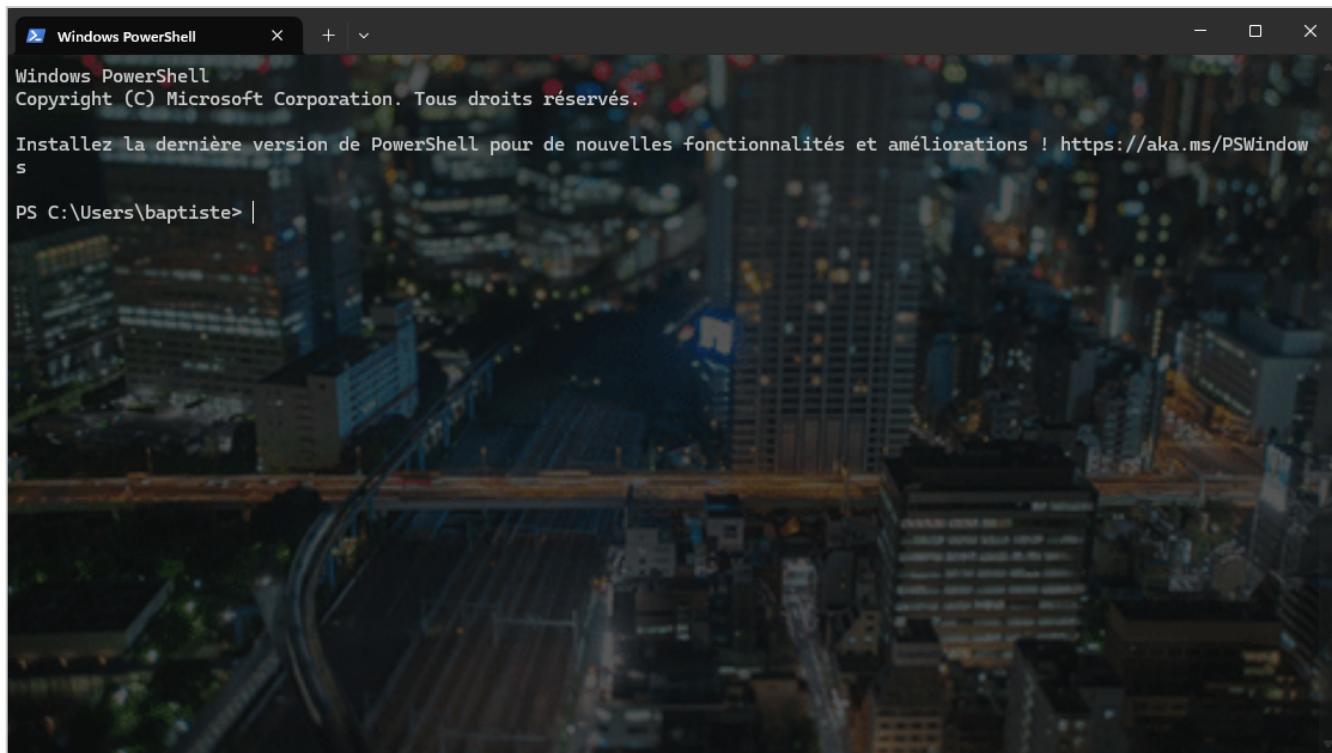


Une fois installé, vous pouvez l'ouvrir en tapant **Windows Terminal** dans la barre de recherche de Windows.



Vous aurez alors accès à un nouveau terminal moderne personnalisable comme le mien :





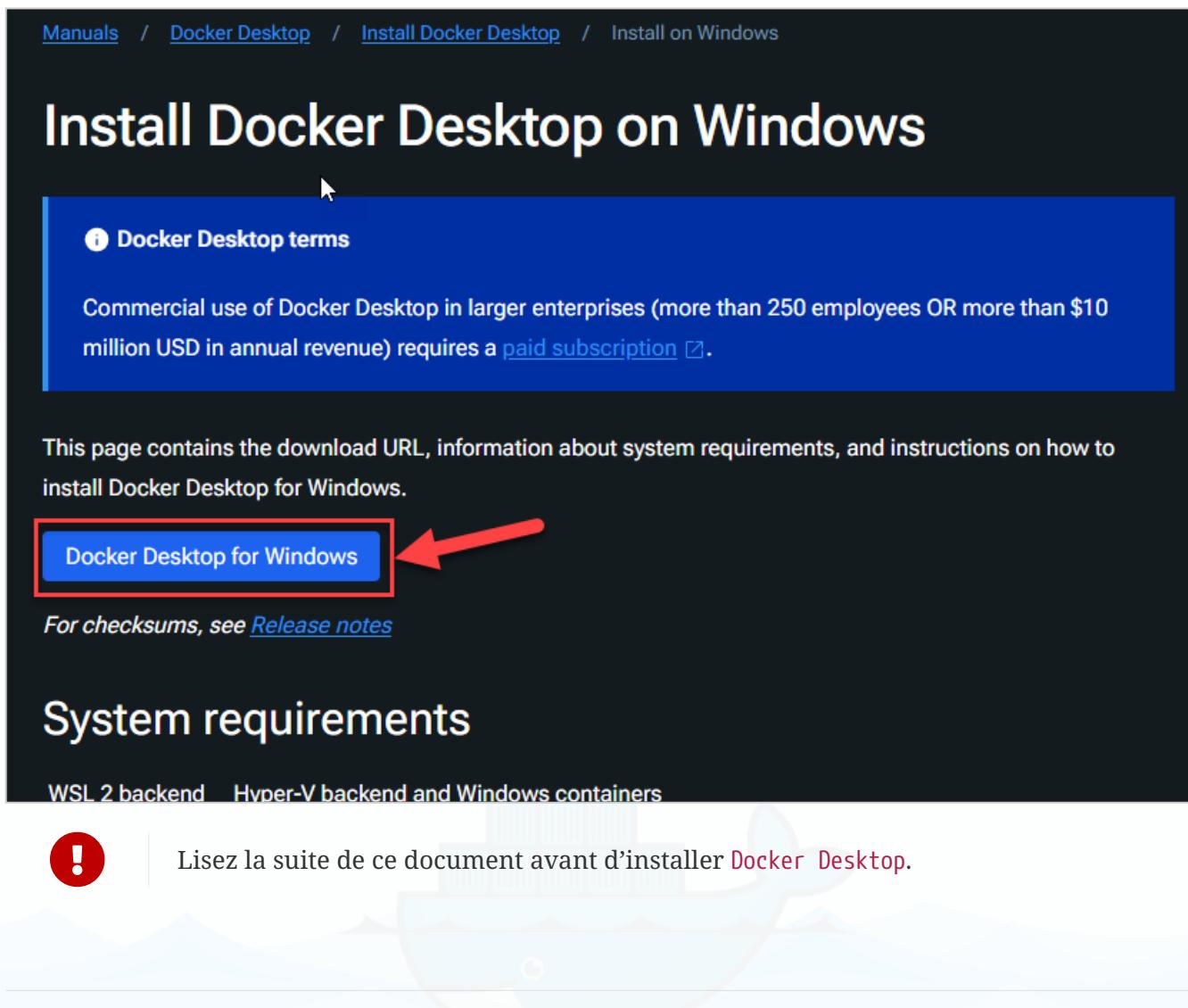
```
Windows PowerShell
Copyright (C) Microsoft Corporation. Tous droits réservés.

Installez la dernière version de PowerShell pour de nouvelles fonctionnalités et améliorations ! https://aka.ms/PSWindows

PS C:\Users\baptiste> |
```

2.2.2. Téléchargement

Téléchargez Docker Desktop pour Windows



Manuals / Docker Desktop / Install Docker Desktop / Install on Windows

Install Docker Desktop on Windows

Docker Desktop terms

Commercial use of Docker Desktop in larger enterprises (more than 250 employees OR more than \$10 million USD in annual revenue) requires a [paid subscription](#).

This page contains the download URL, information about system requirements, and instructions on how to install Docker Desktop for Windows.

Docker Desktop for Windows 

For checksums, see [Release notes](#)

System requirements

WSL 2 backend Hyper-V backend and Windows containers

! Lisez la suite de ce document avant d'installer Docker Desktop.

2.2.3. Virtualisation d'un noyau Linux

Comme nous l'avons vu, Docker utilise un noyau Linux pour fonctionner. Dans le cas de Windows, Docker Desktop utilise une machine virtuelle pour exécuter un noyau Linux. Il existe deux options pour exécuter Docker Desktop sur Windows :

- **WSL 2** (Windows Subsystem for Linux) : C'est une fonctionnalité de Windows 10 qui permet d'exécuter un noyau Linux sur Windows.
- **Hyper-V** : C'est une technologie de virtualisation de Windows.

Dois-je utiliser WSL 2 ou Hyper-V ?

Docker Desktop utilise **WSL 2** par défaut. Si vous avez déjà installé **WSL 2**, Docker Desktop l'utilisera. Toutefois, il est possible de basculer entre **WSL 2** et **Hyper-V**. Il n'y a pas de différences majeures entre les deux.

 **WSL 2** est recommandé, car il est plus rapide et plus léger qu'`Hyper-V`. Mais les deux possèdent des avantages et des inconvénients.

Par exemple, les conteneurs et les images créés avec **Docker Desktop** sont partagés entre tous les utilisateurs de la machine où **Docker Desktop** est installé. Cela est possible, parce que tous les comptes utilisateurs de la machine ont accès à la même machine virtuelle de **Docker Desktop**. Cependant, notez qu'il n'est pas possible de partager des **conteneurs** et des **images** entre plusieurs comptes utilisateurs quand vous utilisez **Docker Desktop** avec **WSL 2**.

2.2.3.1. Avec WSL

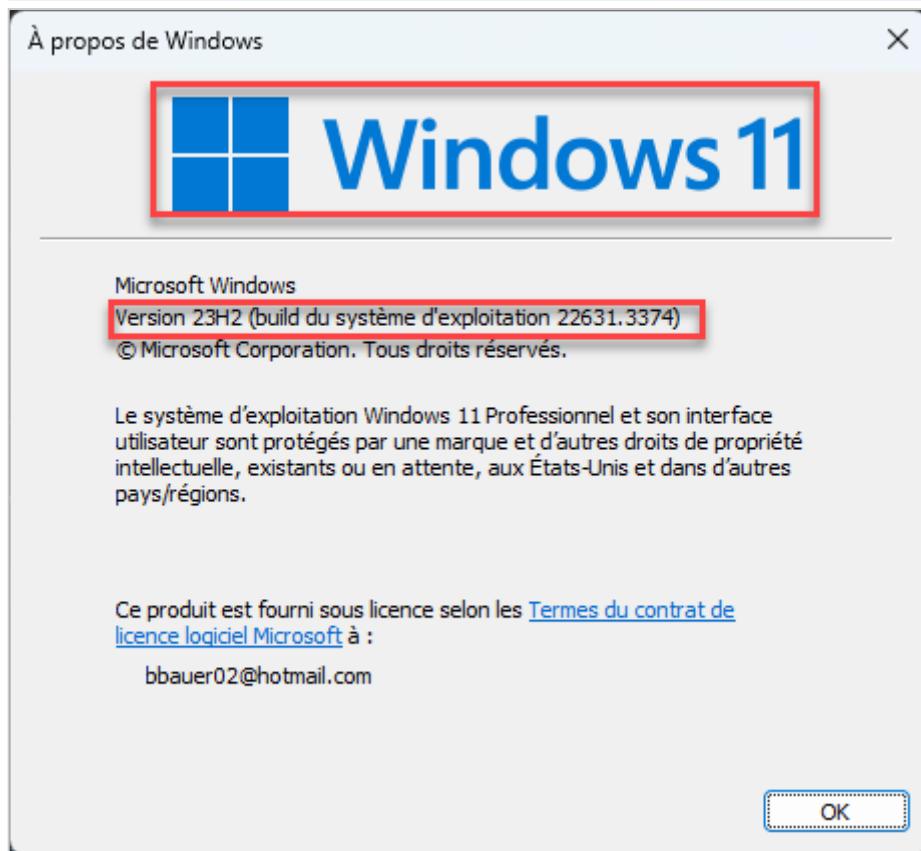
2.2.3.1.1. Prérequis : version du système d'exploitation

Il faut une machine avec :

- **Windows 11 64-bit:** Home ou Pro version 21H2 or higher, or Enterprise or Education version 21H2 or higher.
- **Windows 10 64-bit:**
 - Il est recommandé Home or Pro 22H2 (build 19045) or higher, or Enterprise or Education 22H2 (build 19045) or higher.

Si votre version de **Windows** n'est pas compatible, vous pouvez utiliser **Docker Toolbox**.

Pour vérifier la version de votre système d'exploitation, tapez **winver** dans la barre de recherche de Windows.



2.2.3.1.2. Prérequis : Configuration matérielle

Il faut que votre machine ait :

- Un processeur 64-bit avec prise en charge de la technologie SLAT (Second Level Address Translation).
- Au moins 4 Go de RAM.
- La virtualisation activée dans le BIOS/UEFI.

2.2.3.1.3. Prérequis : Activer WSL

WSL est une fonctionnalité de Windows qui permet de d'installer une distribution Linux sur Windows et d'utiliser des applications, des utilitaires et des outils de ligne de commande Bash directement sous Windows, sans modification et sans le surcoût d'une machine virtuelle traditionnelle ou d'une configuration en double démarrage.

Vérifiez si `wsl` est installé en tapant la commande suivante :

```
wsl --version
```

A screenshot of a Windows terminal window titled 'cmd'. The command `wsl --version` is entered and its output is displayed:
PS C:\Users\baptiste> wsl --version
Version WSL : 2.1.5.0
Version du noyau : 5.15.146.1-2
Version WSLg : 1.0.60

- Il faut que WSL soit à la version 1.1.3.0 ou supérieure pour pouvoir utiliser Docker Desktop.



Si vous obtenez une erreur, cela signifie que **WSL** n'est pas installé sur votre machine.

2.2.3.1.4. Mise à jour de WSL

Si WSL est déjà installé, vous pouvez le mettre à jour en tapant la commande suivante :

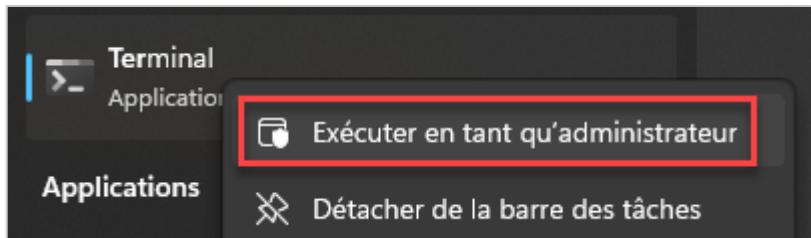
```
wsl --update
```

2.2.3.1.5. Installation

Vous pouvez désormais installer tout le nécessaire pour exécuter **WSL** avec une seule commande. Ouvrez **PowerShell** ou l'invite de commande Windows` en mode **administrateur**.

Ou bien, utilisez le nouveau terminal de Windows.

Pour cela, faites un clic droit sur l'application "**Terminal**" et sélectionnez "**Exécuter en tant qu'administrateur**". Ensuite, tapez la commande wsl --install et redémarrez votre machine. Par défaut, le nouveau terminal de Windows lance un **PowerShell**.



Saisissez la commande suivante :

```
wsl --install
```

Cette commande va installer les outils nécessaires pour lancer **WSL** sur votre machine ainsi qu'une distribution '**Ubuntu**' par défaut.

Vous pouvez changer de distribution en utilisant la commande suivante :

```
wsl --install -d <distribution>
```

En remplaçant **<distribution>** par le nom de la distribution que vous souhaitez installer.

Pour plus d'informations, vous pouvez consulter la documentation officielle de Microsoft sur WSL :

2.2.3.1.6. Vérification de la version de WSL

Sur Windows nous avons deux versions de **WSL** : WSL 1 et WSL 2. DOCKER Desktop a besoin de **WSL 2**.

Pour vérifier la version de **WSL** utilisée par votre machine, tapez la commande suivante :

```
wsl -l -v
```

Vous obtiendrez une liste des distributions installées sur votre machine avec leur version.

NAME	STATE	VERSION
* Ubuntu	Stopped	2
docker-desktop	Stopped	2
docker-desktop-data	Stopped	2

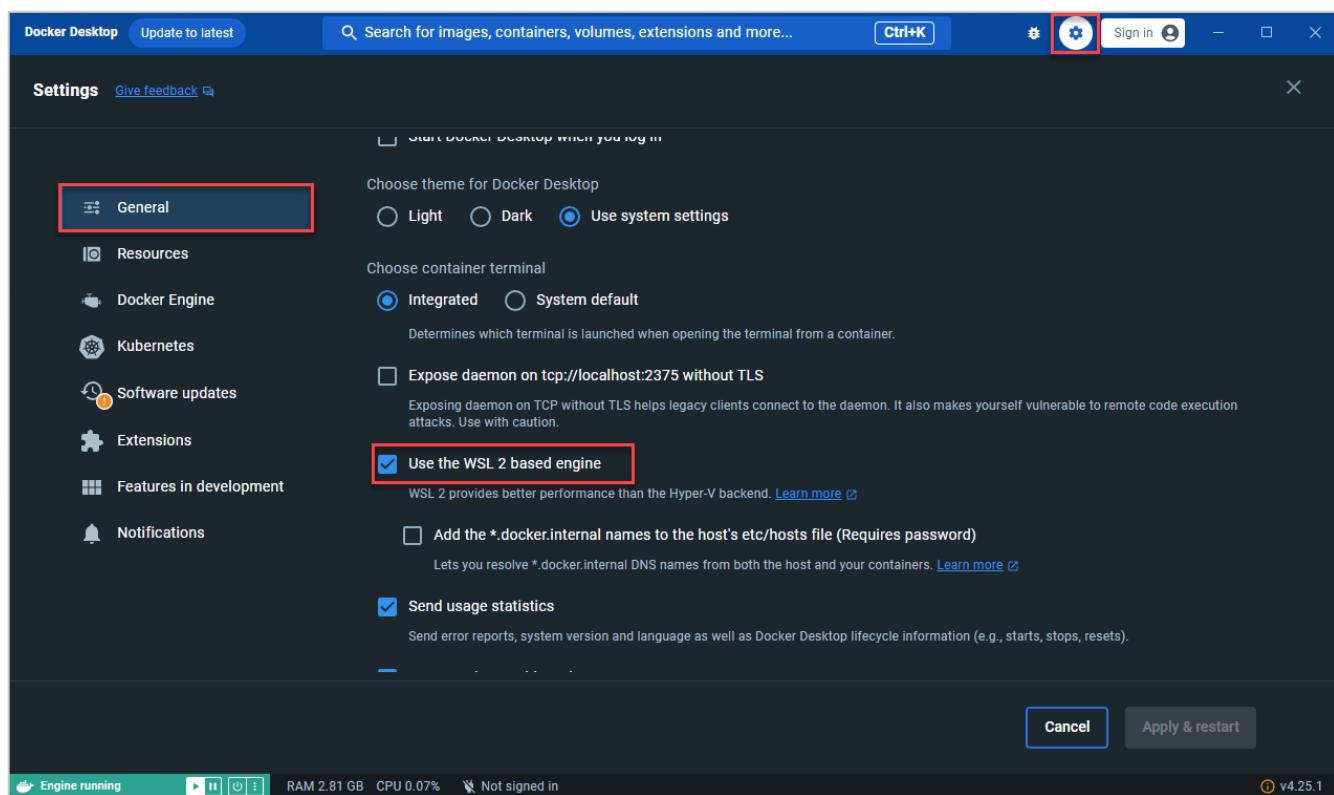
Tapez la commande suivante pour définir **WSL 2** comme version par défaut de **WSL** :

```
wsl --set-default-version 2
```

Il faut également vérifier que Docker Desktop est configuré pour utiliser **WSL 2** comme backend.

Ouvrez Docker Desktop et allez dans les paramètres en cliquant sur le petit engrenage en haut à droite de la fenêtre.

Dans l'onglet "**General**", assurez-vous que "**Use the WSL 2 based engine**" est coché.



Votre machine est maintenant prête à utiliser Docker Desktop !

2.2.3.2. Avec Hyper-V

2.2.4. Installation de Docker Desktop

Après s'être assuré d'avoir tous les prérequis nécessaires, vous pouvez installer [Docker Desktop](#) en double-cliquant sur le fichier d'installation pour lancer l'installation.

3. Les images Docker

Version 1.0.0 | Dernière mise à jour : 17/11/2024 | Auteur : Bauer Baptiste

3.1. Introduction aux images Docker

Dans ce chapitre, nous allons traiter deux concepts fondamentaux de Docker : les images et les conteneurs. Il est essentiel de connaître et de comprendre ces deux concepts pour utiliser Docker de manière efficace.

Dans la présentation de Docker, nous avons brièvement évoqué le concept de conteneurs. À présent, nous allons explorer la notion d'image Docker pour saisir le lien qui les unit aux conteneurs.

Nous allons apprendre comment utiliser des images existantes, ainsi que créer nos propres images personnalisées.

Plongeons maintenant dans ce nouveau concept pour travailler pleinement avec Docker.

3.2. Images et Conteneurs : Quelle différence ?

Comme mentionné précédemment, lorsque nous utilisons **Docker**, nous ne disposons pas seulement de conteneurs, mais aussi d'images.

Quelle est la différence entre ces deux concepts et pourquoi avons-nous besoin des deux ?

Nous savons que les conteneurs, en fin de compte, sont de petits paquets qui contiennent tout ce dont nous avons besoin pour exécuter une application : l'application elle-même, ses dépendances, ses bibliothèques, ses variables d'environnement, ses serveurs, etc. En d'autres termes, c'est l'environnement complet nécessaire pour exécuter l'application.



Un conteneur est donc un **processus**, car c'est finalement ce que nous exécutons sur notre machine.

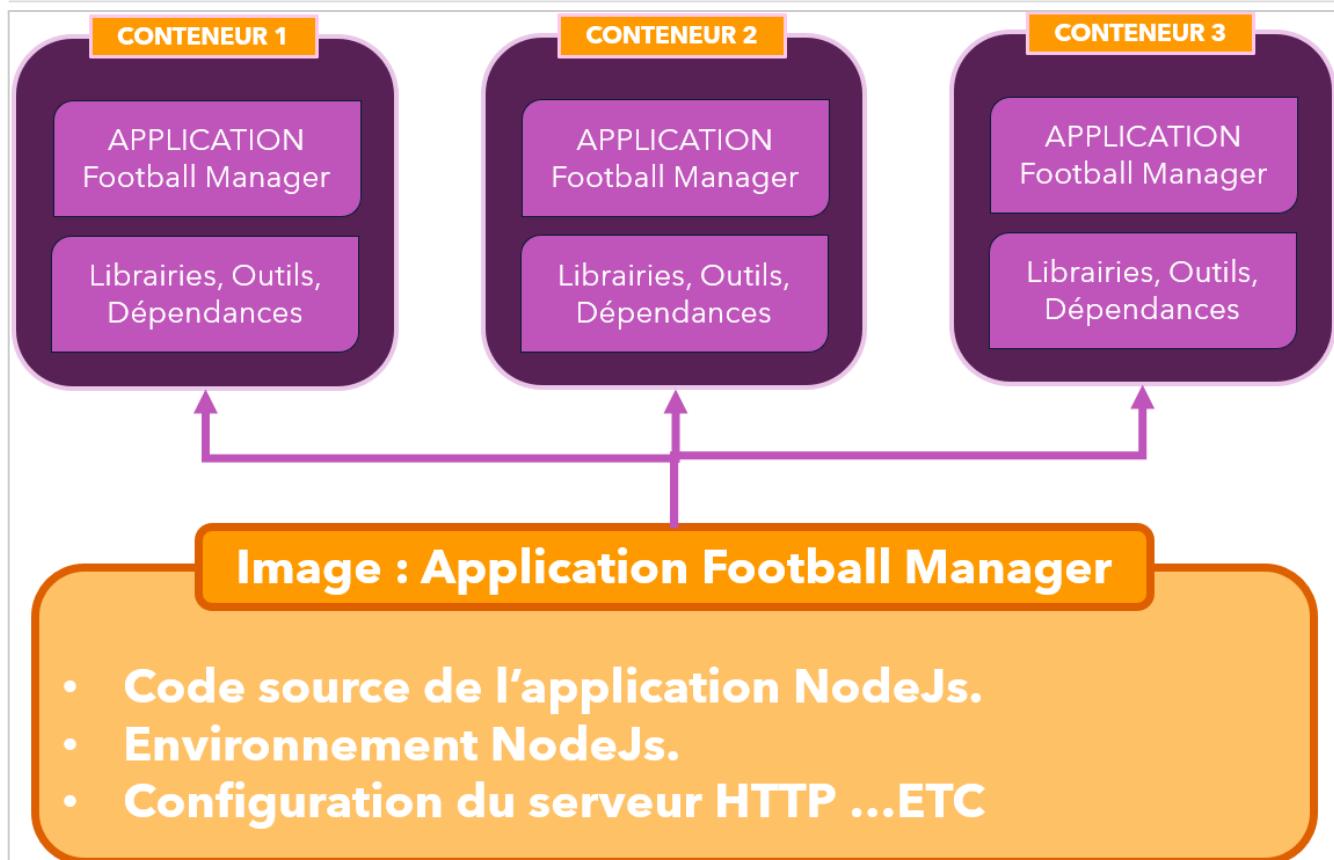
D'autre part, une image est un fichier contenant tout ce dont nous avons besoin pour créer un conteneur. Une image est un modèle, une sorte de gabarit qui nous permet de créer un conteneur. Elle contient le code source et les outils nécessaires pour exécuter une application.

Le rôle du conteneur est de lancer et d'exécuter l'application.



À partir d'une seule image, nous pouvons créer plusieurs **conteneurs** qui exécutent la même application dans le même environnement.

Prenons l'exemple d'une application web écrite en **Node.js**. Nous la définissons une seule fois dans une **image**, puis nous pouvons exécuter cette application plusieurs fois dans des conteneurs différents, sur différentes machines, sur différents serveurs.

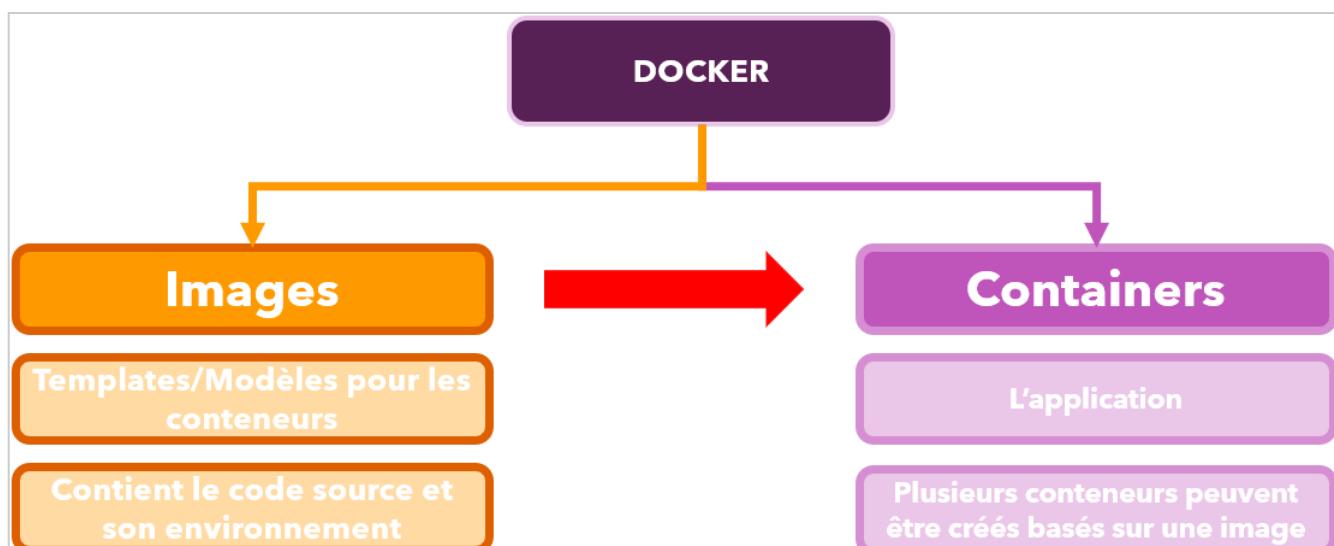


Cette image est un package partageable contenant toutes les instructions d'installation et de configuration de l'application. Le conteneur est une instance de cette image qui exécute l'ensemble des instructions.



Nous lançons des conteneurs qui sont basés sur des images. **C'est là le concept fondamental de Docker.**

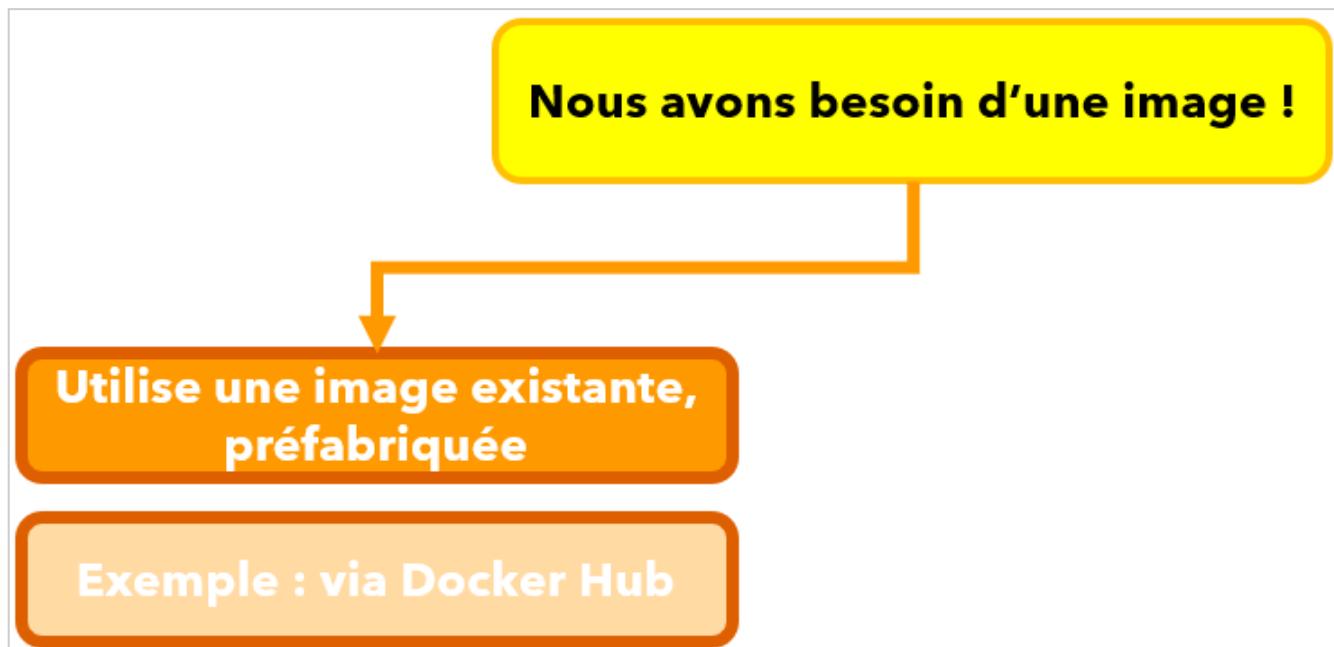
Cela deviendra encore plus clair lorsque nous commencerons à manipuler les images et les conteneurs.



3.3. Les images Docker pré-construites

3.3.1. Préambule

Il y a deux façons de créer ou d'obtenir des images Docker : Nous en étudierons une dans ce sous-titre, et l'autre dans le sous-titre suivant.



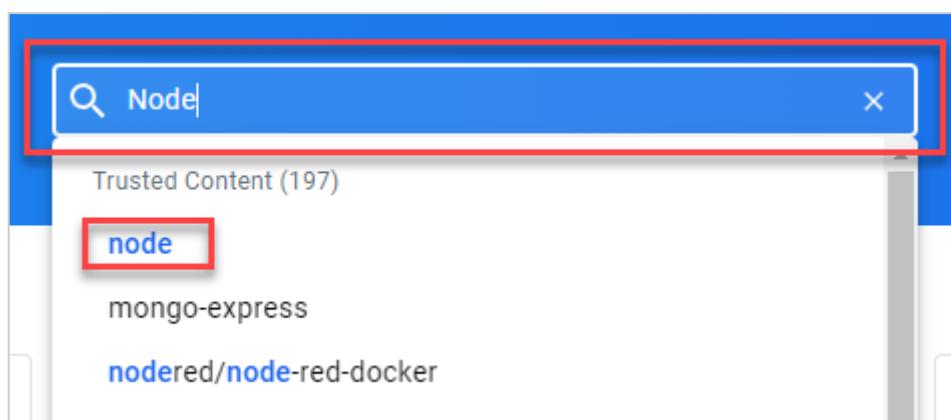
Utiliser des conteneurs existants : Ceux-ci sont créés par la communauté, nos collègues ou officiellement par les éditeurs de logiciels.

Il existe un grand nombre d'images Docker disponibles sur le Docker Hub, le registre public de Docker. Vous pouvez y trouver des images pour des applications populaires telles que MySQL, Redis, Node.js, Python, etc.

[Lien vers le Docker Hub](#)

Vous n'avez pas besoin de vous enregistrer ou de vous authentifier sur le site pour accéder aux images.

Par exemple, dans la barre de recherche, vous pouvez effectuer une recherche pour trouver l'image officielle de Node.js qui pourra être utilisée pour construire un conteneur avec Node.js.



Nous utiliserons beaucoup d'images officielles dans ce cours, mais aussi généralement dans notre travail quotidien avec Docker.

Voici la réponse du moteur de recherche de Docker Hub :

The screenshot shows the Docker Hub search results for the 'node' image. At the top, there's a search bar with the query 'docker pull node'. Below it, a red arrow points to the 'Copy' button next to the search term, with the text 'Nom de l'image' (Image name) written above it. In the main search results table, there's a row for the tag 'lts-alpine3.19'. A red arrow points to the 'Copy' button next to this tag, with the text 'Tag de l'image' (Image tag) written below it. The table also includes columns for Digest, OS/ARCH, Vulnerabilities, Compressed Size, and several image IDs.

TAG	Digest	OS/ARCH	Vulnerabilities	Compressed Size
lts-alpine3.19	73753e08a875	linux/amd64	None found	45.73 MB
2a984bb09202	linux/arm/v6	None found		44.17 MB
84801715e91d	linux/arm/v7	None found		43.41 MB

Pour installer l'image, on vous donne une commande :

```
docker pull node
```

Cette commande télécharge l'image sur votre machine hôte.

Vous pouvez sélectionner une image particulière contenant une version spécifique du service ou du système Linux de base. Pour ce faire, consultez l'onglet **Tag** et récupérez le nom de tag correspondant à la version souhaitée.

Par exemple, si nous voulons utiliser **Node.js** basé sur une distribution **Alpine 3.19** :

```
docker pull node:lts-alpine3.19
```

3.3.2. Utiliser une image existante

Assurez-vous d'avoir Docker Engine de démarré sur votre machine. (Ouvrez l'application Docker Desktop et vérifiez que le statut est "Engine Running" ou "Ressource Saver mode")



Ouvrez un terminal sur votre machine hôte **et** exécutez la commande suivante pour télécharger l'image officielle de NodeJs et monter un conteneur :

```
docker container run node
```



Les commandes `docker container run` et `docker run` ont le même effet. Cependant,

depuis la **version 1.13** de Docker, il est recommandé d'utiliser `docker container run`.

En effet, l'ensemble des commandes et sous-commandes ont été réorganisées pour suivre une structure de ce type : `docker <objet> <commande> <options>`.

Cette nouvelle structure offre une meilleure lisibilité de la commande et permet de connaître son champ d'action.

Ainsi, en tapant une commande de ce type : `docker container <commande>`, nous savons que nous allons manipuler des conteneurs.

Tandis que `docker image <commande>` concerne la manipulation d'une image.

Si vous n'avez pas exécuté la commande `docker pull node` précédemment, vous verrez alors apparaître une erreur signalant que l'image `node` n'a pas pu être trouvé localement. Ainsi, elle sera automatiquement téléchargé depuis le terminal sur les serveurs du **Docker Hub**.

PS C:\Users\baptiste> docker run node
Unable to find image 'node:latest' locally
Latest: Pulling from library/node
71215d55680c: Downloading [=====] 14.2MB/49.55MB
3cb8f9c23302: Downloading [=====] 3.435MB/24.05MB
5f899db30843: Downloading [==>] 2.681MB/64.14MB
567db630df8d: Waiting
f4ac4e9f5ffb: Waiting
375735fcfaa7a: Waiting
c12db77023cd: Waiting
ac50344c1606: Waiting

Message d'erreur indiquant que l'image "node" n'a pas été trouvé localement.
Téléchargement de l'image

Maintenant que l'image est présente sur notre machine, la même commande procède à la création d'un conteneur basé sur cette image et le lance. Cependant, sur le terminal, l'action semble se terminer sans que rien se produise.

PS C:\Users\baptiste> docker run node
Unable to find image 'node:latest' locally
Latest: Pulling from library/node
71215d55680c: Pull complete
3cb8f9c23302: Pull complete
5f899db30843: Pull complete
567db630df8d: Pull complete
f4ac4e9f5ffb: Pull complete
375735fcfaa7a: Pull complete
c12db77023cd: Pull complete
ac50344c1606: Pull complete
Digest: sha256:b9ccc4aca32eebf124e0ca0fd573dacffba2b9236987a1d4d2625ce3c162ecc8
Status: Downloaded newer image for node:latest
PS C:\Users\baptiste> |
Installation terminée mais il ne se passe rien ?

Pourquoi ?



Il est essentiel de noter que par **défaut**, un conteneur est isolé de son environnement immédiat. Cependant, il est possible d'interagir avec lui via un

shell interactif. Lorsqu'il est créé, le conteneur est initialement lancé **en mode détaché**, ce qui signifie qu'il s'exécute en arrière-plan sans offrir de terminal pour interagir directement avec lui.

Donc, même si sur le terminal, il ne semble rien se passer, le conteneur a bien été créé.

Vérifions cela en exécutant la commande suivante :

```
docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
3aaae58260be	node	"docker-entrypoint.s..."	4 minutes ago	Exited (0) 4 minutes ago		objective_archimedes

Lors de la création du conteneur, nous voyons qu'il a reçu un identifiant unique : **CONTAINER ID**, et un nom en caractère alphanumérique généré aléatoirement : **NAMES**. Nous verrons plus en détail comment configurer le conteneur un peu plus tard.

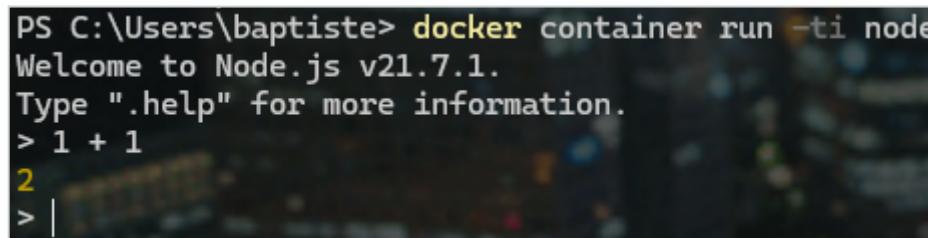
Attardons-nous sur le **STATUS** qui est **Exited**. Qui signifie que le conteneur a bien démarré une fois, puis s'est éteint.

Cela est normal !

Actuellement, le conteneur n'effectue aucune tâche particulière ; il s'agit simplement d'un environnement dans lequel Node.js est installé. Par défaut, le shell interactif ne nous est pas accessible. Ainsi, plutôt que de rester en fonctionnement sans rien faire, le conteneur s'arrête automatiquement.

Pour modifier ce comportement, nous pouvons créer un nouveau conteneur en utilisant la même commande, mais en ajoutant un nouveau paramètre : **-ti**. Ce paramètre indique que nous souhaitons interagir avec le conteneur.

```
docker container run -ti node
```



Nous remarquons que la création du nouveau conteneur n'a pas entraîné le téléchargement de l'image ! (Elle est sur notre machine hôte en local maintenant !).

Et nous avons en plus, un prompt dans lequel nous pouvons saisir des commandes **NodeJs** ou **Javascript** qui s'exécuteront seulement à l'intérieur de notre conteneur **et pas dans notre machine hôte, soyons bien claire avec cela !**

Pour sortir du shell, tapez sur la combinaison de touche du clavier : **CTRL + C** deux fois.

Et listons les conteneurs qui ont été créés :

```
docker container ps -a
```

ou

```
docker ps -a
```

Il existe maintenant plusieurs conteneurs Docker, basés sur la même image **Node**, indépendant les uns des autres.

Nous avons pris l'exemple de l'image **Node** pour illustrer le concept de conteneurs et d'images Docker. Cependant, tout ce que nous avons appris reste valable quelque soit votre environnement technique : PHP, PYTHON, RUBY, Etc.



En règle générale, vous utiliserez à chaque fois une image de base officielle pour créer un conteneur. Puis, vous personnaliserez cette image en ajoutant vos propres fichiers, dépendances, etc.

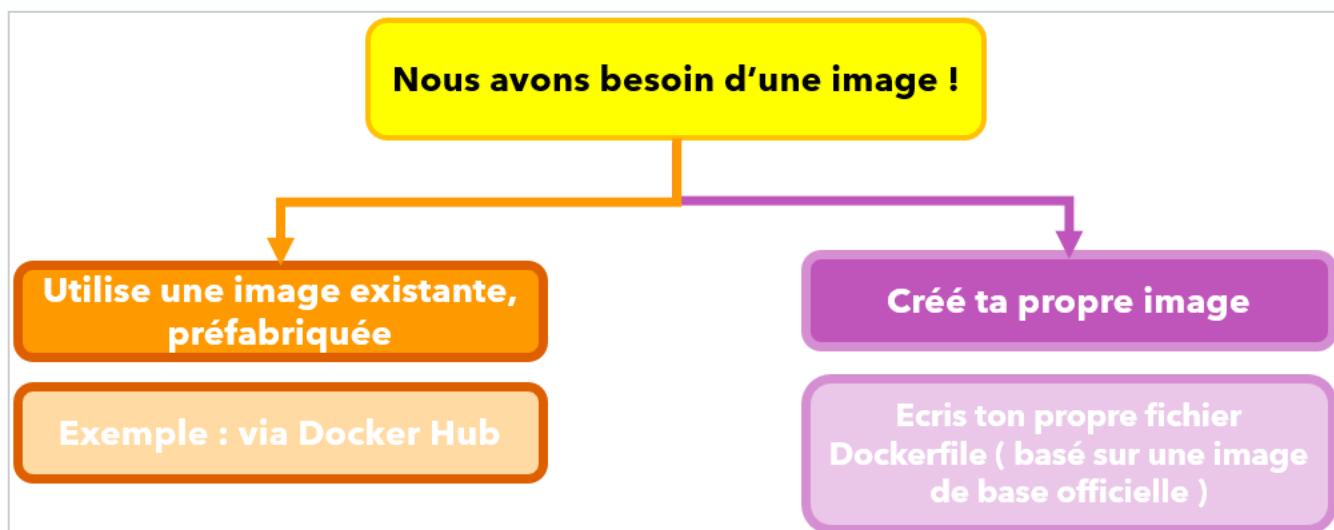
Nous verrons comment créer des images personnalisées dans le chapitre suivant.

3.4. Les images Docker personnalisée

3.4.1. Préambule

Nous avons appris comment utiliser une image **Docker** existante pour créer un conteneur. Cependant, il est souvent nécessaire de créer des images personnalisées pour répondre aux besoins spécifiques de nos applications. Par exemple, si nous souhaitons déployer une application **Node.js**, nous allons utiliser une image de base **Node.js**, puis y ajouter nos fichiers et dépendances.

Dans ce sous-titre, nous allons voir comment procéder en suivant un exemple concret.



3.4.2. TD : Créer une image pour une application Node.js

3.4.2.1. Objectif

Créer une image Docker personnalisée à partir d'une image NodeJs officielle et d'y ajouter le code source d'une application Node.js existante.

3.4.2.2. Préparation

Pour réaliser ce TD, il n'est pas nécessaire de connaître NodeJS, ni Javascript.



Nous allons simplement utiliser Node.js pour illustrer la création d'une image Docker personnalisée.

Nous vous fournirons les fichiers nécessaires pour réaliser ce TD.

- Récupérez le fichier `td01_app_nodejs.zip` dans le répertoire `resources` de ce chapitre.
- Décompressez le fichier dans un répertoire de votre choix.

3.4.2.3. Présentation des fichiers

Le répertoire décompressé contient :

- Un répertoire `public` contenant :
 - Un fichier `styles.css` : une feuille de style simple.
- Un fichier `package.json` : un fichier de configuration pour Node.js.
- Un fichier `server.js` : un fichier Javascript qui crée un serveur web simple.

Le fichier `server.js` contient le code de notre application Node.js. Si vous connaissez Node.js, vous pouvez l'ouvrir pour voir son contenu. Sinon, ne vous inquiétez pas, nous n'aurons pas besoin de le modifier.

Toutefois, voici quelques explications sur ce fichier :

```
1 // Création d'un serveur HTTP avec Express en NodeJS
2 const express = require('express');
3 const bodyParser = require('body-parser');
4 const app = express();
5
6 // Le serveur HTTP démarre et écoute sur le port 80
7 app.listen(80);
```

Le code ci-dessus crée un serveur web simple qui écoute sur le port 80 et nous gérons les requêtes HTTP entrantes (méthodes `GET` et `POST`) pour deux URL différentes : `/` et `/store-goal` :

server.js : Code du traitement de la requête HTTP GET

```
// [...Some Code before]
```

```
app.get('/', (req, res) => {
  res.send(`<html>
    <head>
      <link rel="stylesheet" href="styles.css">
    </head>
    <body>
      <section>
        <h2>Objectif : </h2>
        <h3>${userGoal}</h3>
      </section>
      <form action="/store-goal" method="POST">
        <div class="form-control">
          <label>Course Goal</label>
          <input type="text" name="goal">
        </div>
        <button>Ajouter un objectif</button>
      </form>
    </body>
  </html>
`);
});

// [...Some Code after]
```

Le code ci-dessus gère la requête HTTP GET pour l'URL `/`. Il renvoie une page HTML contenant un formulaire pour saisir un objectif et affiche aussi l'objectif saisi précédemment ou un objectif par défaut.

server.js : Code du traitement de la requête HTTP POST

```
// [...Some Code before]
app.post('/store-goal', (req, res) => {
  const enteredGoal = req.body.goal;
  console.log(enteredGoal);
  userGoal = enteredGoal;
  res.redirect('/');
});

// [...Some Code after]
```

Le code ci-dessus gère la requête HTTP POST pour l'URL `/store-goal`. Il récupère l'objectif saisi dans le formulaire, le stocke dans une variable `userGoal`, puis redirige l'utilisateur vers la page d'accueil.

Le fichier `package.json` central pour les applications Node.js, car il contient toutes les informations nécessaires pour installer les dépendances de l'application.

Extrait du fichier package.json

```
// [...Some Code before]
"dependencies": {
  "express": "^4.17.1",
```

```
"body-parser": "1.19.0"  
}  
// [...Some Code after]
```

Le fichier `package.json` nous montre que cette application nécessite la présence de deux dépendances pour fonctionner : `express` et `body-parser`.



Je ne donnerais pas plus d'explication, ce cours ne traite pas de NodeJs, mais de comment "**Dockeriser**"/"**Conteneuriser**" cette application d'exemple.

3.4.2.4. Lancez l'application Node.js en Local

Avant de créer l'image Docker, nous allons lancer l'application Node.js en local pour vérifier qu'elle fonctionne correctement et surtout pour comprendre son fonctionnement.

Pour exécuter une application Node.js, il faut d'abord installer Node.js sur votre machine.

Pour vérifier si Node.js est installé sur votre machine, ouvrez un terminal et tapez la commande suivante :

```
node -v
```

Si Node.js est installé, vous verrez sa version s'afficher. Sinon, vous devrez l'installer.

Rendez-vous sur le lien suivant pour télécharger et installer Node.js sur votre machine :

[Télécharger Node.js](#)

Sélectionnez la version adaptée à votre système d'exploitation.

Une fois Node.js installé, ouvrez **un nouveau terminal** et déplacez-vous dans le répertoire où vous avez décompressé les fichiers de l'application **Node.js**.

Installez les dépendances de l'application en exécutant la commande suivante :

```
npm install
```

```
PS C:\Users\baptiste\Documents\Cours\SPOT_COURS\chapters\docker\images_docker\code\td01_app_nodejs> npm install  
added 81 packages, and audited 82 packages in 2s  
12 packages are looking for funding  
  run 'npm fund' for details  
2 high severity vulnerabilities  
To address all issues, run:  
  npm audit fix --force  
Run 'npm audit' for details.
```

Répertoire contenant les fichiers et dossiers de l'application NodeJS

Un nouveau dossier `node_modules` est créé dans le répertoire de l'application. Il contient toutes les dépendances nécessaires pour exécuter l'application.

Nom	Modifié le	Type	Taille
📁 public	02/04/2024 00:17	Dossier de fichiers	
📄 package.json	02/04/2024 00:15	Fichier source JSON	1 Ko
📄 server.js	02/04/2024 01:24	Fichier source Jav...	2 Ko
📁 node_modules	02/04/2024 01:24		
📄 package-lock.json	02/04/2024 01:50	Fichier source JSON	32 Ko

Nouveau répertoire contenant les dépendances

Vous pouvez maintenant lancer l'application en exécutant la commande suivante :

```
node server.js
```

Laissez le terminal ouvert et ouvrez un navigateur web.

Tapez l'URL <http://localhost> dans la barre d'adresse pour accéder à l'application.

Objectif :

Apprendre Docker!

Objectif de ce cours :

Ajouter un objectif

Testez l'application en saisissant un objectif dans le champ de texte et en cliquant sur le bouton **Ajouter un objectif** et observez le résultat.

Cette application fonctionne donc localement sans Docker !

Maintenant, arrêtons le server Node.js en tapant **CTRL + C** dans le terminal. Et supprimons le dossier **node_modules** en tapant la commande suivante :

Sous Windows

```
rm node_modules
```

Sous Linux

```
sudo rm -R node_modules
```

et supprimons le fichier **package-lock.json** :

```
rm package-lock.json
```

Nous allons maintenant créer une image Docker spécialement pour cette application !

3.4.2.5. Dockerfile : Création de notre propre image Docker

Pour créer une image Docker personnalisée, nous devons créer un fichier appelé **Dockerfile** à la racine de notre application. C'est un nom spécial qui sera identifié par Docker.

Le **Dockerfile** contient les instructions pour construire une image Docker personnalisée.

Pour plus de confort dans la rédaction du **Dockerfile**, je vous encourage à installer une extension pour vous aider à écrire du code Docker. Par exemple, l'extension **Docker** pour **Visual Studio Code** ou **PHPSTORM**.

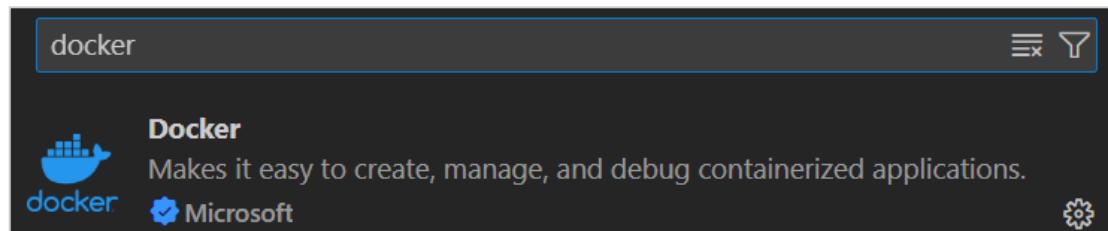


Figure 1. L'extension Docker pour Visual Studio Code

Ou

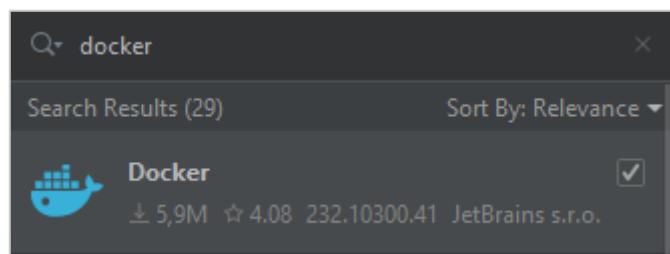


Figure 2. L'extension Docker pour PhpStorm

Nous commençons par le mot clé **FROM** suivi de l'image de base que nous voulons utiliser pour notre image personnalisée. Nous sommes obligé de spécifier une image de base, car nous ne pouvons pas créer une image à partir de rien.

Dans notre cas, nous allons utiliser l'image officielle de **Node.js** dans sa dernière version grâce au tag de version **latest**.

3.4.2.5.1. Instruction : FROM

Dockerfile

```
FROM node:latest
```

Maintenant, nous voulons copier les fichiers de notre application dans l'image Docker. Pour cela,

nous utilisons l'instruction **COPY** suivi du chemin du fichier ou du répertoire à copier dans l'image, puis du chemin de destination dans l'image.



Gardez à l'esprit qu'un conteneur (et donc également son image) contient l'environnement + le code de l'application. Votre code source soit alors être copié dans l'image.

3.4.2.5.2. Instruction : **COPY**

Dockerfile

```
FROM node:latest  
COPY . /app
```

Le point `.` signifie que nous copions tous les fichiers, les répertoires et sous-répertoires du répertoire courant (celui où se trouve le **Dockerfile**) vers le répertoire de destination `/app` dans l'image.

En effet, les conteneurs possèdent leur propre système de fichiers, totalement isolé de celui de la machine hôte.

Si vous utilisez une machine Windows, il est important de noter que le système de fichiers Linux est différent. Sous Linux, le répertoire **racine** est nommé `/`, l'équivalent de `C:\` sous **Windows**.

Par conséquent, `/app` signifie que nous aurons un dossier `app` à la racine de notre conteneur. Ce dossier sera automatiquement créé s'il n'existe pas.

3.4.2.5.3. Instruction : **RUN**

Dockerfile

```
FROM node:latest  
COPY . /app  
RUN npm install
```

L'instruction **RUN** permet d'exécuter des commandes dans l'image Docker. Rappelez-vous, pour tester notre application, nous avons lancé la commande `npm install` pour installer les dépendances de l'application. Nous devons donc exécuter cette commande dans l'image Docker pour installer les dépendances dedans.

Toutefois, il y a un piège, car l'instruction RUN sera exécutée dans le répertoire de travail de l'image et du conteneur, qui est le répertoire `root (/)` par défaut.

3.4.2.5.4. Instruction : **WORKDIR**

Comme nous avons copié les fichiers de notre application dans le répertoire `/app`, nous devons d'abord nous déplacer dans ce répertoire avant d'exécuter la commande `npm install`.

Pour cela, nous utilisons l'instruction **WORKDIR** pour définir le répertoire de travail de l'image par défaut.

Dockerfile

```
FROM node:latest
WORKDIR /app
COPY . /app
RUN npm install
```

Cette instruction `WORKDIR` indique à Docker que toutes les commandes suivantes seront exécutées à partir du répertoire `/app`.

Maintenant que le répertoire de travail est défini, nous pouvons modifier l'instruction `COPY` et changer la définition du répertoire de destination `/app` par `.` ou `./`.

Ce deuxième point symbolisant le répertoire de travail défini par l'instruction `WORKDIR`.

Dockerfile

```
FROM node:latest
WORKDIR /app
COPY . .
RUN npm install
```

Toutefois, pour garantir une certaine lisibilité du `Dockerfile`, il est préférable de spécifier littéralement le répertoire de destination `/app` avec la commande `COPY`. Cela évite de partir à la recherche du `WORKDIR` dans le cas de fichier `Dockerfile` volumineux.

Dockerfile

```
FROM node:latest
WORKDIR /app
COPY . /app
RUN npm install
```

3.4.2.5.5. Instruction : CMD

Désormais, nous désirons lancer notre serveur Node.js. Pour ce faire, nous devons exécuter la commande `node server.js`.

Il pourrait être tentant d'utiliser l'instruction `RUN` pour exécuter cette commande, mais cela ne serait pas efficace.

En effet, l'instruction `RUN` est exécutée lors de la construction de l'image, et non lors du démarrage du conteneur.

Il est important de se rappeler que l'image Docker est un modèle, un gabarit pour créer des conteneurs. Elle ne peut pas exécuter de commandes. Nous pouvons y copier des fichiers, lancer des commandes qui effectuent des installations. Cependant, nous ne pouvons pas exécuter des commandes qui nécessitent une interaction continue comme le démarrage d'un serveur.

Ainsi, si nous démarrons plusieurs conteneurs sur la même image, nous démarrons également plusieurs serveurs Node.js.

Pour résoudre ce problème, nous devons utiliser une autre instruction : **CMD**.

Dockerfile

```
FROM node:latest
WORKDIR /app
COPY . /app
RUN npm install
CMD ["node", "server.js"]
```

Instruction **CMD** permet de définir la commande par défaut qui sera exécutée lors du démarrage du conteneur. La syntaxe est un tableau JSON, où chaque élément du tableau est un argument de la commande.

Dans notre cas, nous exécutons la commande `node server.js` pour démarrer notre serveur Node.js. Nous insérons donc chaque élément qui constitue la commande dans le tableau : `["node", "server.js"]`.

! Si vous ne spécifiez pas d'instruction **CMD** dans le **Dockerfile**, Docker utilisera l'instruction **CMD** de l'image de base. Sans image de base et sans instruction **CMD**, vous générerez une erreur lors de la création du conteneur.

3.4.2.5.6. Instruction : EXPOSE

Comme nous l'avons maintes fois répété, les conteneurs Docker sont isolés de notre environnement local. Par conséquent, ils disposent également de leur propre réseau interne.

Dans notre application web, nous écoutons sur le port 80 les requêtes HTTP entrantes. Toutefois, ce port n'est pas accessible depuis l'extérieur du conteneur.

Pour cela, nous utilisons l'instruction **EXPOSE** suivie du numéro de port 80. Cela va annoncer à Docker qu'il devra exposer le port 80 vers la machine hôte.

Dockerfile

```
FROM node:latest
WORKDIR /app
COPY . /app
RUN npm install
EXPOSE 80
CMD ["node", "server.js"]
```

Nous en avons terminé avec la rédaction du **Dockerfile**. Voyons maintenant comment nous pouvons utiliser cette image personnalisée : la construire et lancer un conteneur basé dessus.

3.4.2.5.7. Construction de l'image

Maintenant que notre fichier Dockerfile est prêt, nous pouvons construire notre image Docker personnalisée.

Pour ce faire, ouvrez un terminal et déplacez-vous dans le répertoire où se trouve le **Dockerfile**.

Exécutez la commande suivante pour construire l'image :

```
docker build .
```

```
PS C:\Users\baptiste\Documents\Cours\SPOT_COURS\chapters\docker\images_docker\code\td01_app_nodejs> docker build .  
[+] Building 3.4s (9/9) FINISHED  
=> [internal] load build definition from Dockerfile  
=> => transferring dockerfile: 135B  
=> [internal] load .dockerrcignore  
=> => transferring context: 2B  
=> [internal] load metadata for docker.io/library/node:latest  
=> [1/4] FROM docker.io/library/node:latest  
=> [internal] load build context  
=> => transferring context: 2.16kB  
=> [2/4] WORKDIR /app  
=> [3/4] COPY . /app  
=> [4/4] RUN npm install  
=> exporting to image  
=> => exporting layers  
=> => writing image sha256:49761d04cef98d646f494b1c37ba3e1fb624750323f861bfff1b7d109ebdb0c0a  
  
What's Next?  
Lorsque l'image est créée, nous recevons son identifiant unique  
View a summary of image vulnerabilities and recommendations → docker scout quickview
```

La commande `docker build .` construit l'image Docker en utilisant le `Dockerfile` situé dans le répertoire courant (le point `.`).

Lorsque vous exécutez cette commande, Docker commence par lire le `Dockerfile` et exécute les instructions une par une, du haut vers le bas, pour construire l'image.

A l'issue de la construction, Docker affiche un message indiquant que l'image a été construite avec succès et qu'elle a reçu un identifiant unique.

Lancez maintenant la commande `docker images` pour lister les images Docker présentes sur votre machine.

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
<none>	<none>	49761d04cef9	9 minutes ago	1.11GB
node	latest	c3978d05bc68	3 weeks ago	1.1GB

Nous voyons que l'image que nous venons de construire est présente sur notre machine hôte.

Grâce à son identifiant unique, nous pouvons maintenant lancer un conteneur basé sur cette image.

```
docker container run 49761d04cef98d646f494b1c37ba3e1fb624750323f861bfff1b7d109ebdb0c0a
```

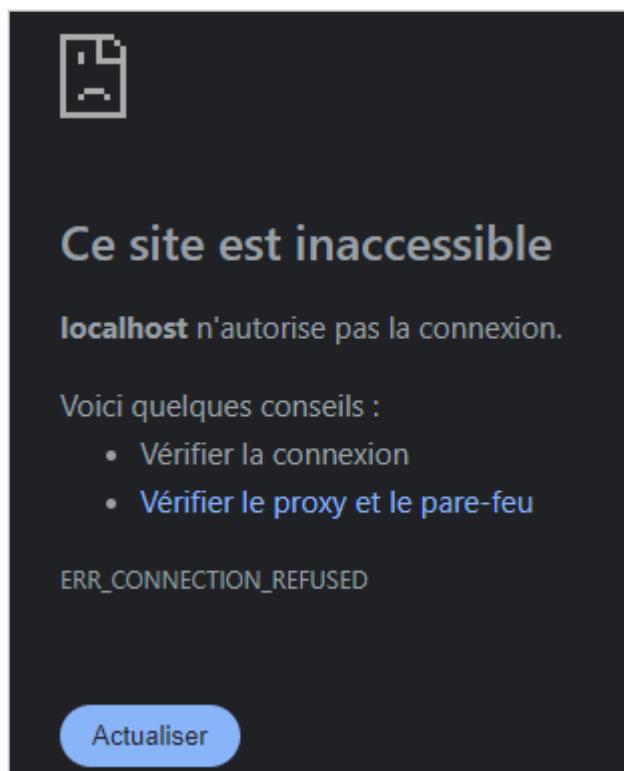
Notre conteneur est donc en cours d'exécution avec le script `server.js` qui écoute sur le port 80.

Pour vérifier, ouvrez "Docker Desktop", cliquez sur l'onglet "Containers" et vous verrez le conteneur en cours d'exécution.



Testons maintenant notre application en ouvrant un navigateur web et en tapant l'URL <http://localhost> dans la barre d'adresse.

Que constatons-nous ? L'application fonctionne-t-elle correctement ?



L'application ne fonctionne pas ! elle est totalement inaccessible. Pourtant, dans notre fichier `Dockerfile`, nous avons bien exposé le port `80`.

Que s'est-il passé ?

Premièrement, arrêtons notre conteneur, car les choses ne semblent pas se passer comme prévu, en tapant la commande suivante dans un nouveau terminal :

```
docker container ps
```

Cette commande liste les conteneurs en cours d'exécution.

Récupérez l'**identifiant** ou le **nom** de votre conteneur. Le mien se nomme `stupefied_meninsky` et possède l'identifiant `b506faabca85`.

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
b506faabca85	49761d04cef9	"docker-entrypoint.s..."	17 hours ago	Up 17 hours	80/tcp	stupefied_meninsky

Puis tapez la commande suivante pour arrêter le conteneur :

```
docker container stop b506faabca85  
ou bien  
docker container stop stupefied_meninsky
```

Une fois la procédure achevée, si vous relancez la commande `docker container ps`, vous ne devriez plus voir votre conteneur.

Pour voir votre conteneur, il faudra alors rajouter le paramètre `-a` :

```
docker container ps -a
```

Cela affichera tous les conteneurs, même ceux qui ne sont plus en cours d'exécution (**status : Exited**).

Revenons maintenant à notre problématique !

Oui, nous avons ajouté dans notre Dockerfile l'instruction `EXPOSE 80` pour exposer le port 80 de notre conteneur vers la machine hôte.

Mais concrètement, cela ne suffit pas ! Car l'instruction `EXPOSE` n'ouvre pas le port `80` de notre conteneur vers la machine hôte, elle est une facultative. Elle ne sert à rien d'autre **qu'à documenter le port** sur lequel notre application écoute. C'est une bonne pratique pour informer les autres utilisateurs de l'utilisation de ce port par votre application, et vous devez continuer à l'utiliser.

Mais nous devons faire plus !

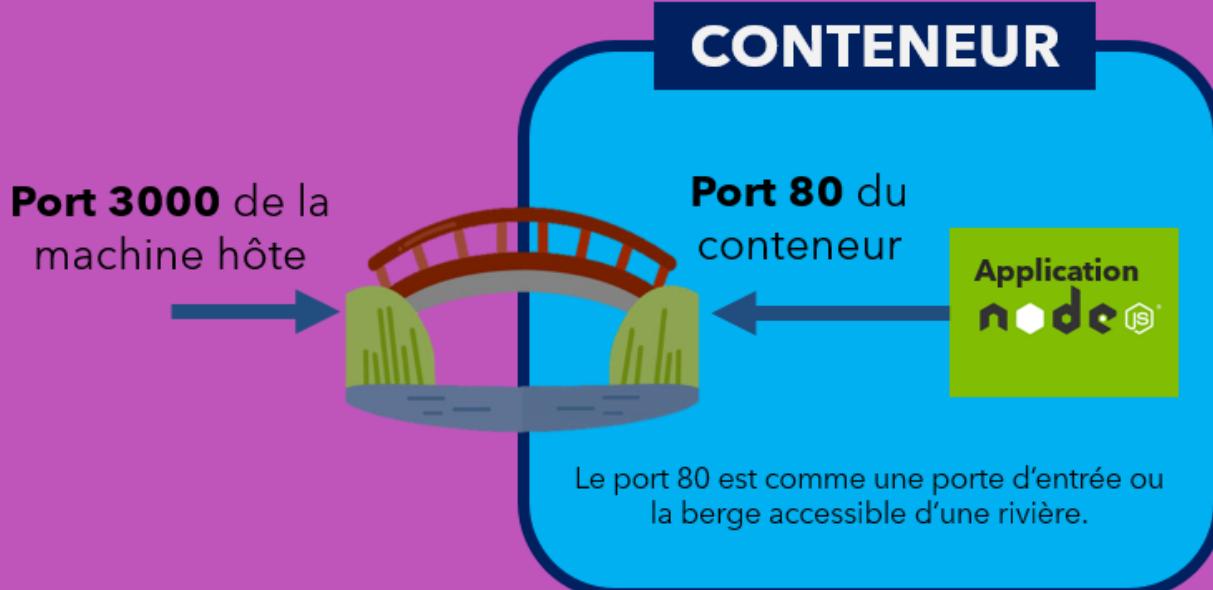
Tout d'abord, nous allons supprimer notre conteneur avant d'en créer un nouveau, qui lui, permettra de communiquer avec notre machine hôte.

```
docker container rm b506faabca85  
ou bien  
docker container rm stupefied_meninsky
```

3.4.2.5.8. Mapping de ports

MACHINE HÔTE

Pour rejoindre les deux berges, il faut un « pont », « une passerelle ». C'est ce que l'on appelle faire du **mapping de port**.



Docker container run -p 3000:80 <mon image>



Pour permettre à notre application de communiquer avec notre machine hôte, nous devons réaliser un **mapping de ports**.

Le **mapping de ports** permet de rediriger le trafic d'un port d'un conteneur vers un port de la machine hôte.

C'est comme si vous aviez **un pont** entre le port du conteneur et le port de la machine hôte.

Pour réaliser un **mapping de ports**, nous utilisons l'option **-p** suivie du numéro de port de la machine hôte, puis du numéro de port du conteneur.

Le numéro de port de la machine hôte est **un choix arbitraire**, mais il ne doit pas être utilisé par un autre service.

Le numéro de port de notre conteneur est le port sur lequel notre application écoute.

Dans notre cas, c'est le **port 80**. C'est le port que nous avons exposé dans notre **Dockerfile**. Vous comprenez alors pourquoi c'est bien pratique de préciser dans le Dockerfile le port sur lequel notre application écoute. Car nous avons besoin de cette information pour réaliser le **mapping de ports**.

Tapons la commande suivante pour lancer un nouveau conteneur avec un **mapping de ports** :

```
docker container run -p 3000:80 <identifiant de votre image>
```

Assurez-vous que le conteneur est bien en cours d'exécution. Cette fois-ci vous pouvez utiliser **Docker Desktop**, dans l'onglet "Containers".

Name	Image	Status	CPU (%)	Port(s)	Last started	Actions
awesome_kalam c10bc7ac8382	49761d04cef98d646f494b1c37ba3e1fb624750323f861bf1b7d109ebdb0c0a	Running	0%	3000:80 ↗	1 minute ago	

Cliquez sur le lien "3000:80" pour ouvrir l'application dans votre navigateur ou saisir l'URL <http://localhost:3000>.

Notre application est maintenant accessible depuis notre machine hôte.

Objectif :

Apprendre Docker!

Objectif de ce cours :

Ajouter un objectif

Vous pouvez tester l'application en saisissant un objectif dans le champ de texte et en cliquant sur le bouton **Ajouter un objectif**.

Félicitations ! Vous avez créé une image Docker personnalisée pour une application Node.js et vous avez lancé un conteneur basé sur cette image.

Vous avez appris à construire une image Docker personnalisée à partir d'une image officielle, à copier les fichiers de votre application dans l'image, à installer les dépendances de l'application, à exposer un port, à lancer un serveur Node.js et à réaliser un **mapping de ports** pour permettre à l'application de communiquer avec la machine hôte.

Maintenant, vous pouvez arrêter ce nouveau conteneur, vous disposez de tous les outils nécessaires pour le faire tout seul.



Note additionnelle :

Pour toutes les commandes Docker où un identifiant peut être utilisé, vous n'avez pas besoin de copier/coller ou d'écrire l'identifiant complet. Il suffit de copier les premiers caractères de l'identifiant (par exemple, les 4 premiers caractères) pour que Docker puisse identifier le conteneur ou l'image.

Par exemple, si l'identifiant de votre conteneur est **b506faabca85**, vous pouvez taper **b506** pour que Docker comprenne que vous faites référence à ce conteneur.

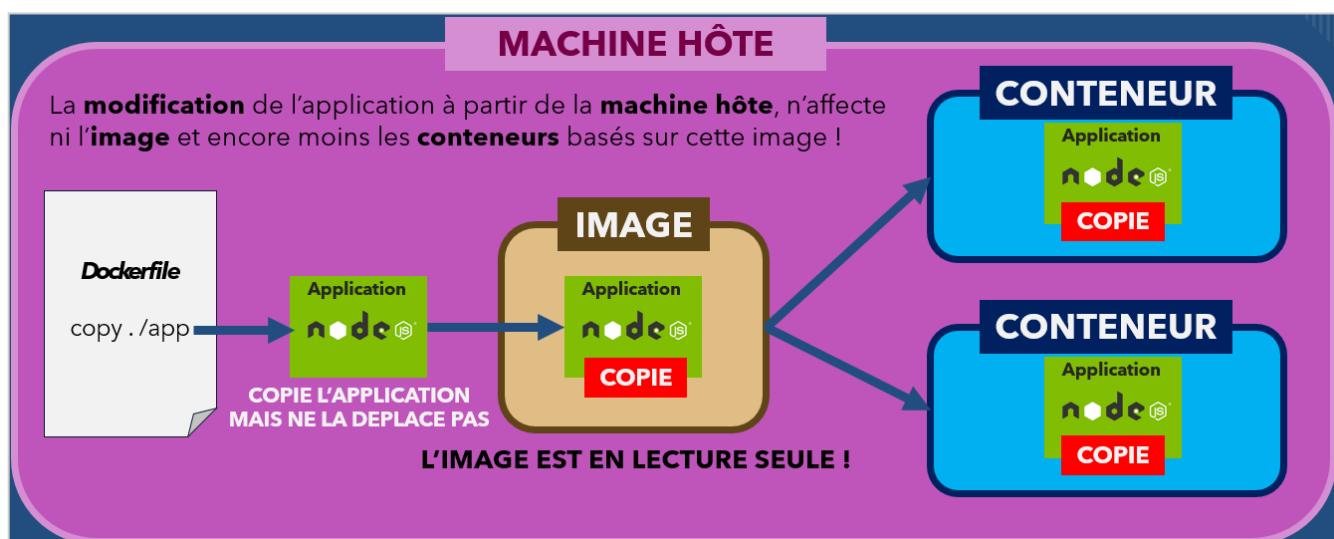
Cela s'applique à TOUTES LES COMMANDES DOCKER.

Bien entendu, nous pouvons attribuer des noms personnalisés à nos conteneurs et images pour les identifier plus facilement. Nous verrons cela dans une section ultérieure.

3.4.3. Précisions sur les images Docker

3.4.3.1. Lecture seule

Lorsque nous avons utilisé l'instruction **COPY** dans notre **Dockerfile**, nous avons copié les fichiers de notre application dans l'image Docker.



Si vous essayez de modifier vos fichiers depuis la machine hôte, et que vous relancez un conteneur basé sur cette image, vous constaterez que les modifications ne sont pas prises en compte.

Ce n'est pas un comportement anormal, c'est le fonctionnement normal des images Docker.

En effet, les images Docker sont en **lecture seule**. Cela signifie que vous ne pouvez pas modifier les fichiers d'une image une fois qu'elle a été créée.

Les fichiers présents dans l'image Docker sont une copie des fichiers de votre application à **un instant T**.



Si vous modifiez les fichiers de votre application, **vous devez reconstruire l'image Docker** pour prendre en compte ces modifications.

3.4.3.2. Comprendre les Couches d'images

Une image est fermée une fois qu'on la construit, une fois que ces instructions ont été exécutées. Sur cette base, il existe un autre concept important lié aux images Docker : les couches d'images.

Avez-vous noté le temps que cela prend pour construire une image Docker ?

Cela prend un certain temps, car Docker construit l'image instruction par instruction.

Maintenant, si vous modifiez un fichier de l'application et que vous reconstruisez l'image, Docker ne reconstruit pas toute l'image, mais seulement les parties qui ont été modifiées.

Cela permet de gagner du temps et de l'espace disque.

```
PS C:\Users\baptiste\Documents\Cours\SPOT_COURS\chapters\docker\images_docker\code\td01_app_nodejs> docker build .
[+] Building 3.4s (9/9) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 135B
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [internal] load metadata for docker.io/library/node:latest
=> [1/4] FROM docker.io/library/node:latest
=> [internal] load build context
=> => transferring context: 2.16kB
=> [2/4] WORKDIR /app
=> [3/4] COPY . /app
=> [4/4] RUN npm install
=> exporting to image
=> => exporting layers
=> => writing image sha256:49761d04cef98d646f494b1c37ba3e1fb624750323f861bfff1b7d109ebdb0c0a
```

Lorsque Docker va parcourir les étapes de construction de l'image, c'est-à-dire les instructions du Dockerfile, il va vérifier dans le cache si le résultat sera le même que la dernière fois. Si c'est le cas, Docker va utiliser le cache et ne pas reconstruire l'image.

Par contre, si une instruction a été modifiée, Docker va reconstruire l'image à partir de cette instruction et des instructions suivantes.

Par exemple, dans notre exemple :

Dockerfile

```
FROM node:latest
WORKDIR /app
COPY . /app
RUN npm install
EXPOSE 80
CMD ["node", "server.js"]
```

Si nous modifions le fichier `server.js`, Docker va reconstruire l'image à partir de l'instruction `COPY . /app`. Et comme Docker ne fait pas d'analyse poussée des fichiers, il n'est pas capable de déterminer si le résultat de `npm install` sera le même ou non. Alors, il reconstruira toutes les couches à partir de l'instruction `COPY`.

```
[+] Building 3.6s (9/9) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 135B
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [internal] load metadata for docker.io/library/node:latest
=> [1/4] FROM docker.io/library/node:latest
=> [internal] load build context
=> => transferring context: 1.24kB
=> CACHED [2/4] WORKDIR /app Si le code source de l'application est modifié,
=> [3/4] COPY . /app l'instruction COPY est rejouée, ainsi que toutes
=> [4/4] RUN npm install } les instructions suivantes.
=> exporting to image
=> => exporting layers
=> => writing image sha256:71c81241a13ce95decdfb65486eac94f70fd2499abc1e33f3693752e7b947bf3
```

C'est ce que l'on appelle une **architecture basée sur des couches**. Une image est donc simplement construite à partir de plusieurs couches basées sur ces différentes instructions.

Le seul cas particulier est celui de l'instruction **CMD** qui crée une couche supplémentaire seulement lorsqu'elle est appelée à l'exécution du conteneur.



Chaque instruction du **Dockerfile** représente une **couche**

Fort de ce que nous venons de comprendre sur le cache et les couches, nous pouvons proposer une astuce pour accélérer la construction de l'image Docker.

Nous pouvons réorganiser les instructions du **Dockerfile** pour que les instructions qui changent le moins souvent soient placées en premier.

Par exemple, si je modifie le fichier **server.js**, nous savons que l'instruction suivante **RUN npm install** sera rejouée. Or, **npm install** est une instruction qui prend du temps, et si je modifie le fichier **server.js**, je n'ai pas besoin de réinstaller les dépendances à chaque fois.

Ainsi, je peux réorganiser mon **Dockerfile** de la manière suivante : On effectue une première copie du fichier **package.json** pour installer les dépendances, puis on copie le reste des fichiers.

Dockerfile

```
FROM node:latest
WORKDIR /app
COPY package.json /app
RUN npm install
COPY . /app
EXPOSE 80
CMD ["node", "server.js"]
```

La prochaine fois que je modifierai le fichier **server.js**, la couche **RUN npm install** ne sera pas reconstruite, car Docker utilisera le cache. La reconstruction de l'image sera super rapide !

C'est une astuce à garder en tête pour optimiser la construction de vos images Docker et qui nécessite de bien comprendre le fonctionnement des couches.

3.5. Manager les images et les conteneurs

To do ...

3.6. Stopper et redémarrer les conteneurs

To do ...

3.7. Comprendre les modes attaché et détaché

To do ...

3.8. Entrer dans un conteneur en marche

To do ...

3.9. Le mode interactif

To do ...

3.10. Suppression des images et des conteneurs

To do ...

3.11. Supprimer un conteneur arrêté automatiquement

To do ...

3.12. Inspecter un conteneur

To do ...

3.13. Copier des fichiers dans et depuis un conteneur

To do ...

3.14. Nommer et tagger des conteneurs et images

To do ...

3.15. Mise en pratique

To do ...

3.16. Partager des images

To do ...

3.17. Push d'images sur Dockerhub

To do ...

3.18. Pull et utilisation d'images partagées

To do ...

3.19. Résumé

To do ...



4. Gestion des images et des conteneurs

Version 1.0.0 | Dernière mise à jour : 17/11/2024 | Auteur : Bauer Baptiste

Jusqu'à présent, nous avons vu comment créer une image et comment exécuter/arrêter un conteneur. Mais il y a beaucoup plus à faire, alors nous allons prendre un peu de temps pour comprendre comment configurer et gérer nos images et nos conteneurs.

Vous pouvez avoir un aperçu ce qui est possible de faire en saisissant l'option de commande `--help`. Par exemple, pour obtenir de l'aide sur la commande `docker image`, vous pouvez saisir :

```
$ docker image --help
```

Pour obtenir de l'aide sur la commande `docker container`, vous pouvez saisir :

```
$ docker container --help
```

À notre niveau nous n'aurons pas besoin de toutes les options disponibles, mais il est bon de savoir qu'elles existent.

4.1. Lister les conteneurs

Afficher les conteneurs en cours d'exécution, vous pouvez saisir :

```
docker container ps  
docker ps
```

Afficher tous les conteneurs, y compris ceux qui ne sont pas en cours d'exécution, vous pouvez saisir :

```
docker container ps -a
```

Il existe des filtres et des options de formatage du résultat obtenu. Regardez la liste des options disponibles pour la commande `docker container ps` :

```
docker container ps --help
```

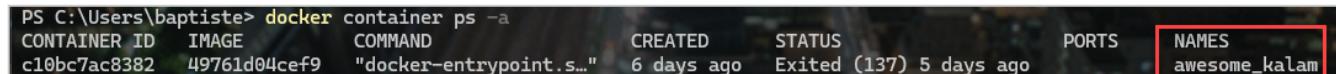
Option	Description
<code>-a, --all</code>	Montre tous les conteneurs (même ceux qui ne sont pas en cours d'exécution)
<code>-f, --filter</code>	Filtre le résultat de la commande en fonction des conditions fournies

Option	Description
--format string	Formater la sortie en utilisant un modèle.
'table': Forme de tableau (default)	'table TEMPLATE': Imprimer la sortie au format tableau en utilisant le modèle Go spécifié.
'json': Format JSON	'TEMPLATE': Imprimer la sortie au format tableau en utilisant le modèle Go spécifié.. voir https://docs.docker.com/go/formatting/ pour plus d'informations sur les templates Go.
-n, --last int	Afficher les n derniers conteneurs créés (incluant tous les états) (par défaut, -1)
-l, --latest	Afficher le dernier conteneur créé (incluant tous les états)
--no-trunc	Ne pas tronquer la sortie
-q, --quiet	Afficher uniquement les IDs des conteneurs
-s, --size	Afficher les tailles totales des fichiers

4.2. Démarrer, arrêter et redémarrer un conteneur

Pour manipuler un conteneur, il faut connaitre soit son nom, soit son ID. Afin de récupérer le nom du conteneur à démarrer par exemple, vous pouvez saisir :

```
docker container ps -a
```



CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
c10bc7ac8382	49761d04cef9	"docker-entrypoint.s..."	6 days ago	Exited (137) 5 days ago		awesome_kalam

Nous y voyons le nom du conteneur, son ID, l'image utilisée, la commande qui a été exécutée pour le démarrer, l'état du conteneur, les ports mappés, le nom du conteneur et la date de création.

Puis **démarrer** le conteneur à partir de son nom :

```
docker container start <nom du conteneur>
```

La commande **start** démarre un conteneur, mais ne le crée pas. Si le conteneur n'existe pas, vous devez le créer en utilisant la commande **run**. De plus, la configuration du conteneur ne peut pas être modifiée avec la commande **start**. Le conteneur est démarré avec la configuration existante.

Voici une liste de commandes pour :

- **Arrêter** un conteneur

```
docker container stop <nom du conteneur>
```

- Redémarrer un conteneur

```
docker container restart <nom du conteneur>
```

4.3. Comprendre le mode détaché et le mode attaché

Lorsque vous exécutez un conteneur, vous pouvez le faire en mode **attaché** ou en mode **détaché**. Il est important de comprendre la différence entre ces deux modes. Pour cela, recréons un conteneur basé sur l'image de notre application **NodeJs**, mappé sur un port différent de celui que nous avons utilisé précédemment :

```
docker run -p 3001:80 <ID DE L'IMAGE>
```

Que constatez-vous ? Le terminal est bloqué, vous ne pouvez pas saisir de nouvelles commandes. C'est parce que le conteneur est en mode **attaché**.

Le mode **attaché** est le mode par défaut. Il permet de voir les informations retournées par le conteneur s'il y en a.

Dans notre application **NodeJs**, nous avons mis une ligne de script qui affiche un message dans la console. C'est pourquoi vous devriez voir ce message dans le terminal.

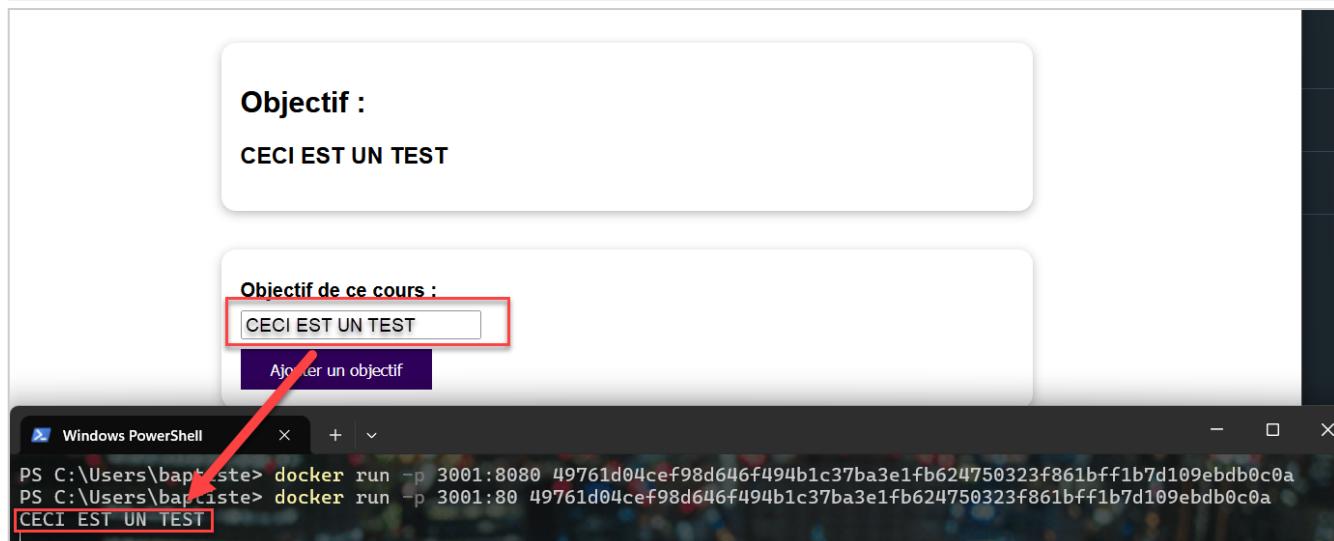
Voici le script JS :

Fichier : server.js

```
app.post('/store-goal', (req, res) => {
  const enteredGoal = req.body.goal;
  console.log(enteredGoal);
  userGoal = enteredGoal;
  res.redirect('/');
});
```

L'instruction `console.log(enteredGoal);` affiche le message saisi dans le terminal. Testons cela !

Ouvrez un navigateur et rendez-vous sur <http://localhost:3001>. Remplissez le champ de texte et cliquez sur le bouton **Ajouter un objectif**.



Retournez dans le terminal où le conteneur est en cours d'exécution. Vous devriez voir le message que vous avez saisi dans le champ de texte.

Si nous voulons exécuter un conteneur en mode **détaché**, nous devons ajouter l'option **-d** ou **--detach** à la commande **run**.

```
docker run -d -p 3001:80 <ID DE L'IMAGE>
```

Nous constatons que le conteneur s'exécute en arrière-plan et que nous pouvons saisir de nouvelles commandes dans le terminal.



Vous pouvez vous attacher à un conteneur détaché en utilisant la commande **docker attach <nom du conteneur>**.

Il est possible de voir les logs d'un conteneur en utilisant la commande **docker logs <nom du conteneur>**.

```
docker logs <nom du conteneur>
```



Et si vous ne souhaitez pas être en mode attaché mais que vous souhaitez suivre les logs en temps réel, vous pouvez utiliser l'option **-f** ou **--follow**.

```
docker logs -f <nom du conteneur>
```



Peu importe le mode dans lequel vous exécutez un **conteneur**, il est toujours **opérationnel**. Mais si vous avez besoin **d'informations à l'intérieur du conteneur**, vous pouvez utiliser les journaux/logs ou vous attacher à lui.

4.4. Le mode interactif

Docker ne se limite pas à la conteneurisation de services web comme nous l'avons vu jusqu'à présent. Vous pouvez également utiliser Docker pour exécuter des applications interactives.

Nous allons illustrer cela en utilisant une application **Python** simple construite pour la version 3 et qui ne crée aucun serveur Web.

Voici le script Python :

Fichier : code/rng.py

```
from random import randint

min_number = int(input('Please enter the min number: '))
max_number = int(input('Please enter the max number: '))

if (max_number < min_number):
    print('Invalid input - shutting down...')
else:
    rnd_number = randint(min_number, max_number)
    print(rnd_number)
```

Cette application récupérera un nombre minimum et un nombre maximum de l'utilisateur, puis générera un nombre aléatoire entre ces deux nombres qui sera affiché dans le terminal.

Nous pouvons installer Python sur notre système et exécuter ce script, mais nous allons plutôt utiliser Docker pour le faire.

Cela va mettre en évidence que Docker n'est pas utilisable uniquement pour des applications NodeJs ou des applications Web.

Et nous allons directement voir l'utilité des modes attachés et détachés.

Dans cette application Python, l'utilisateur doit réellement interagir avec l'application. Lancer le conteneur en arrière-plan n'aurait pas de sens.

4.5. Supprimer des images et des conteneurs

4.6. Supprimer automatiquement des conteneurs arrêtés

4.7. Inspecter des images

4.8. Copier des fichiers vers/depuis un conteneur

4.9. Nommer & Tagger des images et des conteneurs

4.10. Partager des images

4.11. Pusher des images vers un registre

5. Data et Volumes

Version 1.0.0 | Dernière mise à jour : 17/11/2024 | Auteur : Bauer Baptiste

5.1. Introduction

Jusqu'à présent dans ce cours, nous avons beaucoup appris sur les images et les conteneurs Docker. Nous disposons maintenant d'une base solide pour aller plus loin et explorer ces concepts de manière plus approfondie.

Ce chapitre va se concentrer sur la gestion des données à l'intérieur des images et des conteneurs. Cela peut sembler étrange au début, car nous avions déjà des données dans ces éléments, n'est-ce pas ? Par exemple, notre code y était stocké.

En fait, il existe **différents types de données**. Jusqu'à présent, nous ne pouvions en gérer qu'un seul type dans nos images et conteneurs. Or, pour une utilisation courante, cela pose des problèmes.

Nous allons donc découvrir comment les images et les conteneurs peuvent gérer les données de différentes manières. Nous verrons comment se connecter à différents dossiers et bien plus encore. Nous nous pencherons plus particulièrement sur un concept clé de Docker : **les volumes**.

Enfin, vers la fin du chapitre, nous explorerons également une notion complémentaire : les **arguments et les variables d'environnement**. Nous verrons comment les utiliser dans les images et les conteneurs Docker, et pourquoi ces éléments sont utiles.

5.2. Différents types de données

Les types de données ...		
Application (code + environnement)	Les données applicatives temporaires	Les données applicatives permanentes
Ecrites et fournies par vous (=le développeur)	Données générées pendant l'exécution de l'application	Données générées pendant l'exécution de l'application
Ajoutées dans l'image lors de sa construction	Stockées dans la mémoire ou fichiers temporaires	Stockées dans des fichiers ou dans la base de données
Fixe : Ne peut plus être modifié une fois l'image montée	Dynamiques et supprimées régulièrement	Ne doivent pas être perdues si le conteneur redémarre ou s'arrête
Lecture seule, stockées dans l'image	Lecture+Ecriture, temporaires, stockées dans le conteneur	Lecture+Ecriture, temporaires, stockées dans le conteneur

Comme je l'ai mentionné, il existe différents types de données. Afin de comprendre les problèmes que nous pourrions rencontrer et les solutions que nous aborderons dans ce chapitre, il est important de distinguer ces types.

- **Données de l'application (code source et environnements) :**

Ce sont les éléments dont nous avons le plus parlé dans les chapitres précédents. Cela comprend notre code source et l'environnement d'exécution de l'application. Ce code peut inclure des paquets tiers, comme les dépendances d'une application `Node.js` spécifiées dans un fichier `package.json`.

Ces données sont ajoutées à l'image pendant **la phase de construction**. Une fois l'image créée, le code est fixé et ne peut plus être modifié.



C'est pourquoi on les stocke dans une image : **le code source de l'application et son environnement doivent être en lecture seule.**

- **Données applicatives temporaires :**

Il s'agit des données générées pendant l'exécution de l'application. Par exemple, pour un site web, les données saisies par un utilisateur dans un forum.

Ces données peuvent être stockées temporairement en mémoire, dans des variables ou dans un fichier/base de données. Il est acceptable de perdre ces données à l'arrêt du conteneur, car elles ne sont pas critiques.

Ces données sont stockées en **lecture-écriture** dans les conteneurs (pas dans les images,) car les images sont en **lecture seule**.

Docker gère efficacement ces données en combinant le système de fichiers de l'image avec les modifications effectuées dans le conteneur (couche supplémentaire en lecture-écriture).

- **Données applicatives permanentes :**

Ces données doivent persister même après l'arrêt et le redémarrage du conteneur.

Exemple : comptes utilisateurs dans une application d'inscription.

On ne veut pas perdre les données applicatives lors de la mise à jour du code dans le conteneur. Ces données sont en **lecture-écriture** et stockées dans des volumes, un concept clé que nous allons explorer en détail dans ce chapitre.

5.3. Mise en pratique : Manipulation de données temporaires et permanentes

5.3.1. Analyse d'une application de démonstration

Voici une application de démonstration `Node.js` qui permettra d'étudier la manipulation de données temporaires et permanentes.

Ce projet nous servira à illustrer la gestion de différents types de données avec Docker.

Le code source de l'application Node.js est fourni dans les assets de ce chapitre sous le nom : `data-volumes-01-starting-setup.zip`.

L'application est simple et repose sur un serveur **Node.js** qui gère **plusieurs routes et requêtes**. Une fois lancée, elle présente un formulaire permettant la saisie de commentaires. Ces commentaires sont ensuite stockés dans un fichier.

Plus précisément, deux fichiers sont utilisés :

- Un fichier **temporaire**, stocké dans un emplacement temporaire.
- Un fichier **final**, destiné au stockage permanent des commentaires.

Fichier server.js

```
// Extrait de code
//...
    await fs.writeFile(tempFilePath, content);
exists(finalFilePath, async (exists) => {
    if (exists) {
        res.redirect('/exists');
    } else {
        await fs.rename(tempFilePath, finalFilePath);
        res.redirect('/');
    }
});
//...
```

L'application vérifie d'abord si le fichier final existe déjà. Si ce n'est pas le cas, le fichier temporaire est copié vers l'emplacement final. Le fichier final contiendra uniquement le commentaire soumis.

En plus du fichier principal du serveur **Node.js** (`server.js`), l'application comprend plusieurs dossiers :

- **pages** : contient les pages HTML renvoyées par le serveur en réponse aux requêtes.
- **feedback** : stocke définitivement les fichiers de commentaires générés.
- **temp** : sert d'emplacement temporaire pour les fichiers de commentaires avant leur validation.
- **public** : héberge les fichiers de style de l'application.

Initialement, les dossiers **temp** et **feedback** sont vides. Le dossier **temp** sert à stocker temporairement les fichiers de commentaires, avant de vérifier leur existence dans le dossier **feedback**. Si le fichier n'existe pas encore, il est copié vers l'emplacement final.

Ce projet a été conçu pour mettre en évidence **la gestion de trois types de données** :

- **Code source** : l'application **Node.js** elle-même.
- **Données temporaires** : les fichiers stockés dans le dossier **temp**, pouvant être supprimés sans perte critique.
- **Stockage permanent** : les fichiers de commentaires définitifs stockés dans le dossier **feedback**.

Nous allons maintenant conteneuriser cette application pour découvrir les défis rencontrés et les résoudre à l'aide des outils Docker.

5.3.2. Comprendre la problématique

Nous allons maintenant conteneuriser l'application en créant à la racine du projet un fichier **Dockerfile**:

```
FROM node:14 ①
WORKDIR /app ②
COPY package.json .
RUN npm install ④
COPY . .
EXPOSE 80 ⑥
CMD [ "node", "server.js" ] ⑦
```

① Nous créons une image basée sur Node version 14.

② Le répertoire de travail sera `/app`.

③ On copie le fichier `package.json` dans le répertoire de travail.

④ On lance la commande `npm install` afin de télécharger les packages de notre application dans l'image (Cela va créer une couche sur l'image spécialement pour les dépendances).

⑤ On copie tous les fichiers du répertoire dans le répertoire de travail.

⑥ On expose le port 80

⑦ Lorsque le conteneur sera monté et lancé, on exécutera la commande `node server.js`.

Maintenant que nous avons notre **Dockerfile**, nous allons pouvoir créer l'image et démarrer le conteneur pour tester l'application.

```
docker build -t feedback-node .
```

Puis démarrons un conteneur basé sur cette image :

```
docker container run -p 3000:80 -d --name feedback-app --rm feedback-node
```

- `--rm`: Quand le conteneur sera éteint, il sera supprimé automatiquement
- `-d`: Mode détaché

Ouvrez un navigateur, et allez sur : <http://localhost:3000>

MySite

Your Feedback

Title

Document Text

Save

Écrivez du contenu et cliquez sur "save".



Your Feedback

Title

Super

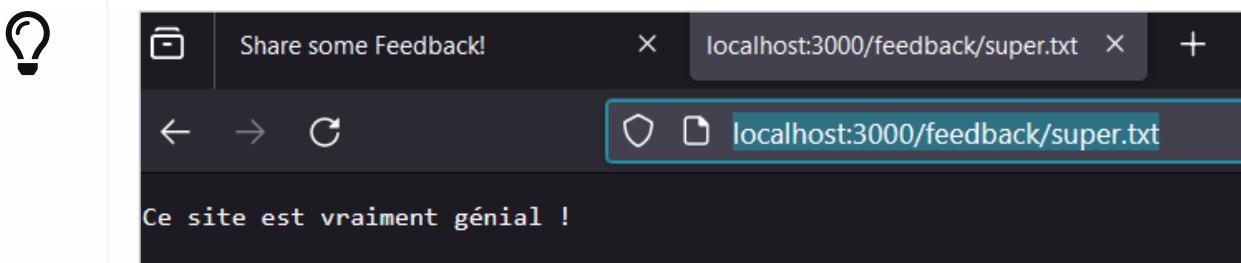
Document Text

Ce site est vraiment génial !

Save

Si vous choisissez comme titre : "Super", un fichier nommé `super.txt` sera créé.

Nous pourrons consulter son contenu à l'adresse : <http://localhost:3000/feedback/super.txt>



Ce fichier n'existe seulement que dans le conteneur, pas en local sur notre machine hôte.

Pourquoi le fichier texte "awesome.txt" n'est-il pas visible dans le dossier "feedback" sur la machine locale ?

Si vous consultez le dossier `feedback` sur votre machine locale, vous ne trouverez pas le fichier `awesome.txt`. Il n'existe qu'à l'intérieur du conteneur **Docker**, c'est pourquoi nous pouvons le voir là-bas, mais pas ici.

La raison en est que, dans le **Dockerfile**, nous copions notre dossier local dans l'image, et le

conteneur est ensuite basé sur cette image. Cela signifie que l'image, et donc aussi le conteneur, possède son propre système de fichiers, basé sur notre dossier local ici, car nous le copions dedans.



Après cela, il n'y a plus de connexion entre notre dossier local et ce système de fichiers interne de l'image.

Les conteneurs **doivent être isolés**. Cela serait plutôt mal si des fichiers étaient créés à l'intérieur du conteneur et se retrouvaient soudainement quelque part sur notre disque dur, sur notre machine hôte. **Ce n'est pas du tout l'idée derrière Docker**.

Et c'est d'ailleurs la même chose qu'auparavant dans ce cours lorsque nous avons changé quelque chose dans notre code source localement sur notre machine hôte et que la modification n'a pas été reflétée dans le conteneur Docker en cours d'exécution. Vous vous souvenez peut-être que nous avons dû :

- Reconstruire l'image, copier le code modifié,
- Créer une nouvelle image basée sur celle-ci, et ensuite les conteneurs fonctionnant sur la base de cette image modifiée auraient nos changements de code.

Pour la même raison, nous copions du code dans une image, il se trouve alors dans un système de **fichiers spécial à l'intérieur de l'image**, il est verrouillé, il n'y a pas de **connexion au dossier hôte ou à la machine hôte et au conteneur**.



Il n'y a pas de connexion entre votre conteneur ou votre image et votre système de fichiers local.

Vous avez juste initialisé son image une fois, vous pouvez copier un instantané de vos dossiers et fichiers locaux, mais après cela, c'est tout. Il n'y a pas de connexion, et c'est pourquoi nous ne voyons pas le fichier texte dans le dossier "feedback" ici sur notre machine hôte, dans ce dossier. Nous ne l'avons disponible qu'à l'intérieur du conteneur Docker en cours d'exécution. Là, nous pouvons y accéder, il existe là.

En résumé :

- Les conteneurs Docker ont leur propre système de fichiers, isolé de votre système local.
- Les fichiers copiés dans une image Docker sont "gelés" et ne sont pas synchronisés avec les fichiers sur votre ordinateur local.
- C'est pourquoi vous ne pouvez pas voir le fichier "awesome.txt" du conteneur Docker dans le dossier "feedback" sur votre machine locale.
- Pour que les modifications apportées aux fichiers locaux soient reflétées dans les conteneurs, vous devez recréer les images Docker correspondantes.



L'isolation des conteneurs est un concept fondamental de Docker. En comprenant ce concept, vous serez en mesure d'utiliser Docker de manière plus efficace et sécurisée.

Poursuivons les essais et arrêtons le conteneur :

```
docker container stop feedback-app
```

Puis, lançons un nouveau conteneur puisque le précédent a été supprimé, mais cette fois-ci sans le paramètre `--rm` :

```
docker container run -p 3000:80 -d --name feedback-app feedback-node
```

Si nous essayons de consulter le fichier précédemment créé, nous obtenons une erreur, car **il n'existe plus !**

Dans ce cas, écrivez un nouveau feedback.

Your Feedback

Title

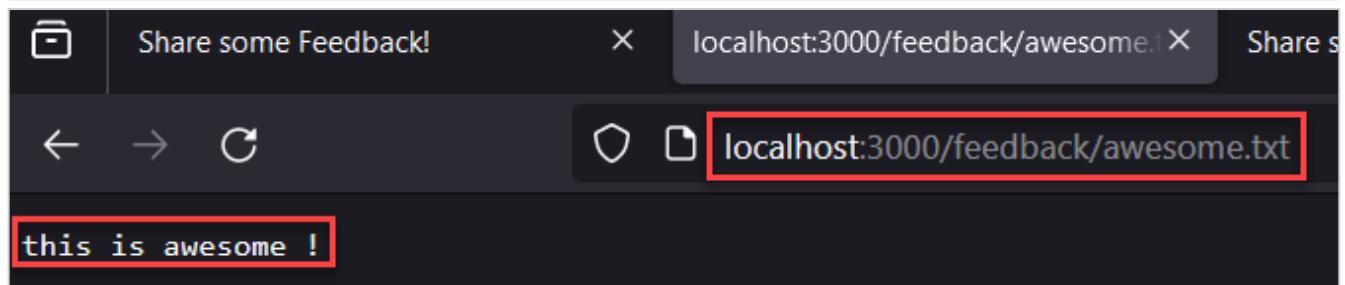
Document Text

this is awesome !

Save

On retrouve le contenu du fichier `awesome.txt` créé :



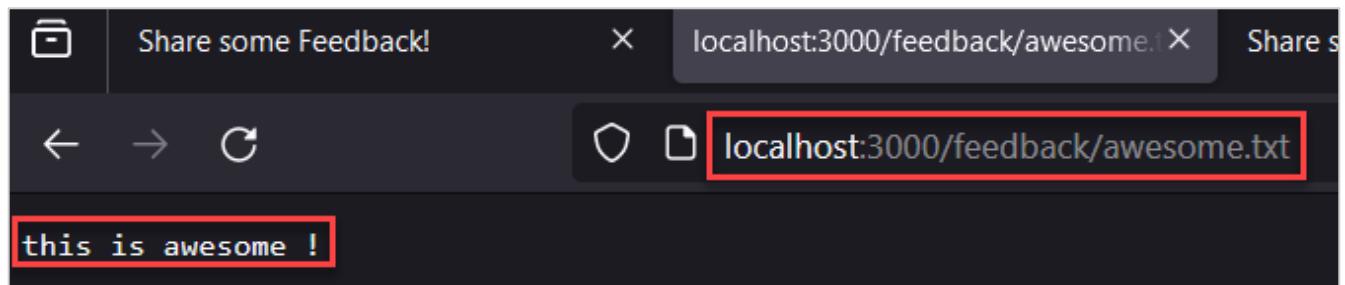


Maintenant, arrêtons le conteneur, et gardons à l'esprit que cette fois, **il ne sera pas supprimé** !

```
docker container stop feedback-app
```

Puis relançons-le :

```
docker container start feedback-app
```

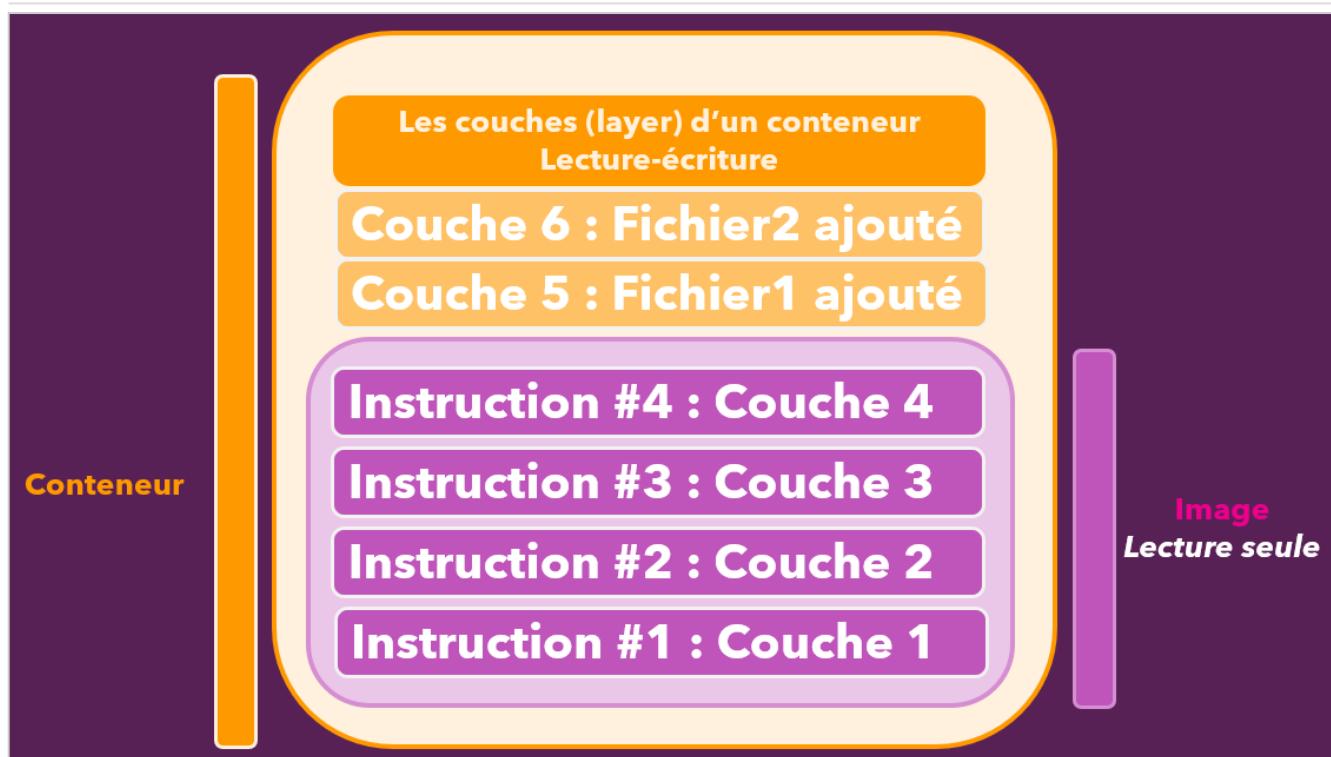


Nous constatons que le fichier est toujours présent !

Un conteneur qui est seulement stoppé, conserve les données qui sont stockées dans son système de fichier.

Chaque fichier créé, va rajouter une nouvelle couche (layer) dans le conteneur





Le système de fichiers d'un conteneur est situé à **l'intérieur de celui-ci**. Lorsque nous arrêtons et redémarrons un conteneur, le système de fichiers reste inchangé, car le conteneur lui-même n'est pas modifié. Cependant, lorsque nous supprimons un conteneur, toutes les données contenues dans celui-ci sont effacées, parce que le conteneur est supprimé dans son ensemble.

Si nous exécutons un nouveau conteneur basé sur la même image, toutes les données créées et stockées dans le conteneur précédent sont perdues, car l'image est en **lecture seule**. En effet, lorsqu'un fichier est généré dans un conteneur, **il n'est pas écrit dans l'image**, mais dans une couche en **lecture-écriture** qui est ajoutée par-dessus. Ainsi, **si le conteneur est supprimé**, nous ne disposons plus que de l'image originale, qui n'a jamais été modifiée. Lorsque nous démarrons un nouveau conteneur, celui-ci démarre avec le même système de fichiers de base, sans les modifications apportées par le conteneur précédent.

Cette isolation des conteneurs est **un concept fondamental de Docker**. Cependant, cela peut poser un problème dans certaines situations. En effet, lorsque nous supprimons un conteneur, nous perdons également toutes les données qui y étaient stockées. Dans de nombreuses applications, il est important de conserver ces données, même si le conteneur est supprimé. Par exemple, si nous gérons des comptes utilisateur ou des données produites soumises par les utilisateurs, nous ne voulons pas que ces données disparaissent soudainement.

En pratique, il est courant de supprimer des conteneurs et d'en créer de nouveaux.

Par exemple, si nous modifions notre code et créons une nouvelle image, nous n'allons pas redémarrer l'ancien conteneur, mais plutôt utiliser **le nouveau conteneur** qui utilise le dernier instantané de code. Dans ce cas, nous perdrons toutes les données stockées dans l'ancien conteneur.

Maintenant que nous avons compris le problème et que nous savons que les données stockées dans un conteneur sont perdues lorsque celui-ci est supprimé, il est important de trouver une solution pour conserver ces données importantes.

5.3.3. Introduction aux Volumes

Maintenant que nous connaissons le problème, quelle est la solution ?

Docker dispose d'une fonctionnalité intégrée appelée **volumes**, qui permet de résoudre le problème de perte de données que nous avons vu précédemment. Mais comment utiliser les volumes dans notre application ?

Tout d'abord, il est important de comprendre ce que sont les **volumes** et comment ils fonctionnent. Les volumes sont des **dossiers** situés **sur la machine hôte**, et non dans le conteneur ou l'image. Ces dossiers sont **montés**, c'est-à-dire qu'ils sont **rendus disponibles ou mappés**, dans les conteneurs.



Les volumes sont donc des dossiers sur la machine hôte, que Docker peut utiliser et qui sont ensuite mappés à des dossiers à l'intérieur d'un conteneur Docker.

Cela peut sembler similaire à l'instruction **COPY** du fichier **Dockerfile**, mais il est important de noter que cette instruction ne fait qu'une copie instantanée des fichiers et dossiers spécifiés, qui sont ensuite copiés dans l'image. **Il n'y a pas de relation ou de connexion continue.**

Avec les **volumes**, c'est différent. Vous pouvez vraiment **connecter un dossier** à l'intérieur du conteneur à un dossier à l'extérieur du conteneur, **sur la machine hôte**. Les modifications apportées à l'un des dossiers seront répercutées dans l'autre.



Si vous ajoutez un **fichier sur la machine hôte**, il sera accessible à l'intérieur du conteneur, et si le conteneur ajoute un **fichier dans ce chemin mappé**, il sera disponible à l'extérieur du conteneur, **sur la machine hôte**.

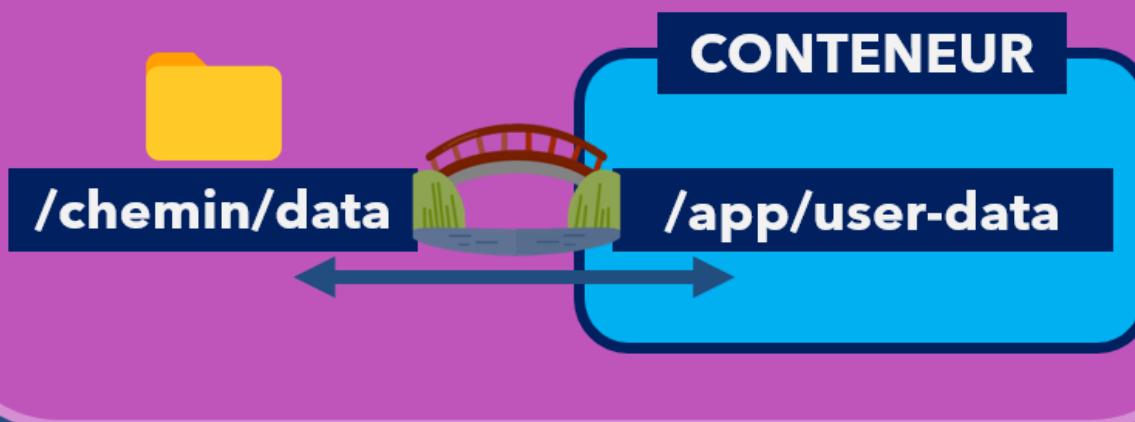
Grâce à ce mécanisme, les volumes permettent de conserver les données. Les volumes continuent d'exister même si un conteneur est arrêté.



C'est important. Si vous ajoutez un volume à un conteneur, le volume ne sera pas supprimé lorsque le conteneur sera supprimé, il survivra, et donc les données contenues dans le volume survivront également.

MACHINE HÔTE

Les volumes sont des dossiers sur le disque dur de la machine hôte qui sont **montés**, « mappés », « rendus disponible » dans les conteneurs.



Les conteneurs peuvent lire et écrire des données à partir et vers un volume. C'est une fonctionnalité puissante que nous pouvons utiliser pour les dossiers que nous voulons accéder depuis l'extérieur de notre conteneur, et/ou simplement pour les données qui doivent survivre à l'arrêt et à la suppression d'un conteneur.

Car si les données sont également enregistrées à l'extérieur du conteneur, elles survivront à la suppression du conteneur.

5.3.4. Un premier essai raté

Comment pouvons-nous maintenant ajouter un volume à notre conteneur ?

L'une des façons les plus simples d'ajouter un volume consiste à ajouter une instruction spéciale au **Dockerfile**. Nous pouvons ajouter l'instruction **VOLUME**.

Extrait du fichier Dockerfile

```
VOLUME ["]
```

Cette instruction attend un tableau où nous pouvons spécifier différents chemins dans le conteneur. Ceux-ci seront utilisés dans notre code d'application, que nous voulons conserver.

Dans notre application de démonstration, nous enregistrons les fichiers permanents dans le dossier **feedback**.

Extrait du fichier server.js

```
// some code
const finalFilePath = path.join(__dirname, 'feedback', adjTitle + '.txt');
```

```
// some code
```

Je sauvegarde également des fichiers dans le dossier `temp`, mais comme le nom l'indique, ce dossier n'est que temporaire. **Il n'a pas besoin d'être sauvegardé.**

Extrait du fichier server.js

```
// some code
const tempFilePath = path.join(__dirname, 'temp', adjTitle + '.txt');
// some code
```

Mais tout ce qui se trouve dans le dossier `feedback` doit être sauvegardé.

Dans mon code d'application, j'écris dans le dossier `feedback`, qui se trouve bien sûr à l'intérieur du dossier d'application défini par l'instruction `WORKDIR`, car c'est notre répertoire de travail général où nous avons copié l'ensemble de l'application.

Par conséquent, le volume que je souhaite enregistrer est `/app/feedback`.

Extrait du fichier Dockerfile

```
.....
VOLUME ["/app/feedback"]
CMD [ "node", "server.js" ]
```

Il s'agit du chemin à l'intérieur de mon conteneur, ce qui est important, à l'intérieur de mon conteneur, qui doit être mappé à un dossier à l'extérieur du conteneur où les données doivent donc survivre.

Vous vous demandez peut-être comment définir le dossier de la machine hôte et le faire correspondre avec le volume dans le conteneur ?



Je reviendrai sur ce point plus tard. Pour l'instant, nous laissons Docker contrôler cela, et je reviendrai sur les raisons pour lesquelles cela a du sens.

Si j'enregistre le Dockerfile avec cette instruction ajoutée, nous pouvons reconstruire l'image.

```
docker build -t feedback-node:volumes .
```

Nous ajoutons le tag `volume` à notre image `feedback-node` pour signifier que c'est la même image qu'auparavant, mais avec cette fonctionnalité de volume supplémentaire.

Supprimons l'ancien conteneur et créons un nouveau basé sur cette nouvelle image :

```
docker stop feedback-app
docker rm feedback-app
docker container run -d -p 3000:80 --rm --name feedback-app feedback-node:volumes
```

Nous pouvons également ajouter le paramètre `--rm` pour supprimer ce conteneur s'il est arrêté, car grâce aux volumes, cela ne devrait plus poser de problème.

Lançons notre application et créons un nouveau feedback :

Your Feedback

Title

Awesome

Document Text

One more time!

Save

L'application "crash", semble chercher quelque chose à l'infini ! Il est donc clair que quelque chose s'est mal passé. Et pour voir ce qui s'est mal passé, j'exécuterai `docker logs feedback-app` pour examiner la sortie de ce conteneur.

```
docker logs feedback-app
```

```
Step 7/8 : VOLUME [ "/app/feedback" ]
--> Running in 76881057cd73
docker-complete $ docker ps
docker-complete $ docker logs feedback-app
(node:1) UnhandledPromiseRejectionWarning: Error: EXDEV: cross-device li
nk not permitted, rename '/app/temp/awesome.txt' -> '/app/feedback/aweso
me.txt'
```

Il semblerait que l'erreur provienne du fait de vouloir déplacer le fichier `awesome.txt` du dossier `temp` vers le dossier `feedback`.

Et cela doit avoir quelque chose à voir avec notre volume récemment ajouté, car cela fonctionnait auparavant.

Cette erreur que nous rencontrons provient de la méthode de renommage `fs.rename` que j'utilise ici dans `Node` pour être précis.

```
await fs.rename(tempFilePath, finalFilePath);
```

Ce n'est pas un cours Node, je ne vais donc pas vous ennuyer avec les détails. Mais en fin de compte, la méthode de renommage ne fonctionne pas si le fichier est déplacé sur plusieurs périphériques

Et grâce au volume que nous avons spécifié ici, c'est en quelque sorte ce qui se passe sous le capot.

Docker ne déplace pas simplement le fichier vers un autre dossier à l'intérieur du système de fichiers du conteneur, mais il le déplace hors du conteneur.

Et la méthode `fs.rename` ne tolère pas ce genre d'opération.

La solution de contournement est simple.

Vous pouvez remplacer la méthode `fs.rename` par la méthode `fs.copyFile` et ajouter ensuite simplement une nouvelle ligne où vous appelez `fs.unlink`

Dans le fichier server.js : REMPLACEZ CECI

```
await fs.writeFile(tempFilePath, content);
exists(finalFilePath, async (exists) => {
  if (exists) {
    res.redirect('/exists');
  } else {
    await fs.rename(tempFilePath, finalFilePath);
    res.redirect('/');
  }
});
```

Dans le fichier server.js : PAR CELA

```
await fs.writeFile(tempFilePath, content);
exists(finalFilePath, async (exists) => {
  if (exists) {
    res.redirect('/exists');
  } else {
    await fs.copyFile(tempFilePath, finalFilePath);
    await fs.unlink(tempFilePath);
    res.redirect('/');
  }
});
```

Bien sûr, nous devons maintenant reconstruire notre image, car nous avons modifié le code source et cela n'est toujours pas pris en compte automatiquement à ce stade.

Il faut arrêter le conteneur qui utilise l'image :

```
docker stop feedback-app
```

Puis, nous supprimerons l'image **feedback-node:volumes**.

```
docker rmi feedback-node:volumes
```

Et reconstruisons l'image et lançons un conteneur :

```
docker build -t feedback-node:volumes .
docker container run -d -p 3000:80 --rm --name feedback-app feedback-node:volumes
```

Démarrons l'application et ajoutons un **feedback** nommé : **awesome** puis allez à l'URL : <http://localhost:3000/feedback/awesome.txt>

L'application fonctionne désormais correctement !

Faisons un autre test ...

Arrêtons le conteneur encore une fois, puis remontons un autre conteneur :

```
docker stop feedback-app
docker container run -d -p 3000:80 --rm --name feedback-app feedback-node:volumes
```

Allons à l'adresse : <http://localhost:3000/feedback/awesome.txt>

Que se passe-t-il ? Nous avons un message d'erreur disant que le fichier n'existe pas !!

Le fichier n'est plus là !

Alors, qu'est-ce qui se passe avec cette affaire de volumes ? Pourquoi est-ce que je vous enseigne des choses qui ne fonctionne pas ou qui casse notre code ? Quelle est l'idée derrière cela ? Pourquoi cela ne fonctionne-t-il pas ?

5.3.5. Les volumes à la rescousse !

Dans Docker, nous disposons de plusieurs mécanismes de stockage de données externes, deux pour être précis : les **volumes** et les **bind mounts**. Nous verrons les **bind mounts** plus tard et pour l'instant, nous allons nous concentrer sur les volumes.

Il existe deux types de volumes, chacun ayant ses propres objectifs et cas d'utilisation.

Jusqu'à présent, nous utilisions des **volumes anonymes**. Plus précisément, avec l'instruction **VOLUME** dans le **Dockerfile**, nous ajoutons un volume anonyme à cette image et les données associées aux conteneurs basés sur cette image.

Extrait d'un .Dockerfile

```
VOLUME [ "/app/feedback" ]
```

Il existe également des volumes nommés, que nous n'avons pas encore utilisés.

Dans les deux cas, **anonyme** ou **nommé**, Docker crée un dossier et un chemin d'accès sur votre machine hôte.

2 types de stockage de données externes

Volumes (Managed by Docker)

Volumes anonymes

Docker créé un dossier dans la machine hôte dont le chemin exact est inconnu par le développeur.
Sa gestion s'effectue par les commandes **docker volume**

Bind mounts (=liaison de répertoires)

Volumes nommés

Un chemin défini dans le conteneur est mappé vers le volume créé/monté. Ex: **/mon-chemin** sur la machine hôte est mappé vers **/app/data** du conteneur.

Pour les données qui doivent être persistées mais qui n'ont pas besoin d'être édités directement

Cependant, vous ne savez pas où. En effet, dans le Dockerfile, nous n'avons spécifié qu'un chemin d'accès à l'intérieur du conteneur, et pas de chemin d'accès sur la machine hôte.

Le seul moyen d'accéder à ces volumes est d'utiliser la commande **docker volume**.

```
docker volume --help
```

La commande **docker volume ls** permet de lister tous les volumes actuellement gérés par Docker.

DRIVER	VOLUME NAME
local	af18c4aa37174aa6437cd1966a5db1905abb2431d79670d0e47bbdc426fb1fd

Vous remarquerez que le nom du volume est énigmatique, car il est généré automatiquement puisqu'il s'agit d'un volume anonyme.

Si nous arrêtons le conteneur de notre application **feedback**, ce volume anonyme disparaît. En effet, il n'est géré que tant que le conteneur existe.



Cela ne résout pas le problème de la disparition des données à l'arrêt

du conteneur.

Nous verrons plus tard dans quel cas les volumes anonymes peuvent être utiles, mais examinons maintenant les volumes nommés.

Le concept clé des **volumes nommés** (et des **bind mounts** que nous verrons plus tard) est le suivant :

- **Un chemin d'accès défini dans un conteneur** est mappé à un volume créé, c'est-à-dire à un **chemin d'accès créé sur la machine hôte**.

Cependant, **le chemin sur la machine hôte est inconnu**, car géré par Docker et caché quelque part sur votre ordinateur. Il n'est pas destiné à être accessible directement.

Heureusement, avec **les volumes nommés**, les volumes persistent au redémarrage du conteneur.



Les dossiers sur votre disque dur seront conservés.

Ainsi, si vous démarrez de nouveaux conteneurs par la suite, les volumes seront de nouveau disponibles, et toutes les données stockées dans ce dossier le seront également.

Les **volumes nommés** sont donc parfaits pour les données persistantes, celles que vous n'avez pas besoin de modifier ou de visualiser directement, car vous n'avez pas vraiment accès à ce dossier sur votre machine hôte.

Nous ne pouvons pas créer de **volumes nommés dans un Dockerfile**, nous devons donc supprimer cette instruction de notre **.Dockerfile** de l'application **Feedback**.

Supprimer dans le Dockerfile l'instruction : **VOLUME**

Et reconstruire l'image !

```
docker rmi feedback-node:volumes  
docker build -t feedback-node:volumes .
```

Nous devons créer un volume nommé lors du démarrage d'un conteneur.

L'option **-v** de la commande docker run permet d'ajouter un volume à un conteneur.

Contrairement au **Dockerfile**, il s'agit ici d'un volume nommé.

Nous spécifions toujours **le chemin d'accès à l'intérieur du système de fichiers du conteneur que nous voulons enregistrer**. Devant ce chemin, nous indiquons maintenant un nom de notre choix, par exemple **feedback** qui correspondra à un nom de volume.

```
docker run -d -p 3000:80 --rm --name feedback-app -v feedback:/app/feedback feedback-node:volumes
```

Docker va alors créer un dossier sur la machine hôte qui sera nommé **feedback**.

Volumes		Give feedback	
		Status	Created
Name	Actions	Size	Actions
feedback		in use	6 seconds ago 0 Bytes ⏪ ⏴ ⏵ ⏷ ⏸ ⏹

La différence principale avec les volumes anonymes est que les volumes nommés ne seront pas supprimés par Docker lorsque le conteneur s'arrête. Les volumes anonymes sont supprimés, car ils sont recréés chaque fois qu'un conteneur est créé.

Si nous arrêtons maintenant le conteneur avec `docker stop feedback-app`, le volume sera toujours là, même si le conteneur a été supprimé à cause de l'option `--rm`.

Par conséquent, si nous redémarrons un nouveau conteneur avec le même volume, nos données seront toujours présentes.

Vérifions d'abord le volume avec `docker volume ls`. Vous verrez que le nom que nous avons choisi est toujours là, même si le conteneur a été arrêté.

Enfin, si nous redémarrons le conteneur en utilisant `docker run` avec la même option `-v` et le même nom de volume, les données seront toujours là.



Nous avons donc finalement réussi à conserver les données à l'aide des volumes nommés.

Supprimer des volumes anonymes

Nous avons vu que les **volumes anonymes** sont supprimés automatiquement lorsqu'un conteneur est supprimé.

Cela se produit lorsque vous démarrez/exécutez un conteneur avec l'option `--rm`.

Toutefois, si vous démarrez un conteneur sans cette option, le **volume anonyme NE sera PAS supprimé**, même si vous supprimez le conteneur (avec `docker rm ...`).

Cependant, si vous recréez et réexécutez le conteneur, (c'est-à-dire si vous exécutez à nouveau `docker run ...`), un nouveau volume anonyme sera créé. Donc, même si le volume anonyme n'a pas été supprimé automatiquement, il ne sera pas non plus utile, car un autre volume anonyme sera attaché la prochaine fois que le conteneur démarrera (c'est-à-dire que vous avez supprimé l'ancien conteneur et en avez exécuté un nouveau).

Vous finirez par accumuler une pile de volumes anonymes non utilisés.

Vous pouvez les supprimer via :

- `docker volume rm <nom ou id du volume>`

ou

- `docker volume prune`

5.3.6. Les Liaisons de répertoires (bind mounts)

Nous allons aborder les **liaisons de répertoires** ("bind mounts") avant de revenir aux volumes anonymes.

Un problème récurrent en développement est que toute modification apportée au code source (par exemple dans le fichier `server.js` ou un fichier HTML) n'est pas immédiatement reflétée dans le conteneur en cours d'exécution, à moins de reconstruire l'image Docker.

Imaginons que nous ayons une application en cours d'exécution. Si nous ajoutons "s'il vous plaît" après "votre avis" dans le fichier `feedback.html` et que nous sauvegardons ce fichier, en rechargeant la page, nous ne verrons pas apparaître "s'il vous plaît". Cela se produit, car nous ne copions qu'un **instantané** du dossier dans l'image Docker lors de sa création. Les modifications ultérieures ne sont donc pas reflétées dans l'image, et par conséquent dans le conteneur.

En phase de développement, il est crucial que ces changements soient immédiatement pris en compte sans avoir à reconstruire l'image et redémarrer le conteneur à chaque modification.

Les liaisons de répertoires sont similaires aux volumes, mais avec une différence clé :

- **Les volumes**, eux, sont gérés par Docker et nous ne savons pas où ils se trouvent sur notre système hôte.
- Les **liaisons de répertoires (bind mount)** quant à eux, nous permettent de définir le chemin exact sur notre machine hôte où le chemin interne du conteneur doit être mappé.

En utilisant une liaison de répertoires pour notre code source, nous assurons que le conteneur utilise le code le plus récent, plutôt que l'instantané initialement copié dans l'image. Cela est particulièrement utile pour les **données persistantes et modifiables**.

Les liaisons de répertoires ne peuvent pas être définies dans le **Dockerfile** car ils sont spécifiques **aux conteneurs et non à l'image**. Nous devons les configurer depuis **le terminal lors de l'exécution du conteneur**.

Tout d'abord, arrêtons le conteneur en cours d'exécution :

```
docker stop feedback-app
```

Ensuite, créons un nouveau conteneur avec un montage de liaison qui va lier un répertoire de la machine hôte avec l'un de notre conteneur.

```
docker container run -d -p 3000:80 --rm --name feedback-app -v  
/chemin/absolu/de/votre/projet:/app feedback-node:volumes
```

Remplacez : `/chemin/absolu/de/votre/projet` par le chemin du répertoire contenant vos codes sources. Pour ma part le chemin sera :

```
C:\Users\baptiste\Desktop\demo_docker\data-volumes-docker
```

```
docker container run -d -p 3000:80 --rm --name feedback-app -v C:\Users\baptiste\\Desktop\demo_docker\data-volumes-docker:/app feedback-node:volumes
```

Nous pouvons aussi mettre le mapping entre des doubles quotes si les caractères spéciaux de votre chemin pose un problème :

```
docker container run -d -p 3000:80 --rm --name feedback-app -v "C:\Users\baptiste\\Desktop\demo_docker\data-volumes-docker:/app" feedback-node:volumes
```

Si vous exécutez la commande, tout semble fonctionner correctement, mais nous constatons que le conteneur a été supprimé ! Retirons alors le paramètre `--rm`

```
docker container run -d -p 3000:80 --name feedback-app -v "C:\Users\baptiste\Desktop\\demo_docker\data-volumes-docker:/app" feedback-node:volumes
```

Si vous ne voulez pas toujours copier et utiliser le chemin complet, vous pouvez utiliser ces raccourcis :



macOS / Linux : `-v $(pwd):/app`

Windows : `-v "%cd%":/app` ou

Et regardons maintenant les `logs` pour comprendre pourquoi l'application a le status `Exited`.

```
docker logs feedback-app
```

```
PS C:\Users\baptiste\Downloads\data-volumes-02-added-dockerfile\data-volumes-02-added-dockerfile> docker logs feedback-app
internal/modules/cjs/loader.js:934
  throw err;
^

Error: Cannot find module 'express'
Require stack:
- /app/server.js
  at Function.Module._resolveFilename (internal/modules/cjs/loader.js:931:15)
  at Function.Module._load (internal/modules/cjs/loader.js:774:27)
  at Module.require (internal/modules/cjs/loader.js:1003:19)
  at require (internal/modules/cjs/helpers.js:107:18)
  at Object.<anonymous> (/app/server.js:5:17)
  at Module._compile (internal/modules/cjs/loader.js:1114:14)
  at Object.Module._extensions..js (internal/modules/cjs/loader.js:1143:10)
  at Module.load (internal/modules/cjs/loader.js:979:32)
  at Function.Module._load (internal/modules/cjs/loader.js:819:12)
  at Function.executeUserEntryPoint [as runMain] (internal/modules/run_main.js:75:12) {
  code: 'MODULE_NOT_FOUND',
  requireStack: [ '/app/server.js' ] }
```

Le code de notre application ne s'est pas exécuté, car il manque le module Express ! Pourtant, si l'on regarde le fichier `Dockerfile` :



```
1 FROM node:14
2
3 WORKDIR /app
4
5 COPY package.json .
6
7 RUN npm install
8
9 COPY . .
10
11 EXPOSE 80
12
13 CMD [ "node", "server.js" ]
```

Nous constatons que l'image contient bien les instructions d'installation des dépendances de l'application. Express devrait en théorie être inclus !

Si vous ne voulez pas toujours utiliser le chemin complet d'un volume, vous pouvez utiliser ces raccourcis qui prendront pour valeur le chemin du répertoire dans lequel vous êtes avec la console :

macOS / Linux : `-v $(pwd):/app`



Windows : `-v "%cd%":/app` ou `-v $PWD/`

Je ne les utilise pas dans les cours, car je veux montrer une approche qui fonctionne pour tout le monde, mais vous pouvez utiliser ces raccourcis en fonction du système d'exploitation sur lequel vous travaillez pour économiser du temps de saisie.

5.3.7. Combiner et fusionner différents volumes

Considérons cette erreur générée précédemment, quel pourrait être exactement le problème avec ce volume nouvellement ajouté ici ?

Etudions le [.Dockerfile](#) :

```
FROM node:14

WORKDIR /app

COPY package.json .

RUN npm install

COPY . .

EXPOSE 80

CMD [ "node", "server.js" ]
```

Dans le **Dockerfile**, nous copions `package.json` puis lançons un `npm install`. Ensuite, tout le dossier de notre application est copié initialement lorsque l'image est créée, et nous installons toutes les dépendances.

Le fichier `server.js` a besoin du package `express` qui existe dans le conteneur, grâce à `npm install` réalisé lors de la création de l'image, mais qui n'existe pas dans ma configuration locale, car je n'ai jamais exécuté `npm install` là-bas.

Gardez à l'esprit que lorsque nous créons le conteneur avec l'option `-v`, nous montons le contenu du dossier, dans `/app` du conteneur.

En fin de compte, cela signifie que nous **écrasons le dossier de l'application** à l'intérieur du conteneur avec notre dossier local !

Les étapes du Dockerfile deviennent alors inutiles, car en local nous n'avons pas le dossier `node_modules` avec toutes les dépendances nécessaires à cette application, et c'est la raison pour laquelle notre application génère l'erreur vue précédemment.

Alors, comment pouvons-nous résoudre ce problème ?

Comprendons d'abord comment les conteneurs interagissent avec les volumes et les répertoires liés.

Si nous avons un conteneur, et que nous n'avons pas de volume ni de répertoire lié, nous pouvons monter les deux dans le conteneur avec l'option `-V`, que je vous ai montrée il y a quelques minutes.

Cela signifie que certains dossiers à l'intérieur du conteneur sont montés, ou sont connectés, à des dossiers de la machine hôte.



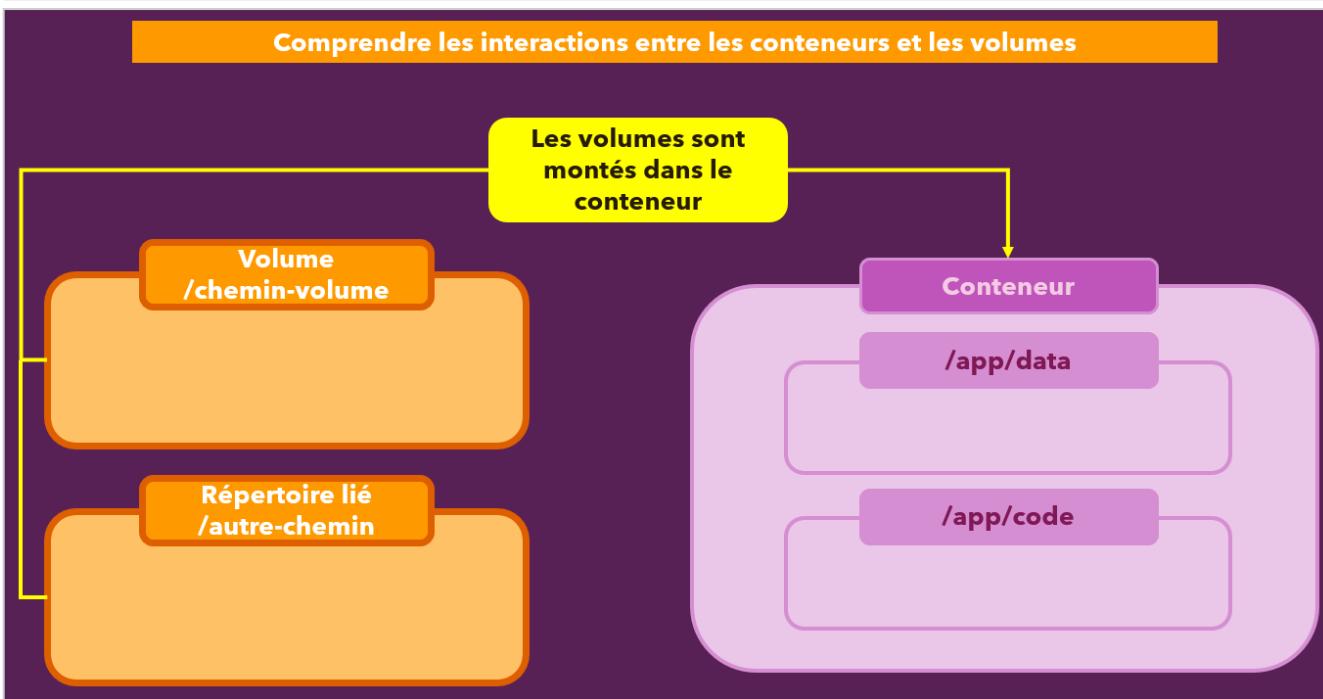


Figure 3. Volumes liés aux dossiers d'un conteneur.

Maintenant, imaginons que nous avions déjà des fichiers à l'intérieur du conteneur au moment du montage. Dans ce cas, ces fichiers existeront également dans le volume extérieur, et si vous créez un nouveau fichier dans le dossier `/app/data` du conteneur, il sera également ajouté dans le dossier de la machine hôte.

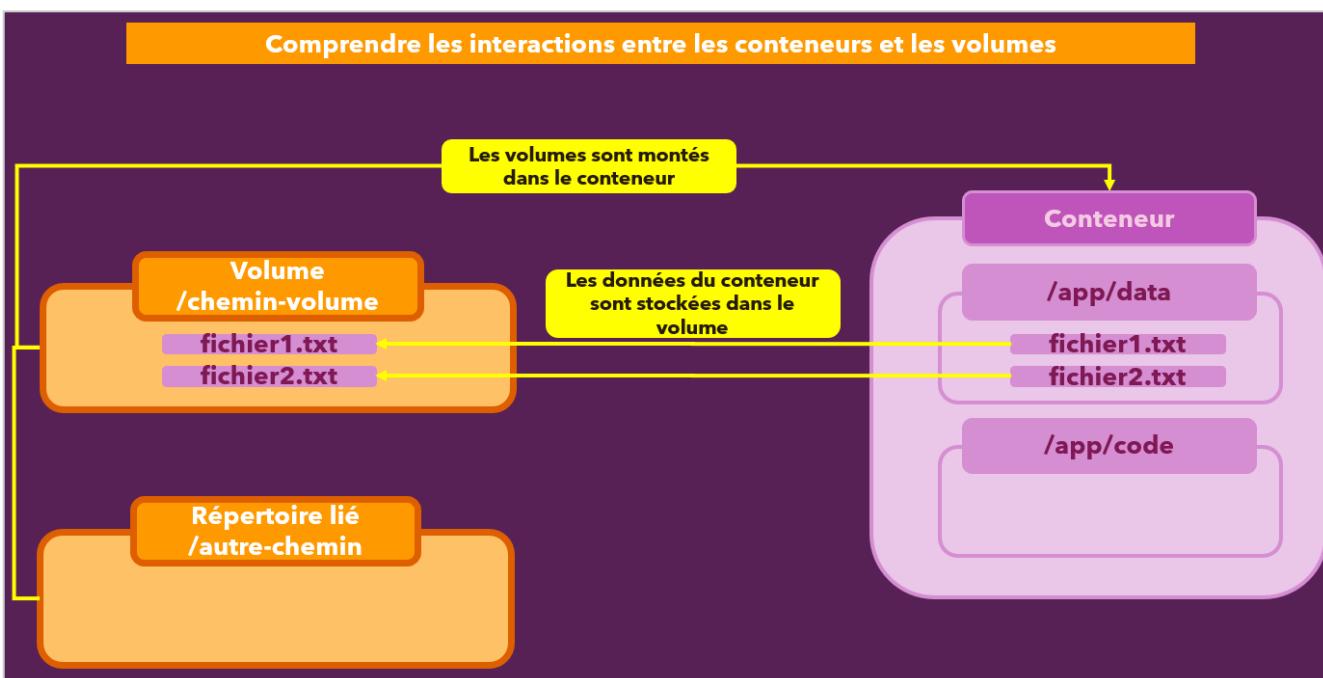


Figure 4. Les données présentes initialement dans le conteneur apparaissent dans le volume local.

Prenons un autre cas de figure.

Si le conteneur se lance, et qu'il trouve des fichiers dans le volume, et qu'il n'a pas encore de fichiers en interne comme dans le répertoire `/app/code` associé, il charge alors les fichiers du volume. C'est ce que nous utilisons avec les répertoires liés (*Bind Mount*).

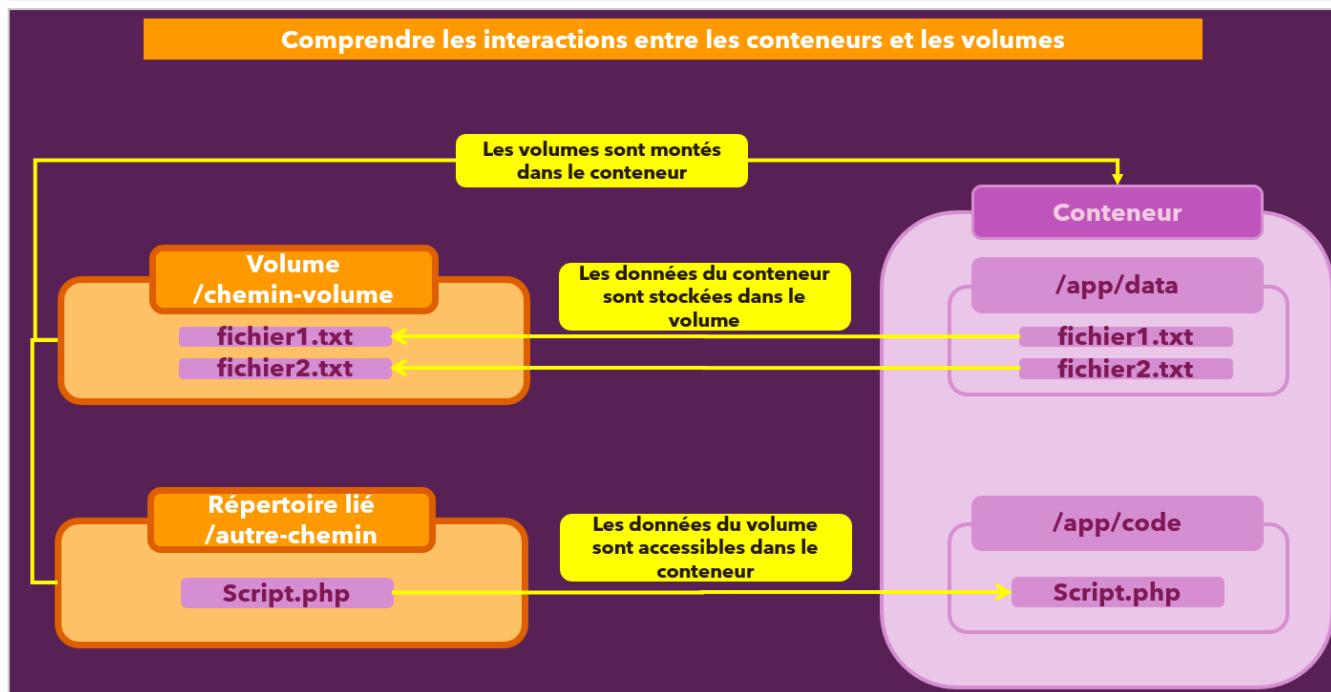


Figure 5. Les données présentes initialement dans le volume apparaissent dans le conteneur.

Revenons à l'étude de notre Dockerfile et de notre image générée :

Dans le conteneur, nous aurons des fichiers dans le dossier `/app` à cause des instructions suivantes :

```
FROM node:14

WORKDIR /app

COPY package.json .

RUN npm install

COPY . .

EXPOSE 80

CMD [ "node", "server.js" ]
```

À l'intérieur du conteneur, nous avons des fichiers dans le dossier de l'application en raison des instructions spécifiées. De plus, nous avons des fichiers et des dossiers à l'extérieur du conteneur, dans le dossier correspondant sur notre machine hôte locale, grâce au paramètre `-v`.

Docker ne remplace pas les fichiers de notre machine hôte, car ce serait désastreux. Une telle action pourrait entraîner la perte de nombreuses données importantes sur notre ordinateur par inadvertance. Ainsi, ce n'est pas ce qui se passe ici.

Docker ne remplacera pas notre dossier local dans ce contexte. En revanche, ce sont les dossiers de la machine hôte et leur contenu qui écraseront ceux présents dans le conteneur Docker. C'est là que réside le problème, car cela entraîne la suppression de `node_modules` et d'autres éléments similaires.

Pour résoudre ce problème, il est nécessaire d'informer **Docker** que certaines parties de son système de fichiers interne ne doivent pas être écrasées par des fichiers externes en cas de conflit, comme c'est le cas ici. Cela peut être accompli **en ajoutant un autre volume au conteneur Docker, spécifiquement un volume anonyme**. Un volume est considéré comme anonyme lorsqu'il n'a pas de nom explicite. Ajoutons un volume anonyme supplémentaire avec `-v app/node_modules` lors de la création du conteneur. Il aurait eu un nom si nous aurions ajouté un libellé devant avec les : comme : `-v nom_volume:app/node_modules`. En l'absence, c'est bien un volume anonyme.

```
docker container run -d -p 3000:80 --name feedback-app -v "C:\Users\baptiste\Desktop\demo_docker\data-volumes-docker:/app" -v app/node_modules feedback-node:volumes
```

Nous aurions pu aussi ajouter dans le `.Dockerfile` une instruction équivalente :

```
VOLUME ["app/node_modules"]
```

Ce qui nous obligerait ici à reconstruire notre image. Pour éviter cela, nous intégrerons ce volume via le paramètre `-v`.

Pourquoi ce volume est-il si utile ici ?

Docker évalue toujours tous les volumes définis sur un conteneur, et en cas de conflit, le chemin interne le plus long l'emporte. Par exemple, ici, nous avons un conflit :

- Un volume `app` est lié à quelque chose, et un volume `app/node_modules` est également lié à quelque chose.

 Même si nous n'avons pas attribué de nom, gardez à l'esprit que même les volumes anonymes sont gérés par Docker, et il y a un dossier mappé quelque part sur la machine hôte. Ce dossier est simplement supprimé lorsque le conteneur est supprimé, mais il existe sur la machine hôte, même pour les volumes anonymes.

Ainsi, Docker voit qu'il y a **un volume mappé** au dossier `app` et **un volume mappé** au dossier `app/node_modules`. La **règle simple** de Docker est que le chemin le plus long et le plus spécifique l'emporte.

Cela signifie que nous pouvons toujours mapper un dossier de la machine hôte dans le dossier `app` du conteneur, mais le dossier `node_modules` à l'intérieur de `app`, qui est créé par la commande `npm install`, survivra et écrasera le dossier éventuel du même nom, venant de l'extérieur. Dans notre exemple, puisque nous ne passons en fait aucun dossier `node_modules` de l'extérieur, le dossier `node_modules` créé pendant la création de l'image survivra et coexistera avec le mappage.

Après cette explication, si nous arrêtons et supprimons le conteneur en cours d'exécution, nous pouvons le relancer avec ce volume anonyme supplémentaire.

En ajoutant également l'option `--rm`, nous démarrons le conteneur et l'application fonctionne à nouveau.

Pour tester cela, nous voyons sous `feedback`, que le fichier `awesome.txt` est toujours présent.

Maintenant, nous avons un avantage supplémentaire :

Si nous modifions quelque chose dans notre fichier **HTML**, comme supprimer le texte "please" et sauvegarder, en rechargeant, nous voyons le changement instantanément sans avoir à reconstruire l'image. Cela est possible grâce à ce montage lié, qui fonctionne, car nous avons ajouté ce volume anonyme pour nous assurer que le dossier `node_modules` ne soit pas écrasé par le contenu de notre montage lié.

5.3.7.1. (NODEJS) Utiliser Nodemon dans un conteneur

Nous avons vu que nous pouvons éditer le fichier HTML et voir le rendu en rechargeant la page du navigateur !

Essayons maintenant de modifier le fichier `server.js` :

```
25 app.post('/create', async (req, res) => [
26   const title = req.body.title;
27   const content = req.body.text;
28
29   const adjTitle = title.toLowerCase();
30
31   const tempFilePath = path.join(__dirname, 'temp', adjTitle + '.txt');
32   const finalFilePath = path.join(__dirname, 'feedback', adjTitle + '.txt');
33
34   console.log("TEST");
35
36   await fs.writeFile(tempFilePath, content);
37   await fs.rename(tempFilePath, finalFilePath);
```

Même en rafraîchissant le navigateur, la modification n'a pas été prise en compte.

(Ajoutez un Feedback et consultez les logs du conteneur : `docker logs feedback-app`).

Le code Javascript est exécuté par NodeJs qui génère donc le rendu des pages et les renvoient au navigateur.

Nous devons donc redémarrer le serveur NodeJs qui se trouve dans le conteneur.

La solution la plus simple est d'arrêter le conteneur et de le relancer.

```
docker container stop feedback-app
docker container start feedback-app
```

Si vous aviez mis l'option `--rm` lors de la création du conteneur, alors créez de nouveau le conteneur avec la commande vue plus haut, car il aura été supprimé lors de son arrêt.

```
docker container run -d -p 3000:80 --name feedback-app -v "C:\Users\baptiste\Desktop\demo_docker\data-volumes-docker:/app" -v app/node_modules feedback-node:volumes
```

Testez de nouveau l'application et consultez les [logs](#).

Vous devriez être en mesure de créer un nouveau **Feedback** et de voir le message "TEST" dans les [logs](#) du conteneur.

Afin d'éviter de redémarrer le serveur NodeJs à chaque modification, nous allons installer un utilitaire qui se nomme **Nodemon** et qui a pour mission de détecter quand un fichier JS est modifié et de redémarrer notre serveur NODe automatiquement.

Ajoutons dans le fichier `package.json` de l'application, l'installation de `nodemon` ainsi que son utilisation via une commande `npm start`:

```
{
  "name": "data-volume-example",
  "version": "1.0.0",
  "description": "",
  "main": "server.js",
  "author": "",
  "license": "ISC",
  "scripts": {
    "start": "nodemon server.js"
  },
  "dependencies": {
    "body-parser": "^1.19.0",
    "express": "^4.17.1"
  },
  "devDependencies": {
    "nodemon": "3.1.4"
  }
}
```

Puis, nous modifierons l'instruction CMD du Dockerfile par:

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

Supprimons l'image courante et les conteneurs, puis reconstruisons une nouvelle image basée sur cette nouvelle configuration et recréons un conteneur.

Lorsque nous modifions nos scripts Javascript, maintenant, le serveur Nodejs se met à jour automatiquement.

 Si vous êtes sous Windows avec WSL 2 pour lancer Docker, vous constaterez que les changements ne sont pas détectés !!

Habituellement, les frameworks de notification de fichiers dépendent de inotify sur Linux, qui lui-même repose sur le système de fichiers du noyau. Si les fichiers sont modifiés sur l'hôte, le noyau de l'hôte enregistrera les événements du système de fichiers. Étant donné que Docker Desktop exécute toujours le moteur Docker dans une machine virtuelle utilitaire (utility vm), le conteneur utilisera le noyau de la machine virtuelle utilitaire et, par conséquent, ne pourra pas enregistrer les événements du système de fichiers de l'hôte.

Reportez-vous au sous-titre suivant pour résoudre ce problème :

5.3.7.2. (NODEJS) Windows, WSL2 et Bond Mounts

Lorsqu'on travaille avec Windows et WSL2, on peut rencontrer des problèmes liés à l'utilisation des "Bind Mounts". Veuillez noter que ces instructions ne s'appliquent que si vous utilisez Windows avec WSL2 et Docker. Elles ne s'appliquent pas si vous n'utilisez pas WSL2 !

En particulier, les événements de modification de fichiers pourraient ne pas être propagés au conteneur - c'est-à-dire que les processus s'exécutant à l'intérieur du conteneur (comme "nodemon") ne sont pas informés des modifications des fichiers.

Il existe trois principales solutions et contournements :

5.3.7.2.1. Solution efficace, simple, mais sale ...

Cette solution est « rapide et sale » mais elle fait le travail.

Pour Nodemon, vous pouvez passer du mode "notifie-moi" "je recherche moi-même les modifications" en modifiant le script de démarrage dans le fichier `package.json` comme ceci :

```
{  
  "scripts": "nodemon -L server.js"  
}
```

Pour les applications **React** (où nous voulons également surveiller les modifications de fichiers), vous pouvez utiliser une astuce similaire.

Ajoutez simplement une variable d'environnement à votre commande docker run :

```
docker run -e CHOKIDAR_USEPOLLING=true <options> <image>
```

5.4. Volumes et "Bind Mounts" : Résumé

coming soon ...

5.5. Zoom sur les volumes en lecture seul

coming soon ...

5.6. Gestion des volumes Docker

coming soon ...

5.7. Utilisation de "COPY" vs "BIND MOUNTS"

coming soon ...

5.8. Le fichier ".dockerignore"

coming soon ...

5.9. Variables d'environnement et sécurité

coming soon ...

5.10. Utiliser les arguments (ARG) lors d'un Build

coming soon ...

5.11. Résumé

coming soon ...

6. Networking : La communication entre conteneurs

Version 1.0.0 | Dernière mise à jour : 17/11/2024 | Auteur : Bauer Baptiste

6.1. Introduction

Dans ce chapitre, nous allons explorer la manière dont les conteneurs peuvent communiquer entre eux. Pour cela nous allons nous appuyer sur une application de démonstration qui nécessite de communiquer avec des services distants : une API et une base de données.

Vous avez dit 'API' ?

Dans le cadre du développement d'applications, il est courant de créer une base de données, comme une base de données SQL, et de la solliciter pour obtenir des données que nous utilisons directement dans notre application. Cependant, il existe une autre approche : développer une application distincte, destinée exclusivement à gérer les données de la base de données, c'est-à-dire l'ajout, la modification et la suppression.

Cette application spécifique ne possède pas d'interface utilisateur. Au lieu de cela, les actions sont accessibles uniquement via des routes ou des URL qui transmettent, par le biais du protocole HTTP, des requêtes de publication (méthode POST), de modification (méthode PUT), de suppression (méthode DELETE) ou de sélection (méthode GET). Chaque action renvoie généralement une réponse du serveur au format JSON.

Par exemple, le service de Météo Nationale fournit des données météorologiques que vous pouvez intégrer dans votre application pour afficher les prévisions météorologiques. Ces données sont accessibles via une ou des URL que vous pouvez interroger pour récupérer les informations. On appelle cela une **API**.

Cette approche est communément appelée API REST (Application Programming Interface Representational State Transfer). Par exemple, l'URL <https://swapi.dev/api/people/1> permet de récupérer les données du personnage de Star Wars dont l'identifiant est 1.

Nous avons déjà vu comment conteneuriser une application simple. Nous allons maintenant compliquer un peu les choses en ajoutant des services externes à notre application.

Si vous avez bien compris le fonctionnement de Docker, vous devriez être en mesure de créer une image Docker contenant une application ainsi que sa propre base de données. Cependant, il est important de respecter un principe fondamental de Docker : un service équivaut à un conteneur. Par conséquent, si nous avons une application et sa base de données, nous devons créer une image pour l'application et une autre image pour la base de données.

Nous devrons ensuite monter les deux conteneurs et les faire communiquer entre eux, ce qui peut compliquer un peu les choses.

Avant d'entrer dans le vif du sujet, nous allons d'abord présenter l'application de démonstration que nous allons utiliser pour illustrer la communication entre conteneurs.

6.2. Présentation de l'application de démonstration

Récupérez l'application via le fichier : [networks-starting-setup.zip](#) et décompressez le contenu :

En lisant le code source dans le fichier `app.js`, vous reconnaîtrez certainement des éléments de code que nous avons déjà vu dans les chapitres précédents :

- Nous créons notre propre API REST à partir des données récupérées par l'API `swapi`, par exemple avec cette route `/movies`

Fichier : `code/networks-starting-setup/app.js`

```
app.get('/movies', async (req, res) => {
  // code
});
```

- On consomme les données de l'API `swapi` grâce à la librairie JS `Axios`.

Fichier : `code/networks-starting-setup/app.js`

```
app.get('/movies', async (req, res) => {
  try {
    const response = await axios.get('https://swapi.dev/api/films');
    res.status(200).json({ movies: response.data });
  } catch (error) {
    res.status(500).json({ message: 'Something went wrong.' });
  }
});
```

- L'application permet aussi de mettre des films ou des personnages en favoris et de les stocker en base de données `MONGODB` en utilisant la librairie `mongoose`.

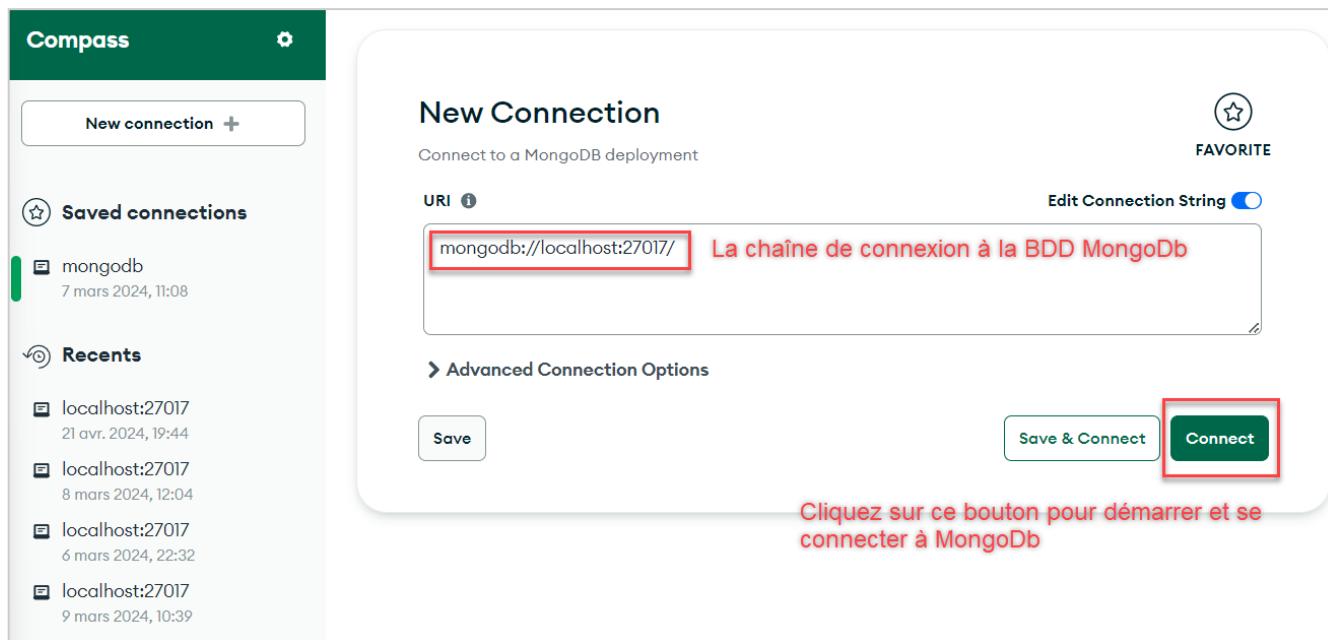
Fichier : `code/networks-starting-setup/app.js`

```
const mongoose = require('mongoose');
// .. some code
app.post('/favorites', async (req, res) => {
  // .. some code
});
```

Bien évidemment, nous pouvons tester notre application en local en installant MongoDB et son environnement, NodeJS, etc..

- Allez sur le site MongoDB et installez MongoDB Compass : <https://www.mongodb.com/products/tools/compass> Il s'agit d'une application avec une interface graphique (GUI) qui permet de créer et manipuler une base de données MongoDB. Je ne détaille pas l'installation ici.

- En lançant l'application, vous devriez voir cet écran :



- Maintenant, nous pouvons ouvrir un Terminal dans le dossier de notre application et installer les packages nécessaires :

```
PS C:\Users\baptiste\Downloads\APP> npm install
npm WARN deprecated axios@0.20.0: Critical security vulnerability fixed in https://github.com/axios/axios/pull/3410

added 100 packages, and audited 101 packages in 6s

14 packages are looking for funding
  run `npm fund` for details

1 high severity vulnerability

To address all issues (including breaking changes), run:
  npm audit fix --force

Run `npm audit` for details.
npm notice
npm notice New minor version of npm available! 10.2.5 => 10.7.0
npm notice Changelog: https://github.com/npm/cli/releases/tag/v10.7.0
npm notice Run npm install -g npm@10.7.0 to update!
npm notice
PS C:\Users\baptiste\Downloads\APP>
```

- Nous vérifions dans le code source que la chaîne de connexion à MongoDB soit la même que sur mon serveur local :

Fichier : code/networks-starting-setup/app.js

```
// ... some code
mongoose.connect(
  'mongodb://localhost:27017/swfavorites',
```

```
{ useNewUrlParser: true },  
(err) => {  
  if (err) {  
    console.log(err);  
  } else {  
    app.listen(3000);  
  }  
};
```

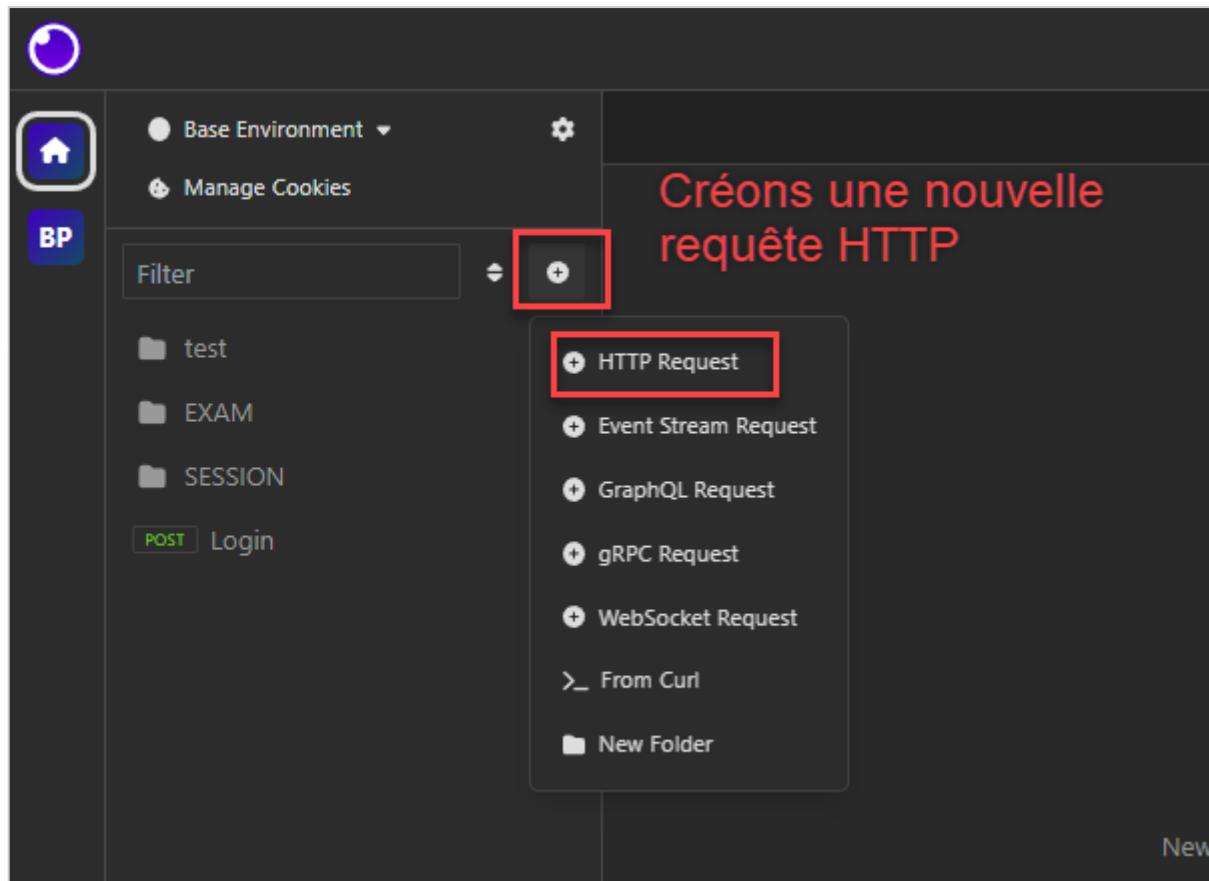
Et maintenant testons l'application :

```
node app.js
```

Comment tester notre application ? nous n'avons pas d'interface graphique (appelée aussi *UI*)!

Notre application est une API ! Il faudra alors installer un logiciel qui nous permettra d'envoyer des requêtes HTTP facilement. Personnellement, j'utilise **Insomnia**, mais il en existe d'autres comme **PostMan**.

- Allez sur : <https://insomnia.rest/> et téléchargez l'application
- Une fois Insomnia lancé :



Nous allons faire une requête de type GET en localhost, sur le port 3000 (c'est le port d'écoute que l'on retrouve dans le code source).

Et nous allons suivre la route : "movies"

Votre requête qui porte le nom par défaut : "New Request"

Après avoir cliqué sur le bouton "Send" ... la requête HTTP GET est envoyée à notre application, qui retourne une réponse captée par Insomnia .

Nous obtenons donc la liste des films Starwars et d'autres données.

```

 1 - {
 2 -   "movies": [
 3 -     {
 4 -       "count": 6,
 5 -       "next": null,
 6 -       "previous": null,
 7 -       "results": [
 8 -         {
 9 -           "title": "A New Hope",
10 -           "episode_id": 4,
11 -           "director": "George Lucas",
12 -           "producer": "Gary Kurtz, Rick McCallum",
13 -           "release_date": "1977-05-25",
14 -           "characters": [
15 -             "https://swapi.dev/api/people/1/",
16 -             "https://swapi.dev/api/people/2/",
17 -             "https://swapi.dev/api/people/3/",
18 -             "https://swapi.dev/api/people/4/",
19 -             "https://swapi.dev/api/people/5/",
20 -             "https://swapi.dev/api/people/6/"
21 -           ],
22 -           "planets": [
23 -             "https://swapi.dev/api/planets/1/",
24 -             "https://swapi.dev/api/planets/2/",
25 -             "https://swapi.dev/api/planets/3/"
26 -           ],
27 -           "starships": [
28 -             "https://swapi.dev/api/starships/1/",
29 -             "https://swapi.dev/api/starships/2/",
30 -             "https://swapi.dev/api/starships/3/",
31 -             "https://swapi.dev/api/starships/4/",
32 -             "https://swapi.dev/api/starships/5/"
33 -           ]
34 -         }
35 -       ],
36 -       "planets": [
37 -         "https://swapi.dev/api/planets/1/",
38 -         "https://swapi.dev/api/planets/2/",
39 -         "https://swapi.dev/api/planets/3/"
40 -       ],
41 -       "starships": [
42 -         "https://swapi.dev/api/starships/1/",
43 -         "https://swapi.dev/api/starships/2/",
44 -         "https://swapi.dev/api/starships/3/",
45 -         "https://swapi.dev/api/starships/4/",
46 -         "https://swapi.dev/api/starships/5/"
47 -       ]
48 -     }
49 -   ]
50 - }

```

Vous pouvez tester d'autres routes que l'on retrouve dans le code source. Je vous laisse découvrir les choses.

Nous avons une meilleure compréhension de l'application de démonstration, et nous voyons qu'elle est beaucoup plus complexe que ce que nous avons utilisé jusqu'à présent.

Nous allons maintenant conteneuriser cette application !



En créant plusieurs conteneurs afin de respecter le principe :

- 1 service = 1 conteneur

6.3. Communication

6.3.1. Préambule

Après avoir étudié le fonctionnement de notre application, nous allons examiner les cas de figure où une **communication** sera initiée entre le conteneur et d'autres services.

Il y a 3 situations que l'on peut répertorier :

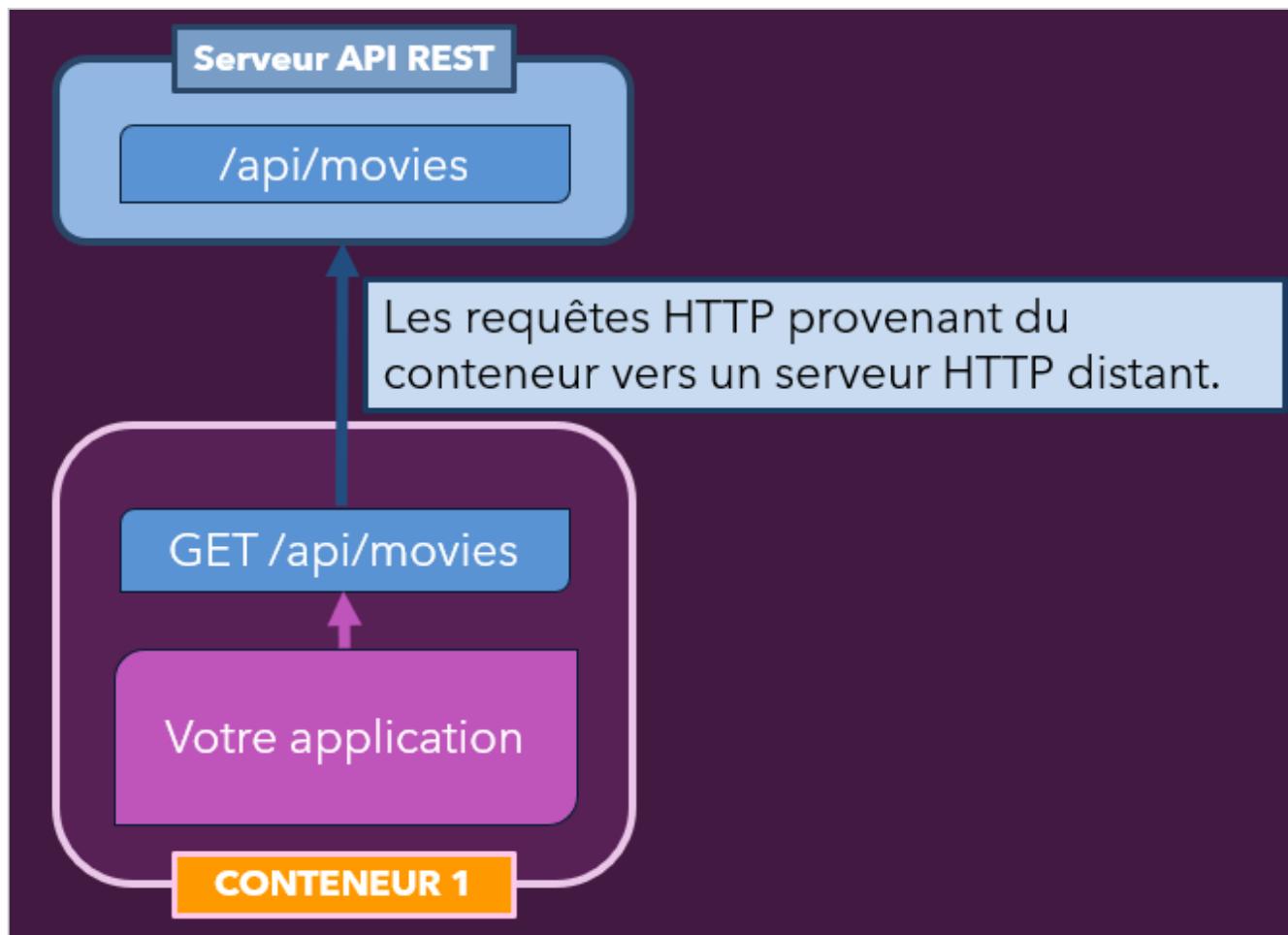
1. Du **conteneur** vers un **serveur distant** (*par exemple via HTTP vers une API*)
2. Du **conteneur** vers la **machine hôte locale** (*par exemple pour accéder à une base de données*)
3. Entre deux conteneurs

Mais avant de continuer, assurez-vous d'avoir fermé :



- Le serveur Node (**CTRL + C** dans le terminal)
- La fenêtre de MongoDB Compass.

6.3.2. Situation 1 : Du conteneur vers un serveur distant (HTTP)



L'API REST que nous interrogeons, n'est pas hébergée dans le conteneur et je ne l'ai pas non plus moi-même créée. C'est un service qui existe et que je désire **consommer** depuis mon conteneur.

Notre application va créer de nouvelles routes comme `/movies` ou `/people` par exemple, qui vont interroger l'API distante et retourner les données reçues.

En d'autres termes, notre application va devenir un proxy pour l'API distante. Elle va récupérer les données de l'API distante et les retourner à l'utilisateur.

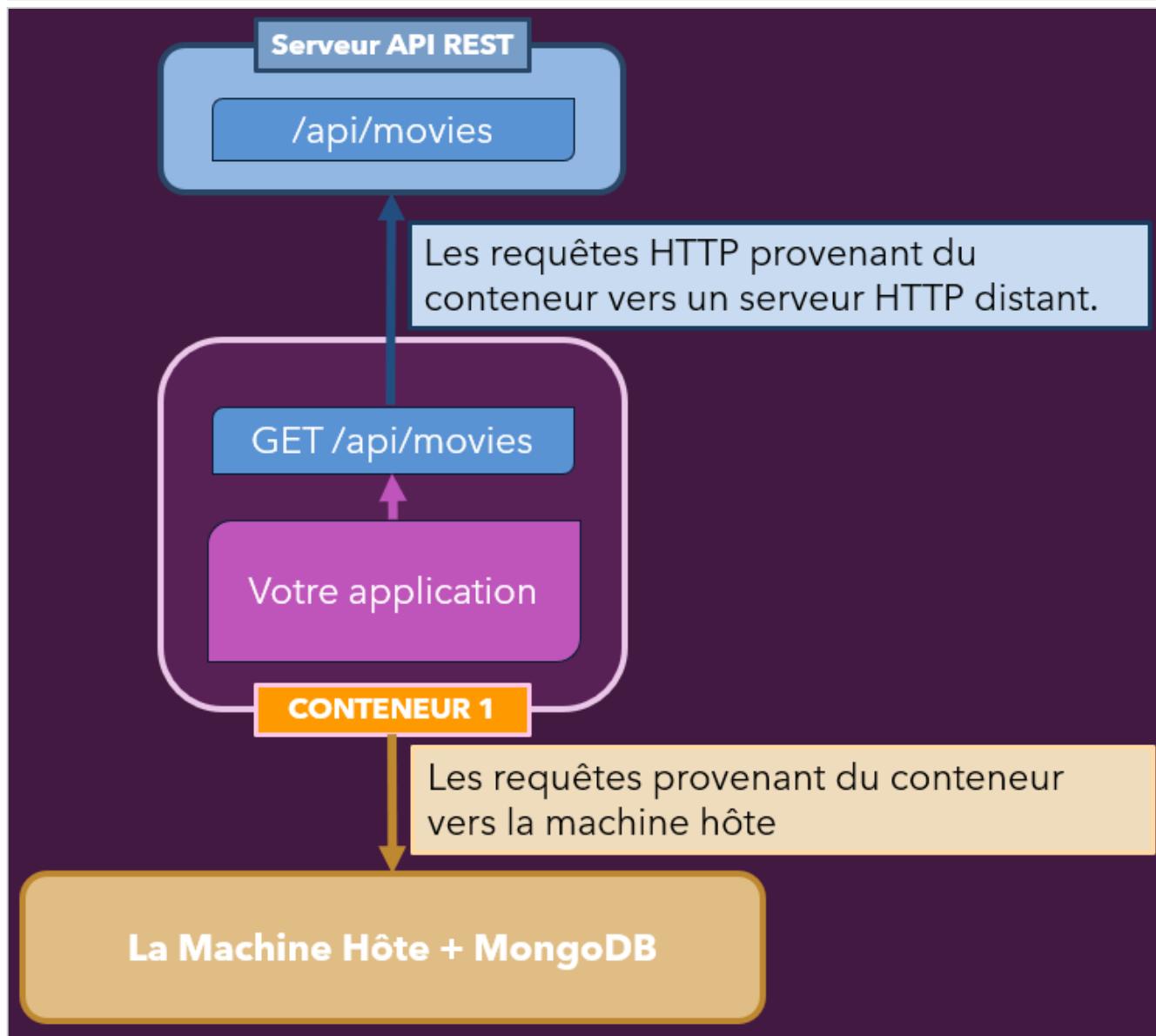
Il y a une communication **HTTP** entre notre application et l'API distante.



Et comme l'application est hébergée dans un conteneur, il faudra **s'assurer que la requête HTTP puisse sortir du conteneur et atteindre l'API distante**, puis, que la réponse puisse revenir à l'intérieur de notre conteneur.

6.3.3. Situation 2 : Du conteneur vers la machine hôte locale

Notre application doit pouvoir communiquer avec MongoDB installé sur la machine hôte pour effectuer des requêtes CRUD sur la base de données(Create, Read, Update, Delete)



```
mongoose.connect(  
  'mongodb://localhost:27017/swfavorites',  
  { useNewUrlParser: true },  
  (err) => {  
    if (err) {  
      console.log(err);  
    } else {  
      app.listen(3000);  
    }  
  }  
);
```

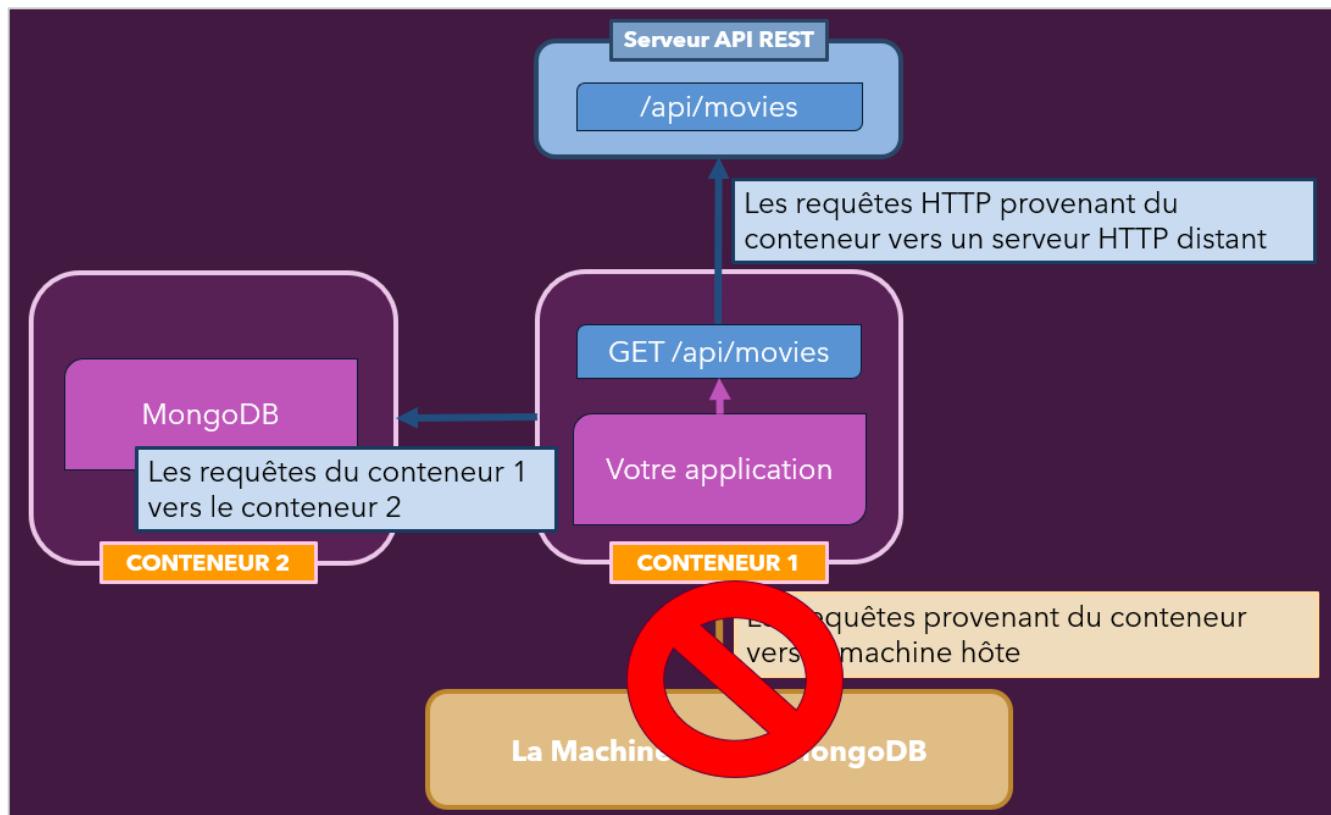
6.3.4. Situation 3 : Du conteneur vers un autre conteneur

Dans la situation 2, nous avons identifié le besoin de communiquer avec MongoDB installé sur la machine hôte.

Mais il est possible que le service de base de données soit lui aussi inclus dans un autre conteneur.

Et c'est d'ailleurs fortement recommandé !

Par conséquent, il faudra que le conteneur de notre application puisse dialoguer avec le conteneur de la base de données.



Donc, nous devons créer une image pour l'application NodeJs et une autre image pour la base de données MongoDB.

6.4. Mise en pratique : Crédit du conteneur et communication avec le serveur API REST

Dans le dossier de l'application de démonstration, supprimer le fichier `node_module` et `package-lock.json` si vous avez réalisé l'installation de l'application vue précédemment.

Nous allons maintenant créer une image avec le **Dockerfile** présent :

```
docker build -t favorites-node .
```

Ensuite lançons un conteneur basé sur l'image `favorites-node` :

```
docker container run --name favorites -d --rm -p 3000:3000 favorites-node
```

Rappels



- `--name` Permet de donner un nom au conteneur

- **-d** Démarrer le conteneur en tâche de fond
- **--rm** Supprime le conteneur dès qu'il est arrêté
- **-p** Associe le port 3000 de la machine hôte avec le port 3000 du conteneur
- **favorites-node** nom de l'image

Il n'est pas nécessaire de définir de **volumes**, car l'application n'écrit rien dans des fichiers qui doivent persister après la suppression du conteneur. Les seules données qui seront écrites sont celles qui seront stockées dans la base de données, mais elles ne seront pas stockées dans ce conteneur.

En exécutant la commande, nous recevons un identifiant de conteneur. Vérifions :

```
docker container ps  
docker container ps -a
```

Nous remarquons que la liste est vide ! Aucun conteneur n'a été démarré ni créé après la commande précédente !

Cela est tout à fait normal ! En effet, nous avons démarré le conteneur avec le paramètre **--rm**, ce qui signifie que le conteneur n'a pas pu démarrer correctement, qu'il s'est arrêté et qu'il a été supprimé.

Regardons plus en détail d'où peut provenir ce problème, en lançant de nouveau la commande sans le mode détaché **-d**.

```
docker container run --name favorites --rm -p 3000:3000 favorites-node
```

Le conteneur se lance bien, mais l'application plante !

```
PS C:\Users\baptiste\Downloads\APP> docker container run --name favorites --rm -p 3000:3000 favorites-node  
(node:1) [MONGODB DRIVER] Warning: Current Server Discovery and Monitoring engine is deprecated, and will be removed in  
a future version. To use the new Server Discover and Monitoring engine, pass option { useUnifiedTopology: true } to the  
MongoClient constructor.  
(Use 'node --trace-warnings ...' to show where the warning was created)  
MongoNetworkError: failed to connect to server [localhost:27017] on first connect [Error: connect ECONNREFUSED 127.0.0.1:27017  
at TCPConnectWrap.afterConnect [as oncomplete] (node:net:1606:16)  
name: 'MongoNetworkError'  
}  
at Pool.<anonymous> (/app/node_modules/mongodb/lib/core/topologies/server.js:441:11)  
at Pool.emit (node:events:520:28)  
at /app/node_modules/mongodb/lib/core/connection/pool.js:564:14  
at /app/node_modules/mongodb/lib/core/connection/pool.js:1000:11  
at /app/node_modules/mongodb/lib/core/connection/connect.js:32:7  
at callback (/app/node_modules/mongodb/lib/core/connection/connect.js:300:5)  
at Socket.<anonymous> (/app/node_modules/mongodb/lib/core/connection/connect.js:330:7)  
at Object.onceWrapper (node:events:635:26)  
at Socket.emit (node:events:520:28)  
at emitErrorNT (node:internal/streams/destroy:170:8)  
at emitErrorCloseNT (node:internal/streams/destroy:129:3)  
at process.processTicksAndRejections (node:internal/process/task_queues:82:21)
```

Il s'agit d'une erreur avec **MongoDb** ! Notre application tente de se connecter au serveur de base de données qui n'est pas inclus dans l'image.

Je vais lancer alors MongoCompass sur ma machine hôte et relancer le conteneur.

J'obtiens les mêmes messages d'erreurs !

Fichier : code/networks-starting-setup/app.js

```
mongoose.connect(  
  'mongodb://localhost:27017/swfavorites',  
  { useNewUrlParser: true },  
  (err) => {  
    if (err) {  
      console.log(err);  
    } else {  
      app.listen(3000);  
    }  
  }  
);
```

L'application, qui tourne dans le conteneur, tente de se connecter au serveur Mongo : `mongodb://localhost:27017/`. Etant cloisonnée dans le cadre du conteneur, `localhost` fait référence, dans l'application, au `localhost` du conteneur.

Pour le moment donc, notre conteneur est incapable de joindre le `localhost` et le port `27017` de la machine hôte !

Ouvrez donc le fichier `app.js` :

- Commentez les lignes 70 à 80. En Javascript, on met en commentaire un bloc de code avec le symbole `/*` placé en début et `*/` placé à la fin du bloc.
- Ajoutez l'écoute de l'application sur le port 3000.

Fichier : code/networks-starting-setup/app.js

```
app.listen(3000);  
/*  
mongoose.connect(  
  'mongodb://localhost:27017/swfavorites',  
  { useNewUrlParser: true },  
  (err) => {  
    if (err) {  
      console.log(err);  
    } else {  
      app.listen(3000);  
    }  
  }  
);  
*/
```

```
70  app.listen(3000);
71  /*
72   mongoose.connect(
73     'mongodb://localhost:27017/swfavorites',
74     { useNewUrlParser: true },
75     (err) => {
76       if (err) {
77         console.log(err);
78       } else {
79         app.listen(3000);
80       }
81     }
82   );
83 */
```

Reconstruisons l'image :

```
docker build -t favorites-node .
```

Créons de nouveau le conteneur :

```
docker container run --name favorites -d --rm -p 3000:3000 favorites-node
```

Vérifions :

```
docker container ps
```

Résultat :

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
6212cb109886	favorites-node	"docker-entrypoint.s..."	57 seconds ago	Up 56 seconds	0.0.0.0:3000->3000/tcp	favorites

L'application est fonctionnelle et testable ! Sauf bien entendu les deux routes qui nécessitent MongoDb :

- GET /favorites
- POST /favorites

Mais les routes appelant seulement l'API externe sont testables :

- GET /movies
- GET /people

Ouvrons le logiciel **Insomnia** et testons la méthode GET sur l'URL :

<http://localhost:3000/movies>

```

GET ▾ http://localhost:3000/movies
Send ▾ 200 OK 441 ms 18.2 KB Just Now
Body Auth Query Headers Docs
Preview ▾ Headers 7 Cookies Timeline
1: {
2:   "movies": [
3:     {
4:       "count": 6,
5:       "next": null,
6:       "previous": null,
7:       "results": [
8:         {
9:           "episode_id": "I",
10:          "opening_crawl": "It is a period of civil war.\nRebel spaceships, striking\nfrom a hidden base, have\nwon their first victory against\nthe evil Galactic Empire.\nDuring the battle, Rebel\nspies managed to steal secret\ncode plans to the Empire's\nultimate weapon, the DEATH\nSTAR, an armored space\nstation with enough\npower to destroy an entire planet.\nPursued by the Empire's\nminister agents, Princess\nLeia races home\naboard her\nstarship, custodian of the\nstolen plans that can save her\npeople and restore\nfreedom to the\ngalaxy...",
11:          "director": "George Lucas",

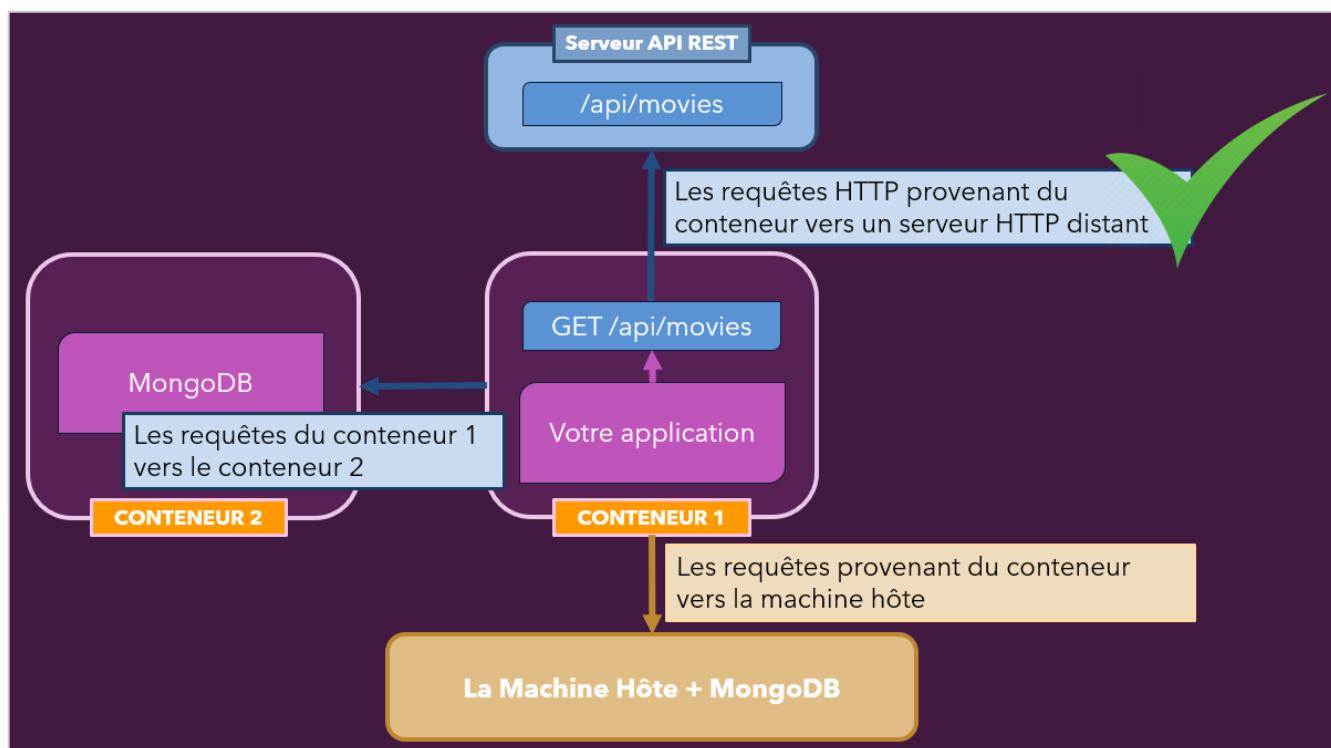
```

Nous constatons que notre conteneur communique parfaitement avec le **serveur HTTP distant** !



Il n'y a pas besoin de faire des modifications dans le code de l'application pour faire communiquer un **conteneur** avec une **API distante** par exemple !

A contrario, il faudra sûrement faire quelques modifications pour permettre **au conteneur de communiquer** avec le **localhost** de la machine hôte.



6.5. Mise en pratique : Faire communiquer le conteneur avec le localhost

Nous allons faire communiquer le service MongodB qui tourne sur la machine hôte avec le conteneur.

Pour cela supprimons les modifications apportées précédemment dans le fichier **app.js** :

Fichier : code/networks-starting-setup/app.js

```
mongoose.connect(  
  'mongodb://localhost:27017/swfavorites',  
  { useNewUrlParser: true },  
  (err) => {  
    if (err) {  
      console.log(err);  
    } else {  
      app.listen(3000);  
    }  
  }  
);
```

La ligne qui pose un problème est la suivante :

'mongodb://localhost:27017/swfavorites' Plus précisément, le nom de domaine `localhost`.

Comme nous l'avons déjà mentionné, lorsque `localhost` est lu par l'application et par le moteur de traduction d'adresse IP de Docker, il est compris comme faisant référence à l'adresse IP du conteneur.

Or, nous voulons qu'il soit interprété différemment ! Nous voulons que `localhost` pointe vers l'adresse IP de notre machine hôte.

Heureusement, il existe une instruction spéciale comprise par Docker qui permet de remplacer le terme `localhost` dans notre code par `host.docker.internal`, ce qui permettra de cibler l'adresse IP de la machine hôte.

Fichier : code/networks-starting-setup/app.js

```
mongoose.connect(  
  'mongodb://host.docker.internal:27017/swfavorites',  
  { useNewUrlParser: true },  
  (err) => {  
    if (err) {  
      console.log(err);  
    } else {  
      app.listen(3000);  
    }  
  }  
);
```

Nous devons maintenant reconstruire l'image encore une fois pour que les modifications soient prises en compte.

Reconstruisons l'image :

```
docker build -t favorites-node .
```

Créons de nouveau le conteneur :

```
docker container run --name favorites -d --rm -p 3000:3000 favorites-node
```

Nous allons maintenant tester en postant un favori :

- Ouvrez **Insomnia** :
- Modifiez la requête HTTP en POST
- Ajouter l'URL : <http://localhost:3000/favorites>
- Modifiez le type du BODY de la requête en **JSON**
- ajoutez la structure **JSON** suivante :

```
{  
  "type" : "movie",  
  "name" : "The Empire Strike Back",  
  "url" : "http://swapi.dev/api/films/2/"  
}
```

- Appuyez sur "Send"

The screenshot shows the Insomnia interface. On the left, a POST request is made to <http://localhost:3000/favorites>. The request body is a JSON object:

```
{  
  "type" : "movie",  
  "name" : "The Empire Strike Back",  
  "url" : "http://swapi.dev/api/films/2/"  
}
```

The response on the right is a 201 Created status with a message: "Favorite saved!". The response body is:

```
{  
  "message": "Favorite saved!",  
  "favorite": {  
    "_id": "663bb8bafefb74d98b4ccdd15",  
    "name": "The Empire Strike Back",  
    "type": "movie",  
    "url": "http://swapi.dev/api/films/2/",  
    "__v": 8  
  }  
}
```

A green box highlights the message "Réponse du serveur!".

Cela fonctionne maintenant parfaitement !

Nous pouvons même récupérer le favori ajouté :

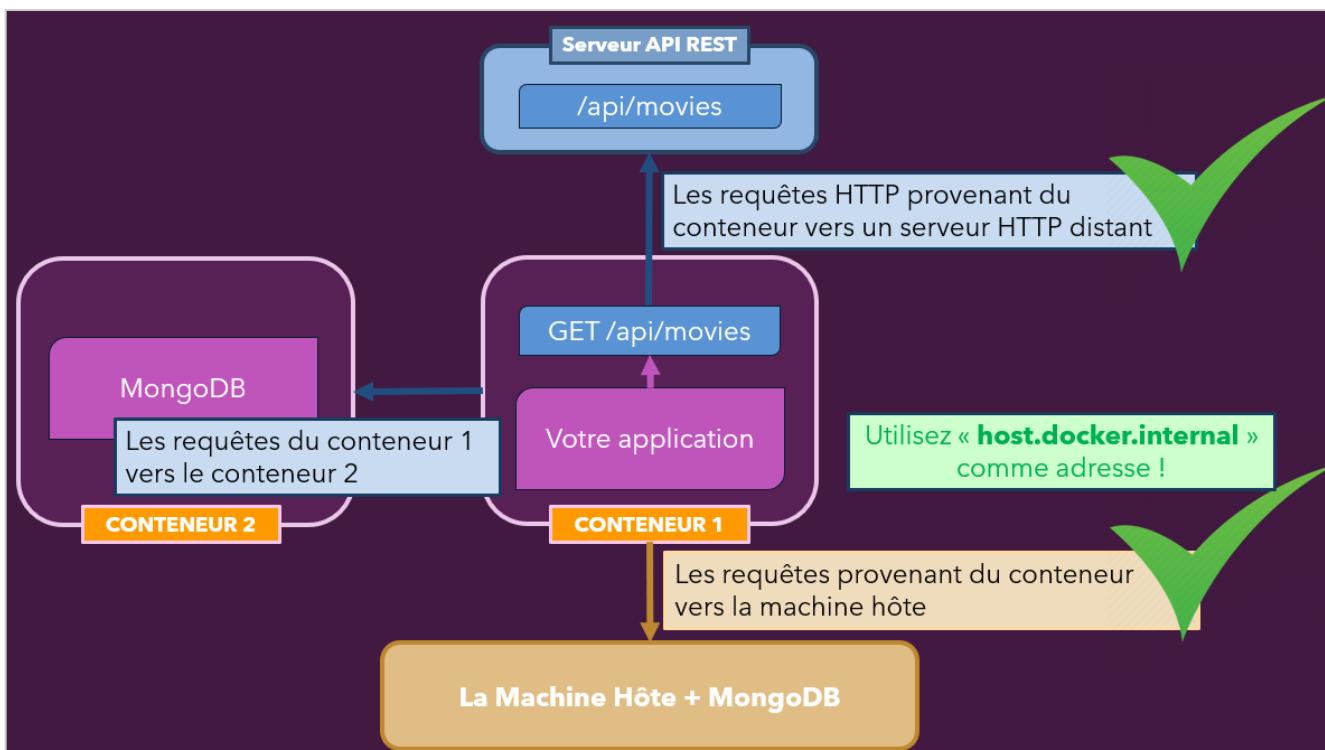
- Modifiez la requête HTTP en GET
- Laissez l'URL : <http://localhost:3000/favorites>
- Modifiez le type du BODY en "No Body"
- Appuyez sur "Send"

```

1 < {
2   "favorites": [
3     {
4       "_id": "663b8b5abfeb74d98b4cccd15",
5       "name": "The Empire Strike Back",
6       "type": "movie",
7       "url": "http://swapi.dev/api/films/2/",
8       "__v": 0
9     }
10   ]
11 }

```

Nous avons réussi à faire communiquer notre conteneur avec la machine hôte ! Mais il faut penser à modifier légèrement notre code !



Toutefois, cela n'est pas la solution idéale. Dans notre situation, il faudra créer un conteneur avec MongoDB et s'arranger pour qu'il puisse communiquer avec le conteneur de l'application !

6.6. Mise en pratique : Communication entre conteneurs [Solution basique]

Premièrement, nous créerons un nouveau conteneur basé sur une image officielle de MongoDB :

```
docker container run -d --name mongodb mongo
```

Maintenant que le conteneur est lancé, comment faire communiquer notre application du conteneur **favorites** avec MongoDB présent dans le conteneur "mongodb" ?

Nous savons déjà qu'il faudra éditer le fichier **app.js** et la ligne contenant cette chaîne de connexion :

```
mongodb://host.docker.internal:27017/swfavorites,
```

`host.docker.internal` peut être remplacé par l'adresse IP de `mongodb`, que l'on peut connaître en inspectant la configuration du conteneur :

```
docker container inspect mongodb
```

La commande retourne un fichier JSON organisant les données du conteneur.

Nous n'avons plus qu'à chercher la clef `IPAddress` !

Le fichier de configuration est long à parcourir. Nous allons modifier un peu la commande et intégrer un filtre.

`IPAddress` se trouve dans la catégorie : `NetworkSettings`, cela est pratique de le savoir, car nous allons pouvoir formater la réponse en n'affichant que la clé qui nous intéresse.

```
docker inspect --format '{{ .NetworkSettings IPAddress }}' mongodb
```

L'adresse IP renvoyée est : **172.17.0.3**

```
PS C:\Users\baptiste\Downloads\APP> docker inspect --format '{{ .NetworkSettings IPAddress }}' mongodb
172.17.0.3
```

Modifions maintenant notre fichier `app.js` de la sorte :

```
// some code before
mongoose.connect(
  'mongodb://172.17.0.3:27017/swfavorites',
  { useNewUrlParser: true },
  (err) => {
    if (err) {
      console.log(err);
    } else {
      app.listen(3000);
    }
  }
);
```

Pour tester, stoppons le conteneur `favorites`, et reconstruisons l'image.

```
docker container stop favorites
```

Puis :

```
docker build -t favorites-node .
```

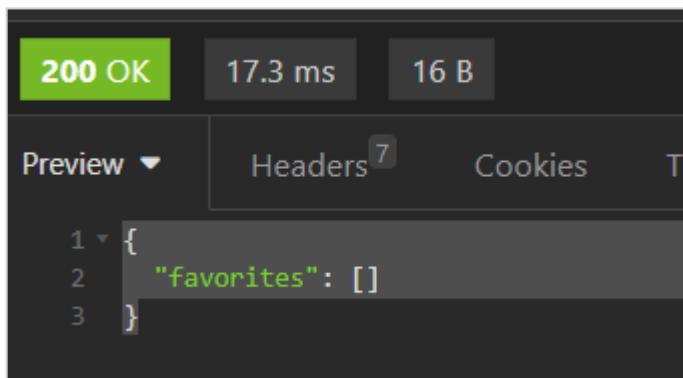
Lançons cette nouvelle version :

```
docker container run --name favorites -d --rm -p 3000:3000 favorites-node
```

Vérifions si nos deux conteneurs sont bien lancés :

```
docker ps
```

Et maintenant avec Insomnia, testons :



Le résultat est correcte puisque nous avons une nouvelle installation de MongoDB et nous n'avons pas encore inséré de données.

Nos deux conteneurs communiquent et échangent des informations ! Nous sommes arrivés à nos fins.

Mais cette solution est tout sauf pratique !

- Il faut chercher l'adresse IP du conteneur soit même.
- Quand l'adresse IP de MongoDB change, il faut recréer une image de l'application.

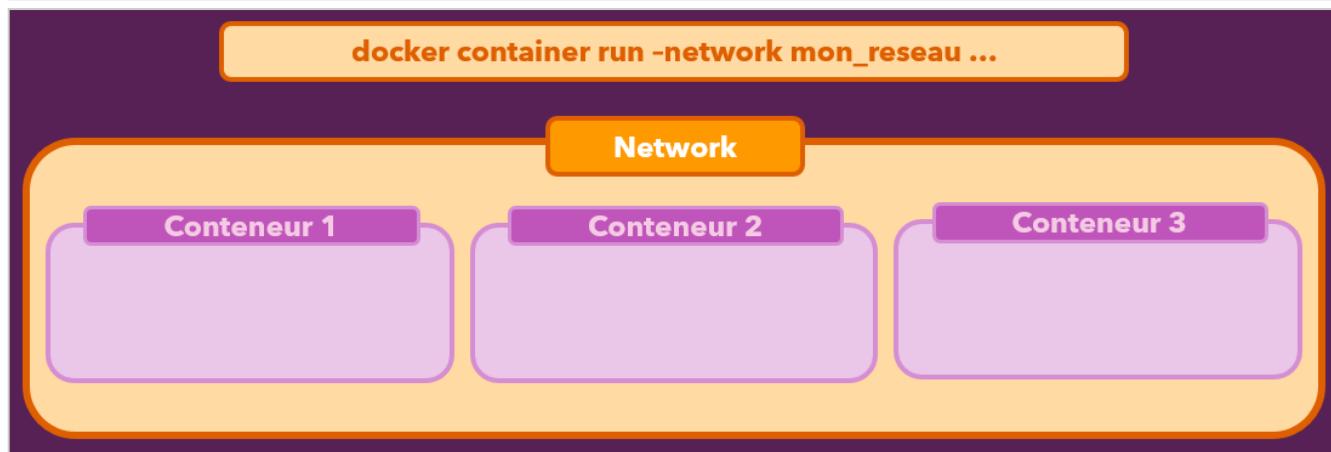
Voyons maintenant une manière élégante de procéder !

6.7. Mise en pratique : Les Networks Docker [Solution élégante]

Avec Docker, nous pouvons créer des réseaux appelés en anglais : [Networks](#).

Lorsque nous avons plusieurs conteneurs qui vont devoir communiquer, nous allons pouvoir les réunir au sein d'un même réseau grâce au paramètre [--network](#) .

Docker ne crée pas automatiquement les réseaux quand nous montons plusieurs conteneurs. Il faut le spécifier manuellement.



Dans un **réseau interne Docker**, tous les conteneurs peuvent communiquer. Les adresses IP sont automatiquement résolues, c'est-à-dire **qu'elles sont attribuées et connues de tous les membres du réseau**.

Mettons en place un réseau pour les conteneurs de notre application.

Stoppons d'abord les conteneurs :

```
docker container stop favorites
```

puis

```
docker container stop mongodb
```

Et supprimons tous les conteneurs arrêtés :

```
docker container prune
```

Maintenant voyons comment nous allons pouvoir intégrer notre premier conteneur **mongodb** dans un réseau nommé arbitrairement **fav-network** :

Il faut créer le réseau en lui-même avec la commande :

```
docker network create fav-network
```

Par curiosité, nous pouvons lister l'ensemble des réseaux existant sur notre Docker :

```
docker network ls
```

NETWORK ID	NAME	DRIVER	SCOPE
10c90d5259f0	bridge	bridge	local
7f9ad61f1dc4	fav-network	bridge	local
833e37b2381c	host	host	local
1c3b41609d6b	none	null	local

Ensuite, créons un autre conteneur, en l'intégrant dans le nouveau réseau :

```
docker container run -d --name mongodb --network fav-network mongo
```

Mongodb est inclus dans un réseau Docker et sera utilisé par un autre conteneur. Vous remarquerez qu'il n'y a pas besoin ici d'exposer un port de mongodb vers une quelconque sortie.



Comme Mongodb sera dans le même réseau que notre application, le lien via l'adresse IP et le PORT du service sera automatiquement réalisé par DOCKER !

La prochaine étape sera de monter le conteneur de notre application de la même manière ! Mais il faut modifier le code source pour permettre la communication entre notre application et le serveur mongodb.

Il faut trouver une solution pour récupérer l'adresse IP du conteneur **mongodb** automatiquement.

Comme les conteneurs font partie du **même réseau Docker**, nous pouvons remplacer l'adresse IP du conteneur par le nom de ce conteneur: **mongodb** dans notre cas.

```
// some code before
mongoose.connect(
  'mongodb://mongodb:27017/swavorites',
  { useNewUrlParser: true },
  (err) => {
    if (err) {
      console.log(err);
    } else {
      app.listen(3000);
    }
  }
);
```

Docker va alors automatiquement traduire ce nom de conteneur par la valeur de son adresse IP.

Faisons les modifications dans notre code, et montons une nouvelle image :



```
mongoose.connect(  
  'mongodb://mongodb:27017/swavorites',  
  { useNewUrlParser: true },  
  (err) => {  
    if (err) {  
      console.log(err);  
    } else {  
      app.listen(3000);  
    }  
  }  
);
```

```
docker build -t favorites-node .
```

Lançons cette nouvelle version :

```
docker container run --name favorites --network fav-network -d --rm -p 3000:3000  
favorites-node
```

Nous pouvons vérifier si le processus s'est bien déroulé :

```
docker ps
```

La commande devrait montrer les deux conteneurs démarrés.

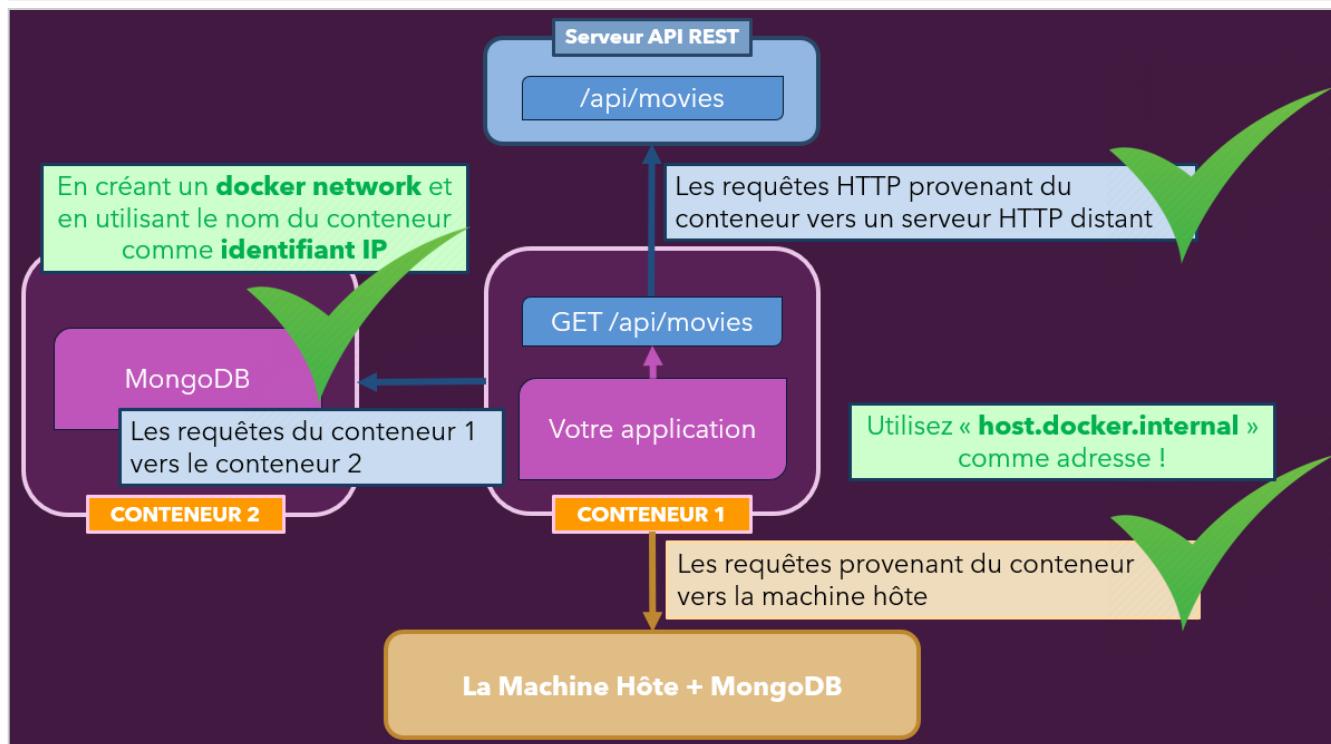
Puis les **logs**, qui nous indiqueront si des problèmes de liaison entre les conteneurs du réseau seraient apparus.

```
docker logs
```

Et au final, avec Insomnia, nous pouvons tester les routes !

En conclusion :

- Dans une communication de conteneur à conteneur, il faut créer un réseau, puis le **Docker Engine** se charge de faire la liaison automatiquement **sans que l'on ait à se soucier de l'exposition des ports**.
- L'**exposition des ports** n'est à réaliser que dans le cadre d'une communication entre un conteneur et la machine hôte.



Docker ne modifie pas le code source de notre application en remplaçant le nom de domaine par l'adresse IP du conteneur. Cependant, Docker contrôle l'environnement réseau et reçoit les requêtes HTTP entrantes et sortantes des conteneurs. Lorsqu'un conteneur envoie une requête au serveur mongodb, Docker tente de résoudre le nom de domaine du destinataire. Si le nom de domaine est remplacé par une adresse IP, Docker enverra la requête à cette adresse. Si le nom de domaine est un nom de conteneur, qui agit comme un nom de domaine local, Docker cherchera son adresse IP pour envoyer la requête.



6.8. Les pilotes (drivers) de Docker Network

Les réseaux Docker prennent en charge différents types de "pilotes" ou "**drivers**" en anglais, qui influencent le comportement du réseau.

Le pilote par **défaut** est le pilote "**bridge**"

- Il fournit le comportement présenté dans ce module, (c'est-à-dire que les conteneurs peuvent se trouver les uns les autres par nom s'ils se trouvent dans le même réseau).

Le pilote peut être défini lorsqu'un réseau est créé, simplement en ajoutant l'option `--driver`.

```
docker network create --driver bridge mon_reseau
```

Bien sûr, si vous souhaitez utiliser le pilote "**bridge**", vous pouvez simplement omettre l'option entière car "**bridge**" est le pilote par défaut.

Docker prend également en charge les pilotes alternatifs suivants - bien que vous utilisiez le pilote "**bridge**" dans la plupart des cas :

- **host** : pour les conteneurs autonomes, l'isolement entre le conteneur et le système hôte est supprimé (c'est-à-dire qu'ils partagent localhost en tant que réseau)
- **overlay** : plusieurs démons Docker (c'est-à-dire Docker en cours d'exécution sur différentes machines) peuvent se connecter les uns aux autres. Fonctionne uniquement en mode "Swarm" qui est un moyen obsolète / presque obsolète de connecter plusieurs conteneurs
- **macvlan** : vous pouvez définir une adresse MAC personnalisée pour un conteneur - cette adresse peut ensuite être utilisée pour la communication avec ce conteneur
- **none** : toute la mise en réseau est désactivée.
- **Third-party plugins** : vous pouvez installer des plugins tiers qui peuvent alors ajouter toutes sortes de comportements et de fonctionnalités

Comme mentionné, le pilote "**bridge**" a le plus de sens dans la grande majorité des scénarios.

7. Construire des applications multi-conteneurs avec Docker

Version 1.0.0 | Dernière mise à jour : 17/11/2024 | Auteur : Bauer Baptiste

coming soon ...



8. Docker Compose : Applications Multi-conteneurs qui ont de la classe !

Version 1.0.0 | Dernière mise à jour : 17/11/2024 | Auteur : Bauer Baptiste

coming soon ...



9. Travailler avec des conteneurs utilitaires

Version 1.0.0 | Dernière mise à jour : 17/11/2024 | Auteur : Bauer Baptiste

coming soon ...

10. Projet Laravel complexe et Docker

Version 1.0.0 | Dernière mise à jour : 17/11/2024 | Auteur : Bauer Baptiste

Notes de version

- 1.1.0 du 22/11/2024 à 14:58 **
- 1.0.0 du 22/11/2024 à 14:58
 - Version initiale

10.1. Introduction

Dans le chapitre précédent, nous avons appris à utiliser des conteneurs utilitaires, et plus généralement, tout au long de ce cours, nous avons exploré **Docker**, les conteneurs **Docker**, ainsi que **Docker Compose**. Nous avons également examiné comment construire des applications composées de plusieurs conteneurs.

Dans ce chapitre, nous allons approfondir ces concepts et apprendre quelques nouveaux aspects et façons d'utiliser **Docker Compose**. Nous allons pratiquer cela avec un projet **Laravel PHP**, en configurant un projet **Laravel PHP** sur notre machine locale avec **Docker**, de manière à ne pas avoir besoin d'installer d'autres outils sur notre machine, à part **Docker**, pour créer des applications fiables.

En faisant cela, nous allons appliquer ce que nous avons appris sur les **images**, les **conteneurs**, **Docker Compose**, les conteneurs utilitaires, et bien plus encore, sur un exemple concret.



Vous n'aurez pas besoin de connaissances en **Laravel** ou en **PHP** pour ce module.

Nous n'écrirons pas de code **Laravel** ou **PHP**, et c'est pourquoi je recommande également de ne pas passer ce chapitre, même si **Laravel** ne vous intéresse pas.

Nous allons découvrir de nouvelles fonctionnalités tout au long de ce module.

Nous allons apprendre de nouvelles manières d'utiliser **Docker Compose**, d'interagir avec plusieurs fichiers **Docker**, de connecter des **images**, et bien plus encore. J'ai choisi **Laravel** et **PHP** comme exemple car :

- Premièrement, nous avons déjà travaillé avec beaucoup d'exemples basés sur **Node.js**, et je souhaite vraiment souligner que **Docker** peut être utilisé pour n'importe quelle technologie, et surtout pour n'importe quelle technologie web.
- Deuxièmement, **Laravel** et **PHP** nécessitent une configuration plus complexe sur votre machine. Alors que pour **Node.js**, il vous suffit de télécharger et d'installer un seul outil, pour **Laravel** et **PHP**, cela demande bien plus de configuration, comme nous le verrons dans la prochaine section.

C'est pourquoi nous allons maintenant explorer cette configuration et voir comment nous pouvons construire un environnement de développement pour une application **Laravel PHP** à partir de

zéro avec **Docker**. En cours de route, comme mentionné, nous allons également découvrir de nombreux nouveaux aspects intéressants et importants.

10.2. La configuration cible

The screenshot shows the official Laravel website. At the top, there's a red header bar with the text "Join the waitlist for [Laravel Cloud](#), the future of shipping.". Below the header is the Laravel logo and a navigation menu with links to Forge, Vapor, Ecosystem (with a dropdown arrow), News, Partners, and Careers. To the right of the menu is a search icon and a red button labeled "DOCUMENTATION". The main content area features the text "The PHP Framework for Web Artisans" in large, bold, black and red letters. Above the text are two small red cube icons. Below the title is a short description: "Laravel is a web application framework with expressive, elegant syntax. We've already laid the foundation — freeing you to create without sweating the small things." At the bottom of the main section are two red buttons: "GET STARTED" and "WATCH LARACASTS".

Qu'est-ce qui rend **Laravel** et **PHP** si spéciaux ?

Laravel est le framework **PHP** le plus populaire. Il est très agréable à utiliser. Cependant, configurer notre machine locale pour le développement avec **Laravel** peut être assez fastidieux. Ce n'est pas vraiment la faute de **Laravel**, mais plutôt à cause de la configuration de **PHP**, qui peut être un peu compliquée.

Si nous consultons la documentation de **Laravel**, nous découvrons les exigences du serveur et les différents éléments à installer sur notre système. Il y a pas mal de choses à installer, à commencer par **PHP**, ce qui peut déjà poser un problème.

Installer **PHP** est faisable, mais contrairement à **NodeJS**, cela ne suffit pas. Le grand avantage de **NodeJS** est qu'il ne s'agit pas seulement du langage dans lequel nous écrivons notre code (le langage est **JavaScript**), mais aussi d'un environnement d'exécution **JavaScript**. Avec **NodeJS**, nous pouvons écrire à la fois le code de l'application et la logique du serveur en une seule pièce, tout est géré par **NodeJS**.

Pour **Laravel** et **PHP**, c'est différent. Installer **PHP** ne suffit pas, car en plus de **PHP**, nous avons besoin d'un serveur pour traiter les requêtes entrantes et déclencher l'interpréteur **PHP** pour exécuter notre code. Configurer tout cela sur une machine locale, en plus d'une base de données **MySQL** ou **MongoDB**, peut devenir assez fastidieux.

C'est pourquoi cet exemple est parfait pour utiliser **Docker**, car vous verrez à quel point il est facile de gérer une configuration complexe sans avoir à installer quoi que ce soit sur notre machine locale, à part **Docker**. En utilisant **Docker**, nous pourrons écrire du code **Laravel PHP** et créer des applications sans avoir besoin d'installer d'autres outils.

Voici l'objectif de cette configuration que nous allons atteindre dans ce module. Nous allons construire un environnement de développement **Laravel** sans avoir à installer quoi que ce soit sur

notre machine hôte, à part **Docker**.

L'idée est d'avoir un dossier sur notre machine hôte qui contiendra le code source de cette application **Laravel PHP**. Ce dossier pourra être ouvert avec l'éditeur de notre choix, et nous pourrons y écrire du code **Laravel**.

Ce dossier de code source sera ensuite exposé à un conteneur, le **conteneur de l'interpréteur PHP**. Il s'agit d'un conteneur qui a **PHP** installé à l'intérieur. Ce conteneur aura accès à notre code source, pourra interpréter ce code et générer une réponse pour les requêtes entrantes.

En plus de cet interpréteur **PHP**, nous avons besoin d'un serveur. Contrairement à **Node**, nous ne construisons pas ce serveur avec **PHP**. Nous allons donc utiliser un second conteneur, qui contiendra **Nginx**, un serveur web. Ce conteneur prendra en charge les requêtes entrantes, les enverra à l'interpréteur **PHP**, et renverra la réponse au client.

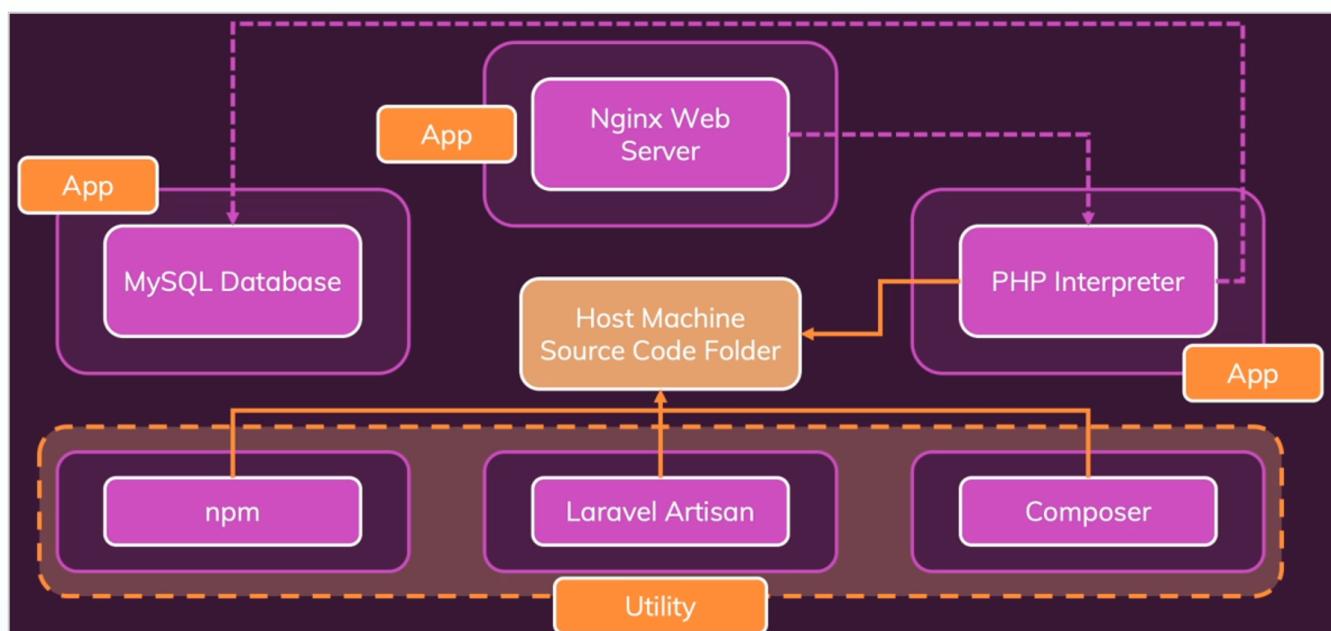
Pour stocker les données, nous allons également ajouter une base de données **MySQL**. **Laravel** peut également fonctionner avec des bases de données **MongoDB**, mais la plupart du temps, on utilise des bases de données SQL. Ainsi, notre application **Laravel** pourra communiquer avec cette base de données.

Ces trois conteneurs (**PHP**, **Nginx**, **MySQL**) sont des **conteneurs d'application**, c'est-à-dire qu'ils resteront actifs tant que notre application sera en cours d'exécution.

En plus de ces conteneurs d'application, notre configuration nécessitera quelques **conteneurs utilitaires**. En effet, les applications **Laravel** utilisent des outils spécifiques comme **Composer**, qui est l'équivalent de **NPM** pour **PHP**. **Composer** est un gestionnaire de paquets que nous utiliserons pour créer une application **Laravel** et installer les dépendances dont elle a besoin.

Laravel dispose également de son propre outil appelé **Artisan**. Cet outil permet d'exécuter des migrations de base de données, d'ajouter des données initiales, et bien plus encore.

Enfin, nous utiliserons également **NPM**, car **Laravel** peut inclure du code **JavaScript** dans ses vues, et je souhaite présenter une configuration complète que vous pourriez utiliser pour développer des applications **Laravel**.



Au total, nous aurons donc six conteneurs qui interagiront avec notre code source. C'est la configuration que nous allons mettre en place dans ce module.

10.3. Préparation

Commençons à écrire un peu de code ici.

Dans un dossier vide et je tiens à souligner que je n'ai pas installé **Composer** ni les autres outils nécessaires. Si j'essaie de l'exécuter, je reçois une erreur **command not found**. Donc ici, il me manque réellement certains outils nécessaires pour créer une application **Laravel**.

C'est pourquoi nous allons utiliser **Docker** pour cela. Je vais commencer par ajouter un fichier `docker-compose.yaml`, car nous allons créer plusieurs conteneurs et ces conteneurs devront interagir. Même s'il n'y avait qu'un seul conteneur, j'aime avoir cette configuration dans un fichier texte, facile à lire et à modifier.

Ce fichier contiendra à la fois les conteneurs d'application et les conteneurs utilitaires. Je vous montrerai également comment exécuter tous ces conteneurs ou juste certains d'entre eux tout au long de ce module.

Je vous ai montré que nous aurons six services différents.

Nous aurons notre service :

- le serveur **nginx**, qui prendra en charge toutes les requêtes entrantes et déclenchera l'interpréteur **PHP**.
- le conteneur **PHP**, responsable de l'exécution de notre code **PHP**, et donc aussi du code **Laravel**, car **Laravel** n'est qu'un framework **PHP**.
- le conteneur **MySQL**, qui contiendra la base de données **MySQL**, ainsi que les conteneurs utilitaires.
- un conteneur **Composer**,
- un conteneur **npm**,
- le conteneur **Artisan**.

Fichier : `./docker-compose.yaml`

```
services:  
  server:  
    php:  
    mysql:  
    composer:  
    artisan:  
    npm:
```

Ce sont les six conteneurs dont nous aurons besoin ici, et je vais bien sûr les ajouter étape par étape. Commençons par le serveur.

10.4. Un conteneur Nginx(Serveur Web)

10.4.1. Son Image

Le serveur, comme je l'ai mentionné, utilisera **nginx**, un serveur web très populaire, puissant et efficace. L'avantage est que si vous recherchez **docker nginx**, vous trouverez une image officielle **nginx** que vous pouvez utiliser. Tout comme avec **MongoDB** ou **Node**, nous avons également une image officielle pour cela.

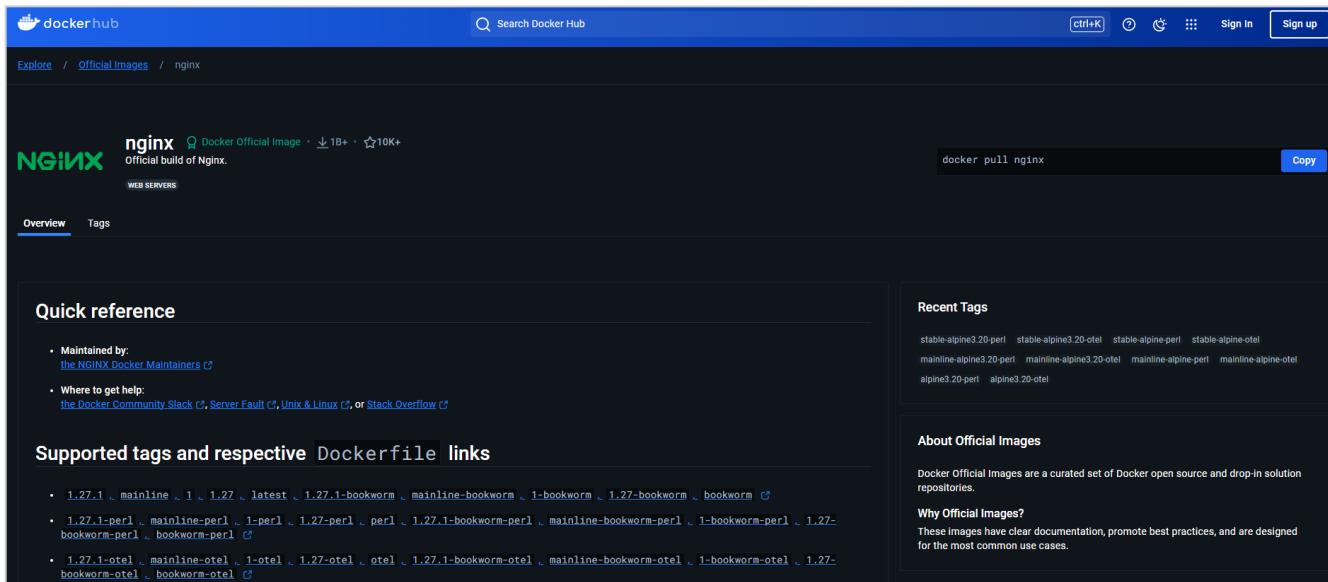


Figure 6. Page officielle de Nginx sur DockerHub

Nous pouvons simplement utiliser cette image pour configurer un serveur **nginx**. Vous trouverez également de la documentation expliquant comment utiliser cette image sur la page **Docker Hub**. Mais bien sûr, nous allons la configurer ensemble.

Dans le fichier **docker-compose**, nous pouvons maintenant spécifier une image pour le serveur et utiliser une image officielle, l'image **nginx**. Il existe plusieurs tags, plusieurs versions de cette image que nous pourrions utiliser. Je vais utiliser le tag **stable-alpine** ici pour obtenir une image basée sur une couche système d'exploitation **Linux** très légère, et une version stable de cette image **nginx**.

Fichier : ./docker-compose.yaml

```
services:
  server:
    image: 'nginx:stable-alpine'

  # php:
  # mysql:
  # composer:
  # artisan:
  # npm:
```

10.4.2. Exposition de son port

Ce serveur expose également un port, et comment le savoir ?

Eh bien, la documentation officielle nous le dit.

Elle nous indique que nous pouvons exposer le port **80**, qui est le port interne exposé par cette image. Nous allons donc lier ce port avec l'option **ports** à un port de notre machine hôte, ici le port **8000**, que nous lierons au port **80** exposé par cette image et donc par le conteneur en cours d'exécution.

Fichier : `./docker-compose.yaml`

```
services:  
  server:  
    image: 'nginx:stable-alpine'  
    ports:  
      - 8000:80  
  
    # php:  
    # mysql:  
    # composer:  
    # artisan:  
    # npm:
```

10.4.3. Ajout d'un montage Bind Mount

Nous pourrons alors exécuter un conteneur avec un serveur à l'intérieur. Mais ce serveur ne sera pas très utile par défaut, car il ne saura pas quoi faire. Ce qu'il devrait faire ici, c'est traiter les requêtes entrantes et les rediriger vers notre conteneur **PHP**, que nous ajouterons plus tard, pour que ce conteneur exécute notre code **PHP**.

Pour fournir notre propre configuration, nous ajouterons un montage (**bind mount**) avec la clé **volumes**. Ici, nous allons lier un dossier local, disons un dossier **nginx** (que nous devons encore ajouter), et y mettre un fichier **nginx.conf**.

Dans le conteneur, nous le lierons à un chemin absolu : `/etc/nginx/conf.d/default.conf`. (*Voir la documentation officielle de NGINX pour plus d'informations sur la configuration de NGINX*)

Nous pouvons également définir cela en lecture seule (**read-only**), car le conteneur ne devrait jamais modifier cette configuration. Cela nous permettra de transmettre notre fichier de configuration personnalisé pour ce serveur web dans le conteneur.

Fichier : `./docker-compose.yaml`

```
services:  
  server:  
    image: 'nginx:stable-alpine'  
    ports:  
      - 8000:80
```

```
volumes:  
- ./nginx/nginx.conf:/etc/nginx/conf.d/default.conf:ro  
  
# php:  
# mysql:  
# composer:  
# artisan:  
# npm:
```

Nous devons maintenant créer ce dossier **nginx** sur notre machine hôte, ainsi qu'un fichier **nginx.conf**. Vous trouverez le fichier de configuration prêt dans cette section, car je l'ai déjà préparé pour vous. Vous pouvez donc simplement utiliser ce fichier.

Ce fichier contient une configuration **nginx** qui écoute sur le port **80** et gère les requêtes, puis redirige celles-ci vers des fichiers **index.php**, ou vers notre interpréteur **PHP** pour traiter les requêtes entrantes.

Fichier : ./nginx/nginx.conf

```
server {  
    listen 80;  
    index index.php index.html;  
    server_name localhost;  
    root /var/www/html/public;  
    location / {  
        try_files $uri $uri/ /index.php?$query_string;  
    }  
    location ~ \.php$ {  
        try_files $uri =404;  
        fastcgi_split_path_info ^(.+\.php)(/.+)$;  
        fastcgi_pass php:9000;  
        fastcgi_index index.php;  
        include fastcgi_params;  
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;  
        fastcgi_param PATH_INFO $fastcgi_path_info;  
    }  
}
```

Je suis conscient que cela était nouveau et peut-être pas quelque chose que vous sentez à l'aise de construire vous-même, mais c'est pourquoi nous parcourons cela ensemble. Nous avons maintenant construit ce conteneur **nginx**, et dans la prochaine section, nous passerons au conteneur **PHP**.

10.5. Un conteneur PHP

10.5.1. Son Image

Passons maintenant au conteneur **PHP**, après avoir configuré le conteneur **nginx**. Le conteneur **PHP** sera relativement simple. Cependant, je vais utiliser un fichier **Dockerfile** personnalisé, car il

n'existe pas d'image prête à l'emploi avec tout ce dont j'ai besoin.

Pour clarifier, si vous recherchez **PHP** sur **Docker Hub**, vous trouverez une image officielle que nous allons utiliser. Cependant, je souhaite construire une image personnalisée à partir de cette image, car je vais ajouter des extensions supplémentaires nécessaires pour **Laravel**.

Je vais donc ajouter un nouveau dossier à côté du dossier **nginx**, que je nommerai **dockerfiles**. À l'intérieur, j'ajouterais un fichier **php.dockerfile**. Vous pouvez choisir le nom que vous voulez, mais en utilisant cette convention de nommage, cela permet à certains éditeurs comme **Visual Studio Code** de reconnaître le fichier comme un **Dockerfile**, ce qui aide pour l'auto-complétion.

Je vais commencer avec l'image de base **PHP**, que je viens de mentionner. En regardant les tags, je vais utiliser l'image **8.2.4-fpm-alpine**, qui est une image **PHP** légère et stable. Il s'agit d'une version spécifique de l'image PHP version 8.2.4, qui inclut FPM (FastCGI Process Manager) pour traiter les requêtes PHP via un serveur web comme Nginx. La partie alpine désigne une version légère basée sur Alpine Linux, ce qui en fait une image optimisée pour les environnements de production, car elle est plus petite et rapide à télécharger.

Fichier : ./dockerfiles/php.dockerfile

```
FROM php:8.2.4-fpm-alpine
```

Nous allons définir un répertoire de travail à l'intérieur du conteneur. Tout ce qui sera exécuté depuis ce répertoire.

Puis, nous allons copier le contenu du dossier **/src/** dans le répertoire de travail. Il faudra créer alors un dossier **src** à côté du dossier **dockerfiles**.

C'est dans ce dossier que nous mettrons notre code source **Laravel**.

Ensuite, je vais exécuter une commande. C'est là la raison pour laquelle je construis cette image personnalisée : je vais installer des dépendances supplémentaires nécessaires pour **Laravel**. Heureusement, cette image de base inclut un outil pratique : **docker-php-ext-install**. Cet outil me permet d'installer les extensions **PDO** et **pdo_mysql**, qui sont essentielles pour **Laravel**.

Fichier : ./dockerfiles/php.dockerfile

```
FROM php:8.2.4-fpm-alpine
```

```
WORKDIR /var/www/html
```

```
COPY src .
```

```
RUN docker-php-ext-install pdo pdo_mysql
```

```
RUN addgroup -g 1000 laravel && adduser -G laravel -g laravel -s /bin/sh -D laravel && chmod 777 -R /var/www/html/storage/
```

```
# RUN chown -R www-data:www-data /var/www/html
```

```
USER laravel
```

Nous lançons également une commande pour créer un utilisateur Laravel et lui donner les droits sur le répertoire de travail.

- `addgroup -g 1000 laravel` : Cette commande crée un groupe utilisateur nommé laravel avec l'ID de groupe (GID) 1000.
- `adduser -G laravel -g laravel -s /bin/sh -D laravel` : Crée un nouvel utilisateur nommé laravel et l'ajoute au groupe laravel. L'option -s /bin/sh définit /bin/sh comme shell par défaut pour cet utilisateur, et -D crée cet utilisateur sans un répertoire personnel.
- `chmod 777 -R /var/www/html/storage/` : Modifie les permissions du répertoire /var/www/html/storage/ pour que tous les utilisateurs puissent lire, écrire et exécuter les fichiers dans ce répertoire. Le -R signifie que cette modification est appliquée de manière récursive à tous les sous-dossiers et fichiers à l'intérieur.

`USER laravel` : Cette commande définit l'utilisateur par défaut pour les instructions suivantes à l'intérieur du conteneur. Ainsi, toutes les opérations futures dans le conteneur seront effectuées par l'utilisateur laravel plutôt que root, ce qui est plus sécurisé.

Ce fichier **Dockerfile** ne contient pas de commande ou de point d'entrée (**entry point**).



Si vous n'ajoutez pas de commande dans un **Dockerfile**, celle de l'image de base est utilisée, et dans notre cas, l'image de base **PHP** a une commande par défaut qui invoque l'interpréteur **PHP**.

Ainsi, notre image personnalisée utilisera cette commande pour traiter les fichiers **PHP**.

10.5.2. Référencement de l'image dans le docker-compose

Dans le fichier **docker-compose.yaml**, nous pouvons maintenant référencer ce **Dockerfile**. Nous allons configurer la section **build** et spécifier le **context** (le dossier contenant notre projet à conteneuriser), qui est `./`, puis nous indiquerons le nom du fichier **Dockerfile** et son chemin par rapport au contexte, ici `php.dockerfile`. Cela permet d'utiliser des liens relatifs dans les Dockerfiles qui sont situés dans des sous-dossiers.

Fichier : `/docker-compose.yaml`

```
services:  
  server:  
    image: 'nginx:stable-alpine'  
    ports:  
      - 8000:80  
    volumes:  
      - ./nginx/nginx.conf:/etc/nginx/conf.d/default.conf:ro  
  
  php:
```

```
build:  
  context: ./  
  dockerfile: ./dockerfiles/php.dockerfile  
# mysql:  
# composer:  
# artisan:  
# npm:
```

10.5.3. Création du dossier SRC

Ensuite, il nous reste deux points importants à aborder. Le premier est de s'assurer que l'interpréteur **PHP** puisse accéder à notre code source. Même si nous n'avons pas encore de code source, nous aurons une application **Laravel PHP**, et ce code doit être accessible dans le dossier **/var/www/html** à l'intérieur du conteneur. Nous allons donc créer un montage (**bind mount**), pour monter notre dossier de projet local (par exemple, un dossier **src**) dans ce dossier à l'intérieur du conteneur.

Je vais créer un dossier **src** dans notre projet local. Ensuite, dans **docker-compose.yaml**, nous ajoutons la section **volumes** pour le conteneur **PHP**. Nous allons monter le dossier local **src** dans **/var/www/html** dans le conteneur. Nous pouvons aussi améliorer la performance en ajoutant **:delegated** à la fin du chemin. Cela optimise les performances en décalant les écritures vers le dossier monté.

Quand vous montez un dossier de votre machine hôte à l'intérieur d'un conteneur Docker, les modifications faites sur le contenu de ce dossier doivent être synchronisées entre le conteneur et l'hôte. Par défaut, chaque lecture ou écriture dans le conteneur est immédiatement répercutée sur votre machine locale, ce qui peut parfois ralentir les performances.

L'option **delegated** permet de décaler cette synchronisation pour améliorer les performances. En d'autres termes, le conteneur peut faire des modifications dans le dossier monté, mais ces changements ne seront pas immédiatement répercutés sur la machine hôte. Le conteneur aura une certaine priorité sur l'accès à ces fichiers, et les mises à jour sur la machine hôte seront effectuées après un léger délai.

Cela est utile dans des cas où les données modifiées à l'intérieur du conteneur n'ont pas besoin d'être immédiatement accessibles sur la machine hôte. Par exemple, si le conteneur écrit des fichiers de cache ou des fichiers temporaires qui n'ont pas besoin d'être visibles instantanément sur votre ordinateur.

Fichier : **./docker-compose.yaml**

```
services:  
  server:  
    image: 'nginx:stable-alpine'  
    ports:  
      - 8000:80  
    volumes:  
      - ./nginx/nginx.conf:/etc/nginx/conf.d/default.conf:ro
```

```
php:  
  build:  
    context: ./  
    dockerfile: ./dockerfiles/php.dockerfile  
  volumes:  
    - ./src:/var/www/html:delegated  
# mysql:  
# composer:  
# artisan:  
# npm:
```

10.5.4. Le port

Le second point important concerne le port sur lequel l'interpréteur **PHP** écoute les requêtes. Dans la configuration **nginx** (`nginx.conf`), nous avons défini un port **3000** pour les requêtes PHP.

Fichier : ./nginx/nginx.conf

```
[...]  
  
fastcgi_pass php:3000;  
  
[...]
```

Cependant, l'image **PHP** expose par défaut le port 9000, comme indiqué dans le fichier **Dockerfile** officiel. Nous devons donc mapper le port 3000 de **nginx** au port 9000 du conteneur **PHP**.

Fichier : ./docker-compose.yaml

```
services:  
  server:  
    image: 'nginx:stable-alpine'  
    ports:  
      - 8000:80  
    volumes:  
      - ./nginx/nginx.conf:/etc/nginx/conf.d/default.conf:ro  
  
  php:  
    build:  
      context: ./  
      dockerfile: ./dockerfiles/php.dockerfile  
    volumes:  
      - ./src:/var/www/html:delegated  
    ports:  
      - 3000:9000  
# mysql:  
# composer:  
# artisan:  
# npm:
```

Cela dit, nous devons nous rappeler que **nginx** communiquera directement avec le conteneur **PHP** via le réseau Docker, sans passer par la machine hôte, sans passer par une URL : localhost, le mapping est alors inutile. Par conséquent, nous allons simplement changer la configuration **nginx** pour qu'elle utilise le port 9000, car le trafic entre les conteneurs se fait directement via leurs noms de service.

Fichier : ./nginx/nginx.conf

```
[...]  
  
fastcgi_pass php:9000;  
  
[...]
```

Avec ce changement, nous avons terminé la configuration du conteneur **PHP**. Passons maintenant à la configuration du conteneur **MySQL**.

Fichier : ./docker-compose.yaml

```
services:  
  server:  
    image: 'nginx:stable-alpine'  
    ports:  
      - 8000:80  
    volumes:  
      - ./nginx/nginx.conf:/etc/nginx/conf.d/default.conf:ro  
  
    php:  
      build:  
        context: ./  
        dockerfile: ./dockerfiles/php.dockerfile  
      volumes:  
        - ./src:/var/www/html:delegated  
  # mysql:  
  # composer:  
  # artisan:  
  # npm:
```

10.6. Un conteneur MySQL

10.6.1. L'image

Passons maintenant à la configuration du conteneur **MySQL**.

Il n'est pas surprenant que nous ayons également une image **MySQL** officielle que nous pouvons utiliser. Nous allons l'utiliser pour lancer une base de données dans un conteneur **MySQL**. C'est très similaire à l'image **MongoDB** que nous avons utilisée précédemment dans le cours.

Dans le fichier **docker-compose.yaml**, nous allons ajouter une image. L'image que nous voulons

utiliser ici est l'image **MySQL**, et nous allons spécifier la version 5.7 avec le tag **5.7**. Cela téléchargera cette image, et lorsqu'elle sera démarrée en tant que conteneur, une base de données **MySQL** sera lancée.

Cette base de données sera accessible par notre code **PHP** dans le conteneur **PHP**. Comme tous ces services font partie du même réseau, nous pourrons communiquer avec cette base de données sans problème, simplement en utilisant le nom du conteneur.

Fichier : ./docker-compose.yaml

```
services:  
  server:  
    image: 'nginx:stable-alpine'  
    ports:  
      - 8000:80  
    volumes:  
      - ./nginx/nginx.conf:/etc/nginx/conf.d/default.conf:ro  
  
  php:  
    build:  
      context: ./  
      dockerfile: ./dockerfiles/php.dockerfile  
    volumes:  
      - ./src:/var/www/html:delegated  
  mysql:  
    image: mysql:5.7  
  
  # composer:  
  # artisan:  
  # npm:
```

10.6.2. Les variables d'environnement ENV

Il n'est donc pas nécessaire de configurer le réseau ici, mais il y a un autre aspect que nous devons configurer : **les variables d'environnement**. Nous devons fournir des variables d'environnement qui seront utilisées par cette image pour configurer une base de données, un utilisateur, un mot de passe, etc.

Nous trouvons la description détaillée de ces variables sur la page **Docker Hub** de l'image **MySQL**. Vous pouvez y voir quelles variables d'environnement peuvent être définies et ce qu'elles font.

Je vais définir plusieurs de ces variables, mais au lieu de les ajouter directement dans le fichier **docker-compose**, je vais les mettre dans un fichier **.env**. Je vais donc créer un dossier **env** et y ajouter un fichier **mysql.env**.

Dans ce fichier **mysql.env**, je vais ajouter la variable **MYSQL_DATABASE**, qui définit le nom de la base de données initiale qui sera créée lors du démarrage du conteneur. Je vais définir cette variable à **homestead**, ce qui correspond à la configuration par défaut dans la documentation **Laravel**.

Ensuite, je vais définir un utilisateur par défaut avec la variable **MYSQL_USER**, et je vais également

l'appeler **HOMESTEAD**, toujours en utilisant les valeurs par défaut de **Laravel**. Puis, je vais ajouter la variable **MYSQL_PASSWORD** pour définir le mot de passe de cet utilisateur initial, et je vais définir ce mot de passe à **secret**.

Enfin, je vais ajouter la variable **MYSQL_ROOT_PASSWORD** pour définir le mot de passe de l'utilisateur **root** de la base de données, et je vais également le définir à **secret**.

Fichier : `code/docker_laravel_complete/env/mysql.env`

```
MYSQL_DATABASE=homestead
MYSQL_USER=homestead
MYSQL_PASSWORD=secret
MYSQL_ROOT_PASSWORD=secret
```

Cela fait, nous retournons au fichier **docker-compose.yaml** pour ajouter l'option **env_file** sous le conteneur **MySQL**, et nous pointons vers le dossier **env** et le fichier **mysql.env**. Cela permettra d'utiliser les variables d'environnement définies dans ce fichier pour configurer le conteneur **MySQL**.

Fichier : `./docker-compose.yaml`

```
services:
  server:
    image: 'nginx:stable-alpine'
    ports:
      - 8000:80
    volumes:
      - ./nginx/nginx.conf:/etc/nginx/conf.d/default.conf:ro

  php:
    build:
      context: .
      dockerfile: ./dockerfiles/php.dockerfile
    volumes:
      - ./src:/var/www/html:delegated
  mysql:
    image: mysql:5.7
    env_file:
      - ./env/mysql.env

# composer:
# artisan:
# npm:
```

C'est tout pour nos trois conteneurs d'application.



Pour tester si tout fonctionne correctement ou si nous avons fait une erreur, nous aurons besoin d'une application **Laravel**. Nous allons la créer à l'aide de l'outil **Composer**. Passons donc à la configuration du conteneur **Composer**.

10.7. Un conteneur Composer

Ajoutons maintenant le service **Composer**.

Nous allons configurer ce conteneur, qui sera un conteneur utilitaire. Ce conteneur ne sera pas seulement utilisé en interne par **Laravel**, mais surtout par nous pour configurer l'application **Laravel** au départ.

Pour cela, je vais ajouter un autre **Dockerfile**, un fichier personnalisé, car je vais avoir besoin d'une image de base avec quelques ajustements. Je vais donc ajouter un fichier **composer.dockerfile**.

Je vais commencer par une image de base, l'image **Composer**. C'est vraiment pratique. Si vous cherchez sur **Docker Hub**, il existe une image de base **Composer**, qui inclut déjà l'outil **Composer**, ce qui est bien sûr génial. Nous pouvons donc partir de cette image **Composer** et utiliser par exemple le tag **latest** pour obtenir la dernière version de l'image.

Voici pourquoi j'ai besoin de mon propre **Dockerfile** personnalisé : je veux spécifier un point d'entrée (**entry point**). Petite parenthèse, il est aussi possible de faire cela directement dans le fichier **docker-compose**, mais je reviendrai sur ce point plus tard. J'aime cette approche, car elle est claire et facile à comprendre.



On ajoute aussi un utilisateur Laravel pour des questions de droits d'accès aux fichiers créés.

Fichier : `./dockerfiles/composer.dockerfile`

```
FROM composer

RUN addgroup -g 1000 laravel && adduser -G laravel -g laravel -s /bin/sh -D laravel

USER laravel

WORKDIR /var/www/html

ENTRYPOINT [ "composer", "--ignore-platform-reqs" ]
```

Dans mon fichier **Dockerfile**, le point d'entrée sera l'exécutable **composer** qui existe dans l'image **Composer** et dans le conteneur, contrairement à ma machine locale où il n'est pas installé (comme je l'ai montré précédemment). J'ajouterais également un flag **--ignore-platform-reqs** à chaque commande exécutée par **Composer**. Ce flag permet d'exécuter **Composer** sans avertissements ou erreurs, même si certaines dépendances manquent.

Il est très important de définir le bon répertoire de travail avec la commande **WORKDIR**, et je le fixe à **/var/www/html**. C'est là que notre code sera placé plus tard.

10.7.1. Configuration du build

Maintenant que nous avons notre **Dockerfile**, nous pouvons ajouter la configuration **build** pour le service **Composer** et définir le contexte avec **context: ./**. Nous définissons ensuite l'option

dockerfile à `./dockerfiles/composer.dockerfile`, car c'est le fichier que nous souhaitons utiliser pour construire cette image.

Nous devons maintenant nous assurer d'exposer notre répertoire de code source à cette image, afin que le conteneur puisse travailler sur ce répertoire lorsque nous utiliserons **Composer** pour installer **Laravel** et configurer le projet **Laravel**. Il doit bien sûr le faire dans le dossier `/var/www/html` à l'intérieur du conteneur.

Nous ajoutons donc un volume et lions notre dossier source local à ce répertoire `/var/www/html` dans le conteneur. Ainsi, si nous utilisons **Composer** pour créer une application **Laravel** dans ce dossier à l'intérieur du conteneur, cela sera reflété dans notre dossier source sur notre machine locale.

Fichier : `./docker-compose.yaml`

```
services:  
  server:  
    image: 'nginx:stable-alpine'  
    ports:  
      - 8000:80  
    volumes:  
      - ./nginx/nginx.conf:/etc/nginx/conf.d/default.conf:ro  
  
  php:  
    build:  
      context: ./  
      dockerfile: ./dockerfiles/php.dockerfile  
    volumes:  
      - ./src:/var/www/html:delegated  
  mysql:  
    image: mysql:5.7  
    env_file:  
      - ./env/mysql.env  
  
  composer:  
    build:  
      context: ./  
      dockerfile: ./dockerfiles/composer.dockerfile  
    volumes:  
      - ./src:/var/www/html  
  # artisan:  
  # npm:
```

Avec cela en place, nous avons désormais trois conteneurs d'application nécessaires pour exécuter l'application **Laravel**, et nous avons ce conteneur utilitaire pour la créer. Nous aurons besoin des deux autres conteneurs utilitaires pour certaines fonctionnalités spécifiques de **Laravel**, mais pour l'instant, nous pouvons les ignorer.



Nous pouvons maintenant commencer à utiliser le conteneur utilitaire **Composer** pour créer une application **Laravel**. Ensuite, nous vérifierons si nous pouvons

lancer cette application avec l'aide de nos trois conteneurs d'application.

10.8. Créer une application Laravel avec le conteneur utilitaire

Pour créer une application **Laravel**, nous pouvons consulter la documentation officielle de **Laravel**. Sous la section **Get Started**, nous voyons les prérequis. Si nous descendons un peu pour atteindre la section **Installing Laravel**, nous voyons cette commande que vous pouvez exécuter pour installer **Laravel**. Nous allons utiliser cette commande ici, car elle utilise uniquement **Composer** pour configurer un projet **Laravel**. Nous allons légèrement ajuster la commande concernant le dossier dans lequel nous voulons installer ce projet, mais c'est essentiellement la commande que nous allons utiliser.

Creating a Laravel Project

Before creating your first Laravel project, make sure that your local machine has PHP and [Composer](#) installed. If you are developing on macOS or Windows, PHP, Composer, Node and NPM can be installed in minutes via [Laravel Herd](#).

After you have installed PHP and Composer, you may create a new Laravel project via Composer's `create-project` command:

```
composer create-project laravel/laravel example-app
```

Or, you may create new Laravel projects by globally installing [the Laravel installer](#) via Composer. The Laravel installer allows you to select your preferred testing framework, database, and starter kit when creating new applications:

```
composer global require laravel/installer
```

```
laravel new example-app
```

Once the project has been created, start Laravel's local development server using Laravel Artisan's `serve` command:

```
cd example-app
```

```
php artisan serve
```

Nous pouvons donc déjà copier cette commande, puis revenir à notre terminal. Maintenant, je veux exécuter uniquement le conteneur **Composer**. C'est quelque chose que je vous ai montré dans la dernière section : vous pouvez exécuter des conteneurs individuels dans votre fichier **docker-compose.yaml** avec la commande `docker-compose run`. Vous n'avez pas besoin de démarrer l'ensemble du système de conteneurs, vous pouvez simplement exécuter des conteneurs individuels, ce qui est généralement le cas avec ces conteneurs utilitaires.

Ici, je veux exécuter le conteneur **Composer**. Donc je vais utiliser la commande `docker-compose run composer`, et j'ajouterai `--rm` pour m'assurer que tout ce qui est lié au conteneur est supprimé lorsqu'il est arrêté, afin que nous n'ayons pas une accumulation de conteneurs inutilisés après avoir exécuté cette commande plusieurs fois.

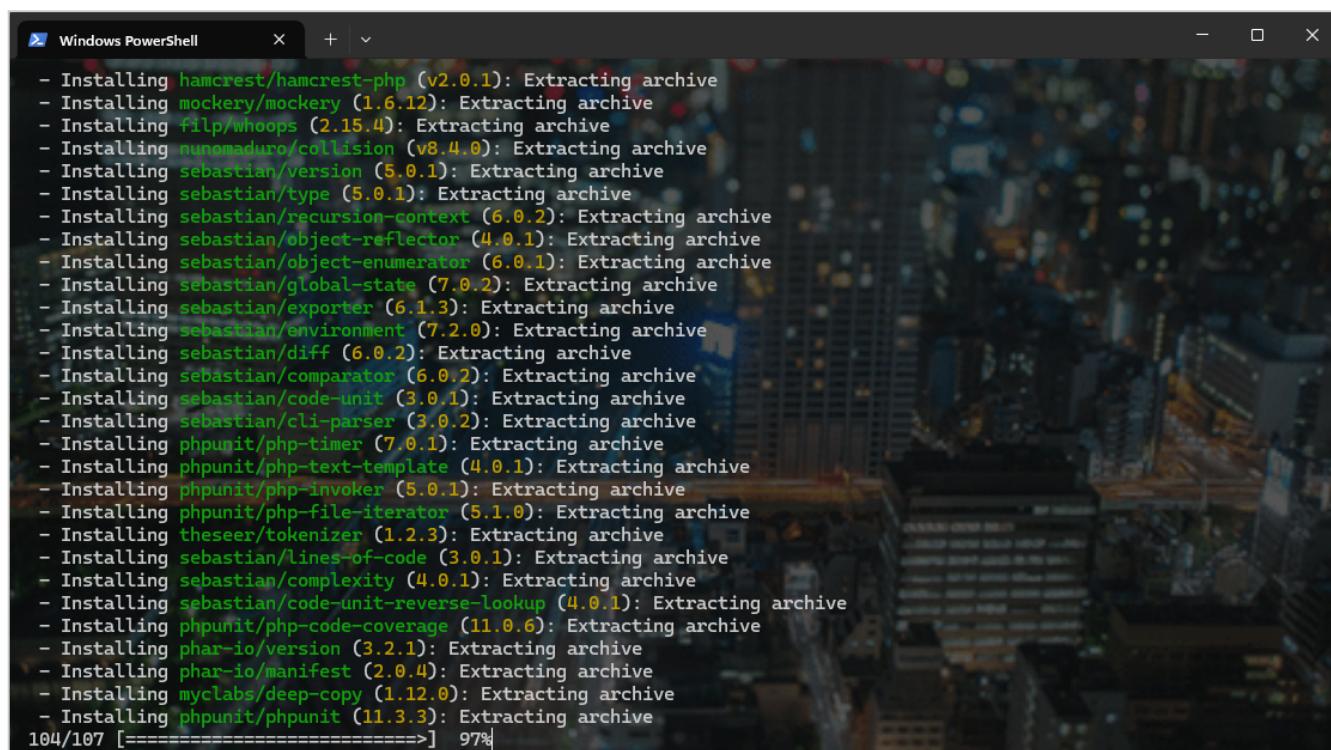
Ensuite, nous pouvons coller la commande que nous avons copiée du site **Laravel**. Toutefois, nous n'avons pas besoin de spécifier **composer** deux fois, car notre point d'entrée dans le **Dockerfile** **Composer** est déjà l'exécutable **composer**. Nous allons donc appeler la commande **create-project** directement sur cet exécutable.

Enfin, nous devons spécifier le dossier dans lequel ce projet doit être créé, et ce sera simplement . (le répertoire courant). Rappelez-vous que grâce à notre **Dockerfile**, tout cela s'exécute dans ce dossier à l'intérieur du conteneur, et ce sera donc le dossier racine où le projet **Laravel** sera créé.

```
docker-compose run --rm composer create-project --prefer-dist laravel/laravel .
```

Grâce à notre montage (**bind mount**), cela sera ensuite reflété dans le dossier **src** sur notre machine hôte.

Essayons maintenant cela en appuyant sur **Entrée**. Cela semble bien démarrer, l'image est en cours de construction, donc cela fonctionne. Voyons ce qu'il fait une fois terminé.



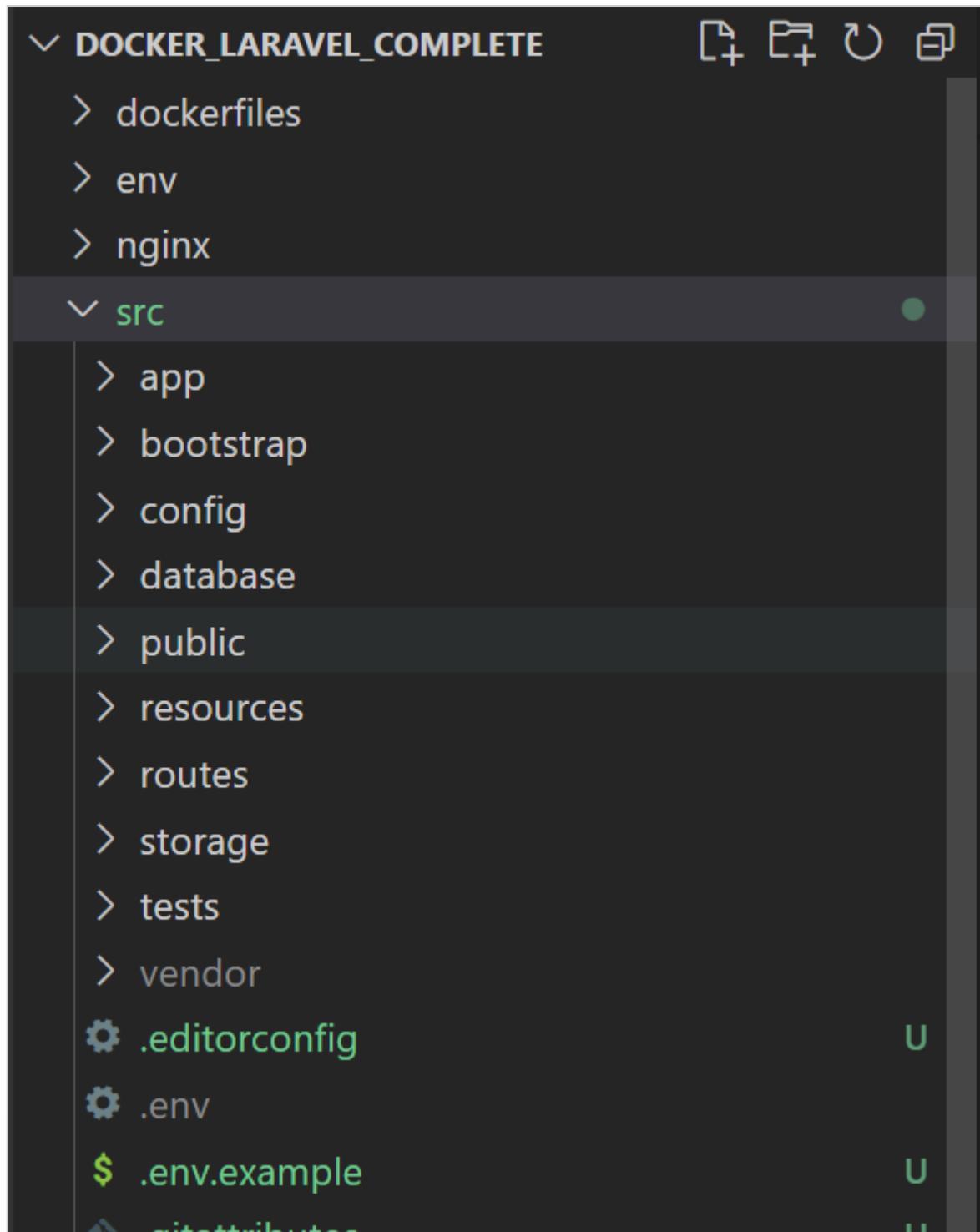
```
- Installing hamcrest/hamcrest-php (v2.0.1): Extracting archive
- Installing mockery/mockery (1.6.12): Extracting archive
- Installing filp/whoops (2.15.4): Extracting archive
- Installing runameduro/collision (v8.4.0): Extracting archive
- Installing sebastian/version (5.0.1): Extracting archive
- Installing sebastian/type (5.0.1): Extracting archive
- Installing sebastian/recursion-context (6.0.2): Extracting archive
- Installing sebastian/object-reflector (4.0.1): Extracting archive
- Installing sebastian/object-enumerator (6.0.1): Extracting archive
- Installing sebastian/global-state (7.0.2): Extracting archive
- Installing sebastian/exporter (6.1.3): Extracting archive
- Installing sebastian/environment (7.2.0): Extracting archive
- Installing sebastian/diff (6.0.2): Extracting archive
- Installing sebastian/comparator (6.0.2): Extracting archive
- Installing sebastian/code-unit (3.0.1): Extracting archive
- Installing sebastian/cli-parser (3.0.2): Extracting archive
- Installing phpunit/php-timer (7.0.1): Extracting archive
- Installing phpunit/php-text-template (4.0.1): Extracting archive
- Installing phpunit/php-invoker (5.0.1): Extracting archive
- Installing phpunit/php-file-iterator (5.1.0): Extracting archive
- Installing theseer/tokenizer (1.2.3): Extracting archive
- Installing sebastian/lines-of-code (3.0.1): Extracting archive
- Installing sebastian/complexity (4.0.1): Extracting archive
- Installing sebastian/code-unit-reverse-lookup (4.0.1): Extracting archive
- Installing phpunit/php-code-coverage (11.0.6): Extracting archive
- Installing phar-io/version (3.2.1): Extracting archive
- Installing phar-io/manifest (2.0.4): Extracting archive
- Installing myclabs深深-copy (1.12.0): Extracting archive
- Installing phpunit/phpunit (11.3.3): Extracting archive
104/107 [=====>] 97%
```

Cela semble également prometteur : le projet **Laravel** est en train d'être créé dans le dossier racine du conteneur, qui est </var/www/html>. Attendez-vous à ce que cela prenne quelques minutes.

Voilà, cela avance bien, et c'est normal que cela prenne du temps. Voyons si tout se termine correctement.

Et nous y sommes. Cela semble bon.

Si nous regardons maintenant dans le dossier **src**, nous y voyons notre application **Laravel**. Et, si vous connaissez **Laravel**, c'est ici que vous pourriez commencer à écrire du code **Laravel**.



Cependant, nous n'allons pas faire cela ensemble ici, car ce n'est pas un cours sur **Laravel**. Je voulais simplement vous montrer comment vous pouvez configurer une application **Laravel**.

Cela dit, nous n'avons pas encore terminé. Nous voulons maintenant voir si nous pouvons exécuter cette application avec nos trois conteneurs d'application.

10.9. Lancer des services Docker Compose à la carte

Pour commencer, à l'intérieur du dossier **source**, dans cette nouvelle application **Laravel**, je vais ouvrir le fichier **.env**. Ce fichier a été généré par **Laravel** et contient certaines configurations spécifiques à **Laravel**.

L'élément le plus important pour nous est ce bloc qui contient les informations de connexion que **Laravel** utilisera pour se connecter à une base de données **MySQL**. Nous devons ajuster ces paramètres pour permettre à **Laravel** de se connecter à la base de données.

Pour cela, nous devons d'abord modifier le nom de la base de données, le nom d'utilisateur et le mot de passe. Ici, nous devrions bien sûr utiliser les valeurs que nous avons définies lors de la configuration de notre serveur **MySQL**. Par exemple, j'ai choisi un nom d'utilisateur **homestead** et un mot de passe **secret**. Dans le fichier **.env**, nous allons donc utiliser **homestead** comme nom de base de données et utilisateur, et définir **secret** comme mot de passe.

Un point très important : l'hôte (**host**) n'est pas cette adresse IP, mais plutôt le nom de notre service **MySQL**. Dans ce cas, c'est **mysql**, car la requête sera envoyée à partir de l'application **PHP Laravel** à l'intérieur du conteneur, et **Docker** pourra résoudre ce nom de conteneur en adresse IP, étant donné que ces serveurs d'application fonctionneront dans le même réseau.

Avec ces ajustements, nous n'avons pas besoin de modifier autre chose pour le moment. Enregistrez le fichier **.env** et nous allons essayer de lancer cette application.

src/.env

```
[...]
DB_CONNECTION=mysql
DB_HOST=mysql
DB_PORT=3306
DB_DATABASE=homestead
DB_USERNAME=homestead
DB_PASSWORD=secret
[...]
```

Pour ce faire, nous devons utiliser la commande **docker-compose up** afin de démarrer les services que nous souhaitons. Mais avant cela, prenons un instant pour évaluer quels services seront démarrés. Ce seront le serveur (**nginx**), **PHP** et **MySQL**. Le serveur est notre point d'entrée principal, qui va servir l'application et rediriger les requêtes vers l'interpréteur **PHP**. L'interpréteur **PHP** communiquera indirectement avec la base de données **MySQL** via notre code, car nous allons nous connecter à celle-ci.

Cependant, nous avons un problème ici. Notre serveur (**nginx**) ne connaît actuellement rien de notre code source. L'interpréteur **PHP** le sait, mais ce n'est pas suffisant. La requête entrante atteint d'abord notre serveur, qui redirige ensuite les requêtes **PHP** vers l'interpréteur **PHP**. Cela signifie que les fichiers **PHP** doivent être accessibles au serveur, ce qui nécessite l'ajout d'un volume supplémentaire.

Ce volume supplémentaire sera un **bind mount** qui lie notre dossier source au dossier **/var/www/html** à l'intérieur du conteneur. Je choisis ce dossier car, dans la configuration **nginx**, c'est ce dossier qui est utilisé pour servir notre contenu, et c'est là que nous recherchons les fichiers. Bien sûr, nous cherchons principalement dans le dossier **public**, qui se trouve dans le dossier **source**.

Fichier : ./docker-compose.yaml

```
services:  
  server:  
    image: 'nginx:stable-alpine'  
    ports:  
      - 8000:80  
    volumes:  
      - ./src:/var/www/html  
      - ./nginx/nginx.conf:/etc/nginx/conf.d/default.conf:ro  
  
  php:  
    build:  
      context: ./  
      dockerfile: ./dockerfiles/php.dockerfile  
    volumes:  
      - ./src:/var/www/html:delegated  
  mysql:  
    image: mysql:5.7  
    env_file:  
      - ./env/mysql.env  
  
  composer:  
    build:  
      context: ./  
      dockerfile: ./dockerfiles/composer.dockerfile  
    volumes:  
      - ./src:/var/www/html  
# artisan:  
# npm:
```

Une fois ce volume ajouté, nous pouvons démarrer nos services. Pour cela, nous utilisons **docker-compose up** à nouveau.

Jusqu'à présent, nous avons toujours utilisé cette commande de manière simple. Cependant, cette fois-ci, nous ne voulons pas démarrer tous les services, comme **composer**. Nous voulons seulement démarrer trois services : **server**, **PHP** et **MySQL**.

La commande **docker-compose up** dispose d'une fonctionnalité spéciale pour cela. Si nous entrons **--help**, nous pouvons voir que nous avons la possibilité de cibler des services spécifiques. Par défaut, si nous exécutons **docker-compose up** sans spécifier de services, tous les services définis dans le fichier **docker-compose** seront démarrés. Mais ici, nous allons spécifier les services **server**, **PHP** et **MySQL**, et seuls ces trois services seront démarrés.

```
docker-compose up -d server php mysql
```

Faisons cela en mode détaché (**--detach**) et voyons si cela fonctionne.

Si nous visitons **localhost:8000** et rechargeons la page, nous devrions voir l'écran de démarrage de

10.10. Conteneurs et leurs dépendances

Bien que cette méthode fonctionne, taper chaque service individuellement peut être fastidieux. Il serait préférable de spécifier que le service **server** dépend de **PHP** et **MySQL**, de sorte que **docker-compose** démarre automatiquement ces services. Nous pouvons ajouter cette configuration en utilisant **depends_on** dans le service **server**.

Fichier : `./docker-compose.yaml`

```
services:  
  server:  
    image: 'nginx:stable-alpine'  
    ports:  
      - 8000:80  
    volumes:  
      - ./src:/var/www/html  
      - ./nginx/nginx.conf:/etc/nginx/conf.d/default.conf:ro  
    depends_on:  
      - php  
      - mysql  
  
  php:  
    build:  
      context: ./  
      dockerfile: ./dockerfiles/php.dockerfile  
    volumes:  
      - ./src:/var/www/html:delegated  
  mysql:  
    image: mysql:5.7  
    env_file:  
      - ./env/mysql.env  
  
  composer:  
    build:  
      context: ./  
      dockerfile: ./dockerfiles/composer.dockerfile  
    volumes:  
      - ./src:/var/www/html  
  # artisan:  
  # npm:
```

En ajoutant cette dépendance, **docker-compose** s'assurera que lorsque nous démarrons le service **server**, les services **PHP** et **MySQL** seront également démarrés. Nous pouvons maintenant lancer tous les services dépendants en une seule commande.

Un dernier ajustement concerne les images personnalisées comme l'image **PHP**. Par défaut, **docker-compose** ne reconstruit pas les images, même si vous avez fait des modifications. Pour

forcer la reconstruction, nous pouvons ajouter l'option `--build` à la commande `up`, ce qui permet à `docker-compose` de vérifier les **Dockerfile** et de reconstruire les images si nécessaire.

En utilisant `--build`, nous nous assurons que toute modification dans les **Dockerfile** sera prise en compte. Avec cela, nous avons notre application **Laravel** qui fonctionne et nous pouvons maintenant modifier le code dans le dossier **source**. Par exemple, nous pouvons aller dans `resources/views` et modifier le fichier `welcome.blade.php` pour ajouter une balise `<h1>`. Après avoir sauvegardé et rechargé la page, nous voyons le changement.

Nous pouvons maintenant travailler sur cette application **Laravel** avec cette configuration.

Il ne nous reste plus qu'à configurer les deux autres conteneurs utilitaires, dont nous n'avons pas besoin pour démarrer l'application, mais qui sont nécessaires pour exécuter des migrations ou gérer le code **JavaScript** côté client. Passons maintenant à ces deux conteneurs.

10.11. Le conteneur Artisan

Continuons avec le conteneur **Artisan**.

Nous avons besoin de l'outil **Artisan** pour exécuter certaines commandes **Laravel**. Par exemple, pour peupler la base de données avec des données initiales.

Le conteneur **Artisan** nécessite un fichier **Dockerfile** personnalisé. Cependant, je vais simplement utiliser le **Dockerfile** du conteneur **PHP** ici, car j'ai besoin de la même configuration que nous avons pour le conteneur **PHP**. Nous pouvons donc ajouter la configuration de construction ici ou simplement copier celle du conteneur **PHP** et l'ajouter pour **Artisan**. Il a besoin de **PHP** pour exécuter du code, car **Artisan** est une commande **Laravel** construite avec **PHP**. Il a donc besoin de **PHP** pour fonctionner.

Pour configurer le conteneur **Artisan**, qui nous permet d'exécuter des commandes spécifiques à **Laravel**, nous allons utiliser le Dockerfile du conteneur **PHP** précédemment défini. Étant donné qu'**Artisan** est un outil intégré à **Laravel**, basé sur **PHP**, il nécessite l'environnement de ce dernier pour fonctionner. Ainsi, nous n'avons pas besoin d'un Dockerfile distinct, mais nous ajouterons un point d'entrée (entrypoint) spécifique dans le fichier `docker-compose`.

Fichier : `./docker-compose.yaml`

```
services:
  server:
    image: 'nginx:stable-alpine'
    ports:
      - 8000:80
    volumes:
      - ./src:/var/www/html
      - ./nginx/nginx.conf:/etc/nginx/conf.d/default.conf:ro
    depends_on:
      - php
      - mysql

  php:
```

```
build:
  context: ./
  dockerfile: ./dockerfiles/php.dockerfile
  volumes:
    - ./src:/var/www/html:delegated
mysql:
  image: mysql:5.7
  env_file:
    - ./env/mysql.env

composer:
  build:
    context: ./
    dockerfile: ./dockerfiles/composer.dockerfile
  volumes:
    - ./src:/var/www/html
artisan:
  build:
    context: ./
    dockerfile: ./dockerfiles/php.dockerfile
  volumes:
    - ./src:/var/www/html
  entrypoint: ["php", "/var/www/html/artisan"]
# npm:
```

L'ajout de l'option `entrypoint` dans `docker-compose` permet de surcharger ou de définir un point d'entrée pour un conteneur sans modifier le Dockerfile de base. Ici, le point d'entrée sera l'exécution du fichier `artisan` avec PHP, situé dans le répertoire `/var/www/html`. Ce fichier, déjà présent dans notre dossier source, exécute des tâches spécifiques à Laravel, telles que les migrations de base de données.

10.12. Le conteneur NPM

Quant au conteneur NPM, nous utiliserons l'image officielle de Node.js, telle que `node:latest`. Plutôt que de créer un Dockerfile personnalisé, nous définirons directement dans `docker-compose` le répertoire de travail à `/var/www/html` et le point d'entrée à `npm`. Cela nous permettra d'exécuter les commandes `npm` à l'intérieur du conteneur, tout en exposant notre répertoire source à Docker.

Fichier : `./docker-compose.yaml`

```
services:
  server:
    image: 'nginx:stable-alpine'
    ports:
      - 8000:80
    volumes:
      - ./src:/var/www/html
      - ./nginx/nginx.conf:/etc/nginx/conf.d/default.conf:ro
    depends_on:
```

```
- php  
- mysql  
  
php:  
  build:  
    context: ./  
    dockerfile: ./dockerfiles/php.dockerfile  
  volumes:  
    - ./src:/var/www/html:delegated  
mysql:  
  image: mysql:5.7  
  env_file:  
    - ./env/mysql.env  
  
composer:  
  build:  
    context: ./  
    dockerfile: ./dockerfiles/composer.dockerfile  
  volumes:  
    - ./src:/var/www/html  
artisan:  
  build:  
    context: ./  
    dockerfile: ./dockerfiles/php.dockerfile  
  volumes:  
    - ./src:/var/www/html  
  entrypoint: ["php", "/var/www/html/artisan"]  
  
npm:  
  image: node:latest  
  working_dir: /var/www/html  
  entrypoint: ["npm"]  
  volumes:  
    - ./src:/var/www/html
```

Avec ces configurations en place, nous pouvons maintenant exécuter des commandes Artisan via docker-compose run pour des tâches telles que les migrations. Cela permettra d'écrire des données dans la base de données et de tester la connexion à MySQL.

Ainsi, nous obtenons une configuration complète pour une application Laravel PHP dans Docker, intégrant plusieurs services tels que nginx, PHP, MySQL, Artisan, et NPM. Cette configuration montre comment utiliser Docker pour orchestrer une architecture plus complexe, en facilitant le développement local sans dépendances supplémentaires sur la machine hôte.

10.13. Test des conteneurs outils

```
docker-compose run --rm artisan migrate
```

10.14. Conclusion

Nous avons maintenant une configuration complète pour une application **Laravel PHP**. C'est une configuration plus complexe, mais c'est l'objectif de ce cours : apprendre les bases et voir comment les appliquer dans des projets plus complexes.

Lorsque vous construisez quelque chose comme cela par vous-même, il est tout à fait normal de le faire étape par étape en cherchant des solutions, en suivant des cours comme celui-ci. Avec le temps, vous arriverez à construire des projets comme celui-ci de manière autonome.

Il est évident que cela aide aussi si vous connaissez **Laravel** et **PHP**, car vous savez quels blocs constitutifs utiliser, comme **Composer**. Si vous ne connaissez pas ces technologies, il est normal de ne pas savoir que vous en avez besoin. Ne vous découragez donc pas si vous ne connaissez pas **PHP** ou **Laravel**.

10.15. Docker Compose AVEC et SANS Dockerfiles

Avec cette configuration en place, nous avons un environnement fonctionnel en utilisant les outils vus dans ce cours. Cependant, certains concepts introduits ici méritent d'être abordés, comme l'ajout d'instructions **Docker** directement dans le fichier `docker-compose`, telles que `entrypoint` ou `working_dir`. Bien que cela soit possible, ce n'est pas obligatoire. Une alternative consiste à créer un fichier Dockerfile séparé et à le référencer dans `docker-compose`.

Personnellement, je préfère avoir des fichiers **Dockerfile** distincts, car cela clarifie mes intentions et allège le fichier `docker-compose`. Cependant, cela implique de consulter plusieurs fichiers pour comprendre la configuration complète. Pour des instructions plus complexes, comme exécuter une commande spécifique ou copier des fichiers, un **Dockerfile** est de toute façon nécessaire, car `docker-compose` ne prend pas en charge ces instructions.

Concernant les **bind mounts**, illustrés ici par le service `nginx`, il est important de rappeler qu'ils sont très utiles en phase de développement. Ils permettent de lier un dossier local à un conteneur, comme le dossier source ou la configuration nginx. Cependant, les bind mounts ne sont pas adaptés pour la production, car ils lient des dossiers qui existent uniquement sur la machine hôte.



Si vous déployez un conteneur sur un serveur distant, les dossiers montés ne seront pas disponibles.

L'objectif des conteneurs est d'inclure toutes les ressources nécessaires à leur exécution, sans dépendre de l'environnement hôte. Par conséquent, pour le déploiement, il serait préférable de créer un Dockerfile pour le serveur, incluant une copie du code source et de la configuration nginx dans l'image. Cela garantirait que l'image déployée contienne tout ce dont elle a besoin, indépendamment des bind mounts utilisés en développement.

10.16. Bind Mount ou COPY : Quand ?

Pour ajouter un fichier `nginx.dockerfile` dans notre dossier dockerfiles, nous pouvons partir de l'image de base `nginx:stable-alpine`. Ensuite, nous définirons le répertoire de travail (working directory) dans le dossier de configuration, en omettant le nom du fichier, afin de copier notre

fichier de configuration local `nginx.conf` dans ce répertoire. Cela se fait via l'instruction COPY, qui permet de transférer le fichier de configuration local dans le répertoire de travail défini dans le conteneur.

Une fois le fichier copié, il est nécessaire de le renommer en `default.conf`, car c'est ce qu'attend nginx. Cela peut être fait avec la commande `mv`, qui permet de renommer `nginx.conf` en `default.conf` directement dans le répertoire de travail.

Ensuite, nous passons au répertoire `/var/www/html` et copions notre code source, contenu dans le dossier `src`, dans ce répertoire à l'intérieur du conteneur. Cela permet d'inclure un instantané de notre code source dans l'image, sans dépendre uniquement du bind mount utilisé pour le développement.

En procédant ainsi, nous garantissons que l'image du conteneur contient toujours une version instantanée du code source et de la configuration au moment de la construction. Le bind mount reste utile pendant le développement, car il permet de lier les modifications du code source local au conteneur, mais il n'est pas utilisable lors du déploiement.

Nous n'avons pas besoin de définir de entrypoint ou de commande dans ce fichier Dockerfile, car l'image nginx inclut déjà une commande par défaut qui démarre le serveur web. Dans le fichier `docker-compose.yaml`, il suffit alors de remplacer l'image de base par notre nouvelle image personnalisée, en définissant la configuration de construction appropriée.

Cependant, il est important de bien définir le context dans le fichier `docker-compose`. Ce context ne doit pas seulement pointer vers le dossier `dockerfiles`, mais également inclure le dossier principal du projet. Cela permet de s'assurer que tous les fichiers nécessaires, tels que le dossier `nginx` et le dossier `src`, sont accessibles pendant la construction de l'image. Pour ce faire, nous définissons le context à `.` (le répertoire racine du projet) et ajustons l'instruction `dockerfile` en précisant le chemin vers notre fichier `nginx.dockerfile`.

Une fois ces modifications effectuées, nous pouvons tester notre configuration en exécutant la commande `docker-compose up` sans les bind mounts, afin de vérifier que l'application fonctionne correctement avec l'instantané de code intégré dans l'image.

Un ajustement similaire doit être effectué pour le fichier `php.dockerfile`. Nous ajoutons une instruction COPY pour inclure le dossier `src` dans le répertoire `/var/www/html` à l'intérieur du conteneur. Cela permet de garantir que le code source est intégré au conteneur lors du déploiement. Cependant, pendant le développement, nous pouvons continuer à utiliser les bind mounts pour bénéficier de la mise à jour en temps réel des modifications du code.

Si nous rencontrons des erreurs liées aux permissions d'accès au code source à l'intérieur du conteneur, nous pouvons les résoudre en utilisant la commande `chown` dans le fichier Dockerfile pour accorder des droits en lecture et écriture à l'utilisateur par défaut de `php` (`www-data`), ce qui est essentiel pour permettre à Laravel de générer des fichiers pendant l'exécution.

Après avoir apporté ces ajustements, nous pouvons relancer le projet avec `docker-compose up --build` pour reconstruire les images et vérifier que l'application Laravel fonctionne comme prévu.

Enfin, pour faciliter l'exécution des commandes Artisan dans notre environnement, nous devons nous assurer que le context du service Artisan est correctement défini pour inclure le chemin du

php.dockerfile. Cela garantit que toutes les commandes Artisan fonctionnent comme prévu, y compris les migrations de base de données.

Cette configuration aboutit à un environnement de développement et de déploiement complet et flexible, adapté à des projets complexes comme une application Laravel PHP.

11. Déploiement de conteneurs Docker

Version 1.0.0 | Dernière mise à jour : 17/11/2024 | Auteur : Bauer Baptiste

11.1. Introduction



Dans ce module, nous allons aborder un sujet très vaste et détaillé : le déploiement de conteneurs Docker. Contrairement aux chapitres précédents où nous nous sommes limités à travailler sur notre machine locale, nous allons maintenant exploiter ce que vous avez appris pour exécuter des conteneurs sur des machines distantes, que ce soit dans le **cloud** ou sur le **web**. Cela signifie que nos applications seront accessibles par les utilisateurs que nous souhaitons atteindre.

11.1.1. Objectif du chapitre

L'**objectif principal** de ce chapitre est de se concentrer sur le déploiement d'applications web. Bien que Docker puisse être utilisé pour bien d'autres types d'applications, nous avons choisi de nous concentrer sur les applications web pour plusieurs raisons. Notamment, Docker offre de nombreux avantages en développement web et permet de créer des applications accessibles via un navigateur, facilitant ainsi les tests et la démonstration des concepts abordés dans ce cours.

- **Passer du Développement Local à la Production**

Dans ce chapitre, vous apprendrez à passer du **développement local à la production**. Cela implique de **déplacer vos conteneurs sur une machine hôte distante** et de les exécuter depuis cet emplacement. Nous couvrirons les aspects généraux de ce processus de déploiement, les éléments à garder à l'esprit, les problèmes courants auxquels vous pourriez être confrontés, et les modifications nécessaires pour que vos conteneurs soient prêts pour un environnement de **production**.

Je vous fournirai également de nombreux scénarios concrets, des exemples pratiques et des problématiques que vous pourrez rencontrer en réalité. Vous apprendrez à déployer des applications comprenant un seul conteneur ou plusieurs conteneurs, et ce, sur une ou plusieurs machines hôtes.

• Approches de Déploiement

Nous examinerons deux approches principales pour le déploiement :

Approche auto-gérée : Dans cette méthode, vous serez responsable d'une machine distante, sur laquelle vous installerez Docker et exécuterez vos conteneurs. Cela vous donne un contrôle total, mais nécessite une gestion et une maintenance continues.

Approche gérée : Ici, nous utiliserons un service géré pour le déploiement de conteneurs. Cette méthode offre certains avantages, notamment en termes de simplicité et de gestion allégée, au détriment d'une flexibilité réduite.

Nous explorerons ces deux méthodes en détail, afin de vous donner une compréhension complète des options disponibles.

Utilisation d'AWS comme Exemple

Pour illustrer ces concepts, nous utiliserons **AWS (Amazon Web Services)** comme exemple de fournisseur d'hébergement. AWS étant l'un des plus grands fournisseurs de services cloud, il sert d'excellent point de référence.



Les principes et concepts que vous apprendrez resteront applicables même si vous choisissez un autre fournisseur.

Bien que certains exemples soient spécifiquement conçus pour AWS, les leçons générales, les points sensibles à surveiller et les aspects importants pour la préparation de vos conteneurs pour la production sont pertinents, quel que soit l'environnement de déploiement choisi.

11.1.2. Conclusion

Ce chapitre est très riche en contenu et en exemples pratiques. À la fin de cette section, vous aurez une compréhension claire de ce que signifie déployer des conteneurs, des éléments à garder à l'esprit, et des étapes à suivre pour déployer vos applications conteneurisées avec succès.

11.2. Du Développement au Déploiement

Jusqu'à présent, nous avons appris que les conteneurs sont des outils formidables qui simplifient considérablement le développement. Nous nous sommes principalement concentrés sur l'utilisation des conteneurs en phase de développement. Cependant, il est essentiel de comprendre que les conteneurs sont des paquets indépendants et isolés, contenant le code de l'application ainsi que son environnement d'exécution, ce qui nous permet de les déployer partout où Docker est installé. Grâce à leur standardisation, les conteneurs créés avec Docker peuvent être exécutés de manière cohérente sur n'importe quelle machine dotée de Docker.

11.2.1. Les Conteneurs : Une Solution aux Problèmes de Déploiement



Dans la première section de ce cours, nous avons évoqué comment les conteneurs peuvent résoudre les problèmes courants liés au déploiement des applications. Sans conteneurs, les différences entre les environnements de développement et de production peuvent souvent entraîner des dysfonctionnements lorsque l'application est déployée sur un serveur distant. Docker et les conteneurs permettent d'unifier ces environnements, en garantissant que le même environnement utilisé lors du développement local est également disponible lors de l'exécution de l'application sur un hôte distant.



Cela élimine le besoin de configurer manuellement la machine elle-même ou d'installer des outils comme **Node.js**, que ce soit en **local** ou sur un serveur de **production**.

Les Avantages des Conteneurs en Développement et en Production

Les conteneurs offrent un environnement isolé et autonome, aussi bien en développement qu'en production. C'est pourquoi nous avons consacré de nombreuses sections à l'environnement de développement local : ce que vous apprenez à ce stade vous sera également utile en production. Vous serez en mesure de déployer les mêmes conteneurs sur lesquels vous avez travaillé localement sur des machines distantes, ce qui est précisément l'objectif de ce chapitre.

Nous verrons comment ces environnements **reproductibles**, **facilement partageables** et **utilisables**, facilitent le déplacement de l'application depuis la machine locale (où elle est encapsulée dans une image Docker) vers une machine distante, prête à servir les utilisateurs finaux de l'application.

Les conteneurs sont la solution parfaite!

En développement

Isolés, autonomes,
indépendants

En production

Isolés, autonomes,
indépendants

Environnement
reproductible, facilement
partageable et utilisable

Environnement
reproductible, facilement
partageable et utilisable



En utilisant Docker et les conteneurs, nous minimisons les surprises indésirables : ce qui fonctionne localement dans un conteneur fonctionnera également après le déploiement sur une machine distante.

11.2.2. Déploiement de Conteneurs : Points Clés à Considérer

Le déploiement des conteneurs implique de les transférer sur des machines distantes. Il y a plusieurs aspects importants à prendre en compte :

- Utilisation des "Bind Mounts"** : Pendant le développement, nous utilisons fréquemment les bind mounts. Cependant, **il est déconseillé de les utiliser en production**. Nous reviendrons sur les raisons de cette différence un peu plus loin dans le module.
- Définitions de Configuration** : Les applications conteneurisées peuvent nécessiter des configurations différentes pour le développement et la production. Bien que le concept des conteneurs repose sur un environnement identique, certaines applications, comme celles utilisant React, nécessitent une étape de compilation où le code est converti et optimisé avant le déploiement. Ce processus de construction se déroule après le développement et avant le déploiement. Ce chapitre vous montrera comment garantir des environnements reproductibles malgré ces différences nécessaires.
- Projets Multi-Conteneurs** : Dans des projets utilisant plusieurs conteneurs, il peut être pertinent de distribuer ces conteneurs sur plusieurs hôtes, c'est-à-dire plusieurs machines distantes. Bien que le test en local puisse être effectué avec **Docker Compose** sur une seule machine, le déploiement pourrait bénéficier d'une répartition sur plusieurs serveurs.
- Compromis de Contrôle et de Responsabilité** : Tout au long de ce module, nous aborderons des situations où il peut être avantageux de choisir des solutions offrant moins de contrôle sur la plateforme de déploiement, mais aussi moins de responsabilités. Par exemple, la gestion d'un

hôte distant complet entraîne une charge de travail supplémentaire pour assurer la sécurité et la maintenance de cette machine. Opter pour des solutions avec moins de contrôle, mais aussi moins de responsabilité peut parfois être plus avantageux et rendre la tâche du développeur plus aisée.

En résumé, ce chapitre vous guidera à travers le processus de déploiement de conteneurs sur des machines distantes, en tenant compte des différences entre développement et production, et en abordant les stratégies pour les projets multi-conteneurs. Vous découvrirez comment gérer les compromis entre contrôle et responsabilité, tout en maintenant des environnements cohérents et reproductibles. À la fin de ce chapitre, vous serez capable de déplacer vos conteneurs depuis une machine locale vers un environnement de production sécurisé et efficace, que ce soit sur le cloud ou ailleurs.

Du développement à la mise en prod : les choses à garder en tête !

Ne pas utiliser de
« Bind Mounts »

Certaines applications
conteneurisées nécessitent une
étape de compilation

Les projets multi-conteneurs ont souvent
besoin d'être séparé vers plusieurs
machines (hôtes) distantes

Le compromis entre le contrôle et la
responsabilité pourrait en valoir la
peine.

11.3. Processus de déploiement et fournisseurs

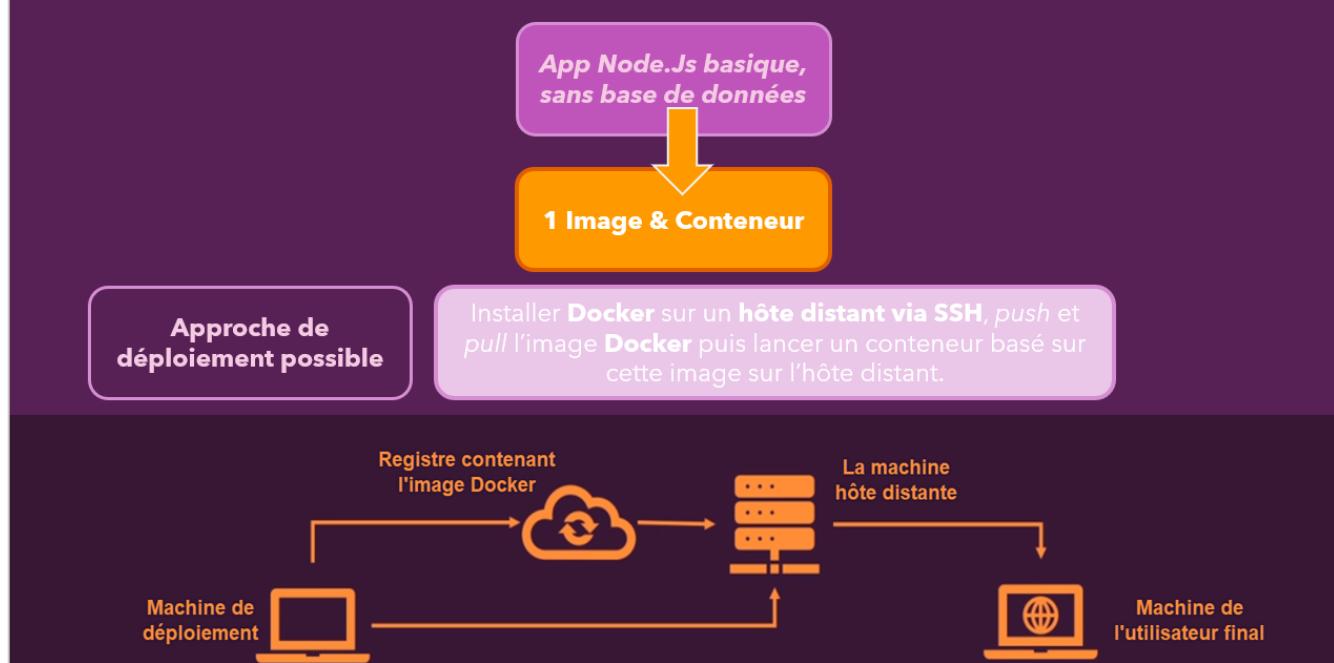
Dans cette partie, nous allons explorer différents types de projets utilisant Docker, que ce soit pour déployer un ou plusieurs conteneurs sur un ou plusieurs serveurs cibles. Nous allons examiner ces différents scénarios de manière progressive, en commençant par des exemples simples et en construisant sur cette base pour aller vers des situations plus complexes.

11.3.1. Introduction avec un Exemple Simple

Pour débuter, nous allons nous concentrer sur un exemple de base afin de comprendre les fondamentaux avant de plonger dans des cas plus complexes. Nous utiliserons une simple application `Node.js` sans base de données ni autres composants. Ce scénario de départ impliquera une seule image Docker, un seul conteneur, et une seule application contenue dans cette image et ce conteneur.

Voici les étapes de base que nous suivrons pour ce déploiement :

Exemple d'Application simple



1. Configuration d'un **Serveur Distant** : Nous commencerons par configurer un serveur distant, en installant Docker sur cet hôte via une connexion SSH, par exemple.
2. **Transfert de l'Image Docker** : Nous transférerons notre image Docker depuis notre machine locale, où nous avons développé l'application, vers un registre Docker comme Docker Hub. Ensuite, nous tirerons cette image depuis le registre vers notre serveur distant.
3. **Exécution du Conteneur** : Une fois l'image sur le serveur distant, nous exécuterons le conteneur et exposerons les ports nécessaires pour que les utilisateurs puissent accéder à notre application via le Web.

Cette approche nous permettra de comprendre les bases du déploiement de conteneurs **Docker** sur un serveur distant, y compris la configuration, le transfert et la mise en ligne d'une application.

11.3.1.1. Choix d'un Fournisseur de Serveurs Distants

Pour démarrer, nous aurons besoin d'un **serveur distant**, et il existe de nombreux fournisseurs d'hébergement prenant en charge Docker.

Une recherche rapide sur Google pour "**fournisseur d'hébergement Docker**" vous donnera une liste d'options, ainsi que des articles comparatifs sur les meilleures solutions disponibles. Il est important de consulter la documentation spécifique du fournisseur choisi, car les étapes pour connecter votre machine distante et déployer des conteneurs peuvent varier.

Les trois grands fournisseurs de services de **cloud computing** que nous considérons sont :





1. AWS (Amazon Web Services)
2. Microsoft Azure
3. Google Cloud Platform

Ces plateformes ne se limitent pas à l'hébergement web; elles proposent une large gamme de services cloud, allant du développement web au "**deep learning**". Dans cette section, nous utiliserons l'une de ces plateformes, **AWS**, pour nos démonstrations de déploiement d'applications web.

Pourquoi Choisir AWS pour ce Module?

Nous utiliserons **AWS** dans ce cours, car c'est le plus grand fournisseur de services cloud. **AWS** propose plusieurs services qui peuvent être utilisés avec des conteneurs Docker, et nous examinerons certains des plus importants pour les déploiements d'applications.

Bien que ce cours ne soit pas **un cours complet sur l'hébergement cloud**, il vous donnera une bonne compréhension des différentes options disponibles pour déployer des conteneurs **Docker** sur **AWS**.

11.3.1.2. Conditions Préalables et Options Gratuites



Pour suivre ce cours, il vous faudra une carte de crédit, une condition requise par la plupart des principaux fournisseurs d'hébergement. Si vous n'en avez pas, vous pouvez lire le cours et passer à la pratique une fois que vous aurez **un projet concret à déployer**.

- Allez sur : <https://aws.amazon.com/>
- Placez le curseur sur : Tarification puis cliquez sur Offre gratuite d'AWS

The screenshot shows the AWS Pricing page with several sections:

- Tarification**: A large icon of a person with a dollar sign inside a book.
- En savoir plus sur la tarification d'AWS**: Text explaining AWS's pay-as-you-go pricing model.
- Offre gratuite d'AWS**: A box containing information about the AWS Free Tier offer.
- Calculateur de tarification AWS**: An icon of a calculator and charts.
- Faites une estimation des coûts de votre solution d'architecture**: Text below the calculator icon.
- Optimisez vos coûts**: An icon of a checklist and a calculator.
- Découvrez les étapes à suivre pour optimiser efficacement vos coûts AWS**: Text below the optimization icon.

Il est également utile de consulter la page de tarification d'AWS, qui propose une offre gratuite (free tier) pour de nombreux services, y compris EC2, qui est leur service de machine distante.

Par exemple, **AWS offre 750 heures d'instance gratuites par mois** pour les 12 premiers mois, ce qui vous permet de faire fonctionner un serveur distant gratuitement pendant cette période.

The screenshot shows the "Détails des offres gratuites" (Details of free offers) section on the AWS website. It lists three main categories of free offers:

- CALCUL**: Offers a **12 MOIS GRATUITS** (12 months free) for **Amazon EC2**. It provides **750 heures** (750 hours) per month of resizable compute capacity in the cloud. It also mentions **750 heures d'utilisation des instances t2.micro**.
- STOCKAGE**: Offers a **12 MOIS GRATUITS** (12 months free) for **Simple Storage Service (Amazon S3)**. It provides **5 Go** (5 GB) of standard storage, infrastructure for securely storing objects, and scalability.
- BASE DE DONNÉES**: Offers a **12 MOIS GRATUITS** (12 months free) for **Amazon RDS**. It provides **750 heures** (750 hours) of monthly usage of database engines like MySQL, PostgreSQL, MariaDB, or SQL Server.

A red box highlights the **Amazon EC2** offer under the **CALCUL** category.

11.3.1.3. Lancement d'une Instance EC2

Pour commencer, nous lancerons une instance EC2 (*un serveur distant hébergé par AWS*). Je vous montrerai comment utiliser cette instance pour installer Docker, lancer un conteneur Docker sur cette machine distante, et enfin tester notre application dans un navigateur.

Cette section vous guidera à travers le processus complet de configuration d'un serveur distant, de l'installation de Docker, du transfert d'images Docker, de l'exécution des conteneurs et de la mise en ligne de votre application. Vous serez ainsi bien équipé pour déployer vos propres projets Docker en production sur des serveurs distants.

11.4. Déploiement d'une Application Node Basique sur AWS EC2

Téléchargez le fichier ressource contenant l'application NodeJs qui servira de fil conducteur pour

cette section :

- Fichier ressource : [deployment-01-starting-setup.zip](#)

Dans cet exemple, nous allons déployer une application Node.js simple, sans base de données ni autres composants, sur une instance **EC2 d'AWS**. Comme expliqué dans la leçon précédente, **EC2 (Elastic Compute Cloud)** est un service proposé par **AWS** qui vous permet de créer vos propres machines d'hébergement distantes, autrement dit, vos propres ordinateurs dans le cloud.

Vous pouvez vous connecter à ces ordinateurs pour installer le logiciel de votre choix. Dans notre cas, nous allons y installer Docker et l'utiliser pour exécuter nos applications conteneurisées.

11.4.1. Étapes Principales pour Déployer une Application conteneurisée sur EC2

Pour mettre en ligne notre application conteneurisée sur une instance EC2, nous allons suivre trois étapes principales :

1. Créer et Lancer une **Instance EC2** : La première étape consiste à créer et démarrer une instance EC2, c'est-à-dire un ordinateur distant. Nous devrons également créer un **VPC (Virtual Private Cloud)** et un groupe de sécurité, afin de contrôler l'accès à cette instance. Je vous montrerai comment configurer tout cela.
2. Configurer le **Groupe de Sécurité** : Nous devrons ajuster les paramètres du groupe de sécurité pour ouvrir les ports nécessaires et permettre le trafic entrant vers notre instance EC2. Cela permettra aux utilisateurs d'accéder à notre application via Internet.
3. Se Connecter à l'**Instance via SSH** : Enfin, nous nous connecterons à l'instance EC2 en utilisant **SSH (Secure Shell)**, un protocole de connexion sécurisé basé sur le terminal. Une fois connectés, nous pourrons exécuter des commandes sur cette machine distante, notamment pour installer Docker, puis tirer et exécuter notre conteneur.

Déploiement vers AWS EC2

EC2 est un service proposé par **AWS** qui vous permet de lancer vos propres machines d'hébergement distantes.

1

Créer et Lancer une Instance EC2, VPC et le groupe de sécurité.

2

Configurer le groupe de sécurité afin d'exposer les ports nécessaires au service www

3

Se connecter via SSH, installer Docker et lancer le conteneur



Ces étapes permettront de faire fonctionner notre application conteneurisée avec l'aide d'AWS.

11.4.2. Préparation de l'Application conteneurisée

Avant de commencer, il nous faut une application conteneurisée. À cet effet, je vous ai fourni une simple application de démonstration, qui est une application **Node.js** servant un fichier HTML de bienvenue, le fichier `welcome.html`, en réponse aux requêtes entrantes sur le **port 80**.

J'ai déjà créé un fichier Docker pour cette application. Il s'agit d'un **Dockerfile** standard, sans complexité particulière.

Voici comment procéder :

1. **Construire l'Image Docker** : Sur votre machine locale, naviguez dans le dossier du projet où se trouve le **Dockerfile**, puis utilisez la commande suivante pour construire l'image Docker :

```
docker build -t node-dep-example .
```

(Ici, `node-dep-example` est le nom donné à l'image, indiquant qu'il s'agit d'un exemple de déploiement Node.)



Nous utilisons la version **Alpine** de l'image Node, car elle est plus légère, ce qui

accélère le processus de construction et de déploiement.

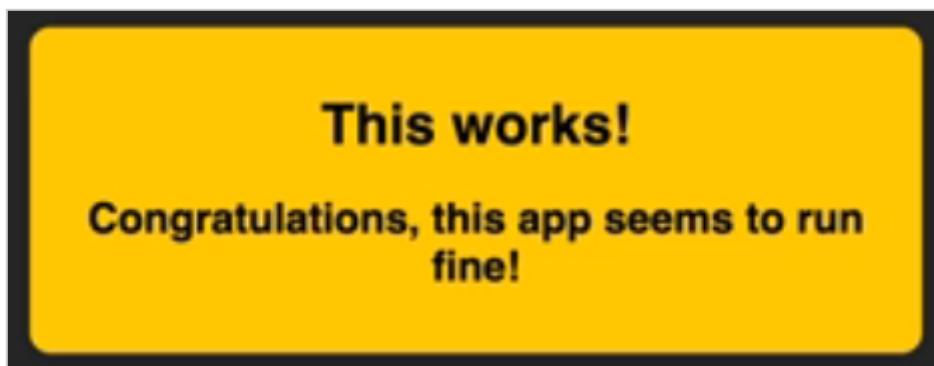
2. **Exécuter le Conteneur** : Une fois l'image construite, vous pouvez exécuter un conteneur basé sur cette image :

```
docker run -d --rm --name node-dep -p 80:80 node-dep-example
```

- **-d** exécute le conteneur en mode détaché.
- **--rm** le supprime automatiquement s'il est arrêté.
- **--name node-dep** donne un nom au conteneur.
- **-p 80:80** expose le **port 80** de l'hôte au port **80** du conteneur.

3. **Tester l'Application** : Une fois le conteneur démarré, ouvrez un navigateur et accédez à <http://localhost>.

Vous devriez voir la page de bienvenue de cette application de démonstration.



11.5. Différence entre le Développement et la Production avec Docker

Nous allons aborder une distinction importante entre l'utilisation de Docker en mode **développement** et en mode **production**, en particulier en ce qui concerne l'utilisation des volumes et des bind mounts.

11.5.1. Mode Développement

Lorsque nous travaillons sur une application en mode développement, le conteneur doit encapsuler l'environnement d'exécution nécessaire (runtime), mais pas nécessairement le code source de l'application. L'idée est que le conteneur contienne tous les outils requis pour exécuter l'application, mais le code source lui-même peut provenir de l'extérieur du conteneur.



Cela permet au conteneur de récupérer les dernières modifications du code sans nécessiter de **reconstruire l'image ou de redémarrer le conteneur**.

Pour faciliter cela, nous utilisons des **bind mounts** pendant le développement. Un bind mount permet de **lier un dossier de projet sur la machine locale à un dossier à l'intérieur du**

conteneur en cours d'exécution. De cette manière, le conteneur a toujours accès à la version la plus récente du code du projet, et toute modification apportée au code est immédiatement reflétée dans le conteneur sans besoin de redémarrage. Cette fonctionnalité est très pratique en développement, car elle permet des mises à jour instantanées et améliore l'efficacité du processus de développement.

11.5.2. Mode Production

En production, les choses sont différentes. Une fois que nous déplaçons notre image et notre conteneur sur une machine distante pour rendre l'application accessible aux utilisateurs du monde entier, le conteneur doit fonctionner de manière autonome et ne doit dépendre d'aucune configuration externe sur la machine hôte distante. L'**image Docker**, et donc le **conteneur** basé sur cette image, doit être la source unique de vérité.

Cela signifie que le conteneur doit contenir tout ce dont l'application a besoin pour fonctionner, y compris le code source. Il ne doit pas être nécessaire de déplacer des fichiers de code source dans des dossiers spécifiques sur la machine distante. Faire cela irait à l'encontre du principe des conteneurs, qui vise à éviter toute dépendance envers la configuration de la machine hôte.

Pour cette raison, lors de la création d'images pour la production, nous utilisons la commande **COPY** au lieu des **bind mounts**. En utilisant **COPY**, nous intégrons le code source directement dans l'image Docker lors de sa construction. Ainsi, l'image construite contient à la fois le code source et l'environnement nécessaire pour exécuter l'application. Cela garantit que l'image possède tout ce qu'il faut pour faire fonctionner l'application, indépendamment de l'environnement d'hébergement.

11.5.3. Flexibilité entre Développement et Production

Il est également important de noter que nous pouvons toujours utiliser des **bind mounts** pendant le développement, même si le code source est copié dans l'image pour la production.

Les **bind mounts** ne sont pas configurés dans le **Dockerfile**, mais via la commande docker run avec l'option **-v**. Cela nous permet de différencier facilement les environnements de développement et de production en utilisant simplement des commandes différentes pour exécuter le conteneur, sans avoir à modifier le **Dockerfile**.

Cela nous offre une flexibilité accrue en développement tout en maintenant une cohérence entre les environnements.

11.5.4. Conclusion

Bind mounts, Volumes et Copy

En développement

Les conteneurs doivent encapsuler l'environnement (runtime) mais pas nécessairement le code

Utiliser « Bind Mounts » pour fournir les fichiers du projet depuis la machine hôte locale vers le conteneur

Permet des mises à jour de l'application dans le conteneur sans avoir à le redémarrer ou recréer son image

En production

Image/Conteneur sont la « seule source de vérité »

Un conteneur doit être totalement indépendant, le code source de l'application ne DOIT PAS être sur la machine distante

Utiliser COPY pour copier un instantané du code source dans l'image

S'assurer que chaque image fonctionne sans extra, sans configuration ou code

En résumé, pour les projets utilisant un seul conteneur et une seule image, la commande **COPY** est cruciale pour s'assurer que le code et l'environnement sont correctement encapsulés dans l'image. Cela garantit que chaque image peut s'exécuter sans configuration ou code supplémentaire, quel que soit l'endroit où elle est déployée. Ce principe est essentiel pour maintenir la portabilité et la fiabilité des applications conteneurisées.

11.6. Déploiement d'une Application de Démonstration sur un Serveur Web avec AWS EC2

11.6.1. Mise en place

Dans cette section, nous allons voir comment prendre une application de démonstration que nous avons développée localement et la déployer sur un serveur web afin qu'elle soit accessible par n'importe quel utilisateur, et pas seulement depuis notre machine de développement locale.

- Étape 1 : Construire l'Image Docker

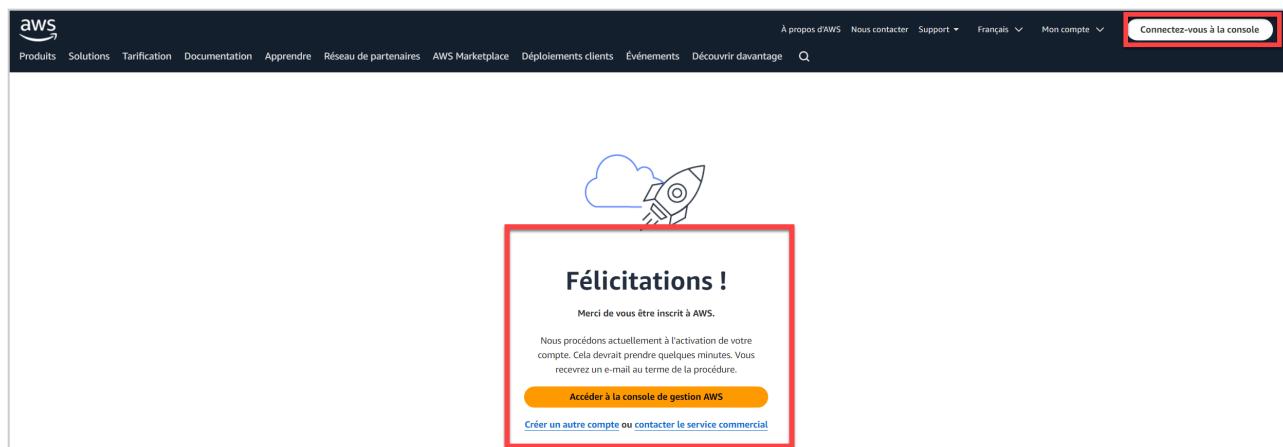
Jusqu'à présent, nous avons simplement construit une image Docker pour notre application, ce que nous avons fait à plusieurs reprises tout au long de ce cours. La construction de l'image est une étape essentielle, surtout lorsque nous apportons des modifications à notre code source. Chaque fois que nous modifions le code, nous devons reconstruire l'image pour inclure ces changements.

• Étape 2 : Exécuter le Conteneur sur une Machine Distante

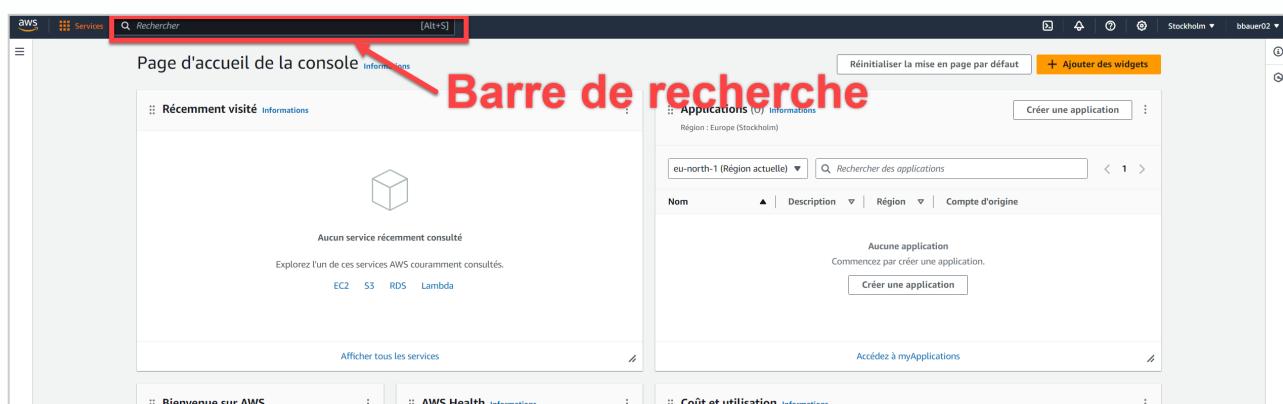
Une fois que nous avons l'image prête, l'objectif est de ne pas exécuter le conteneur sur notre machine locale, mais plutôt sur un serveur distant. Cela permet à l'application d'être accessible par tous les utilisateurs sur Internet. Pour ce faire, nous allons utiliser **AWS (Amazon Web Services)** pour créer cette machine distante, sur laquelle nous pourrons exécuter notre conteneur.

Création de la Machine Distante sur AWS:

- Créer un Compte AWS :** Pour commencer, vous devrez créer un compte AWS et vous y connecter. *Vous aurez besoin d'une carte de crédit pour configurer votre compte, même si ce que nous allons faire ici est couvert par l'offre gratuite d'AWS (Free Tier), donc normalement cela ne devrait pas générer de coûts.*



- Accéder à la Console de Gestion AWS :** Une fois connecté à votre compte AWS, vous accéderez à la console de gestion AWS (AWS Management Console). L'interface peut légèrement varier au fil du temps, mais vous devriez voir une **barre de recherche** où vous pouvez rechercher des services AWS spécifiques.



- Recherche du Service EC2 :** AWS propose une multitude de services et de produits, il est donc

courant d'utiliser la barre de recherche pour trouver le service que vous souhaitez utiliser. Dans notre cas, recherchez EC2, qui est le service permettant de créer des serveurs virtuels dans le cloud (c'est-à-dire, nos propres ordinateurs dans le cloud).

4. **Lancer une Nouvelle Instance EC2 :** Une fois que vous êtes sur la page **EC2**, vous verrez une option pour lancer une nouvelle instance. Cliquez sur le bouton « Lancer une instance », ce qui signifie essentiellement démarrer un nouvel ordinateur basé sur le cloud. Cela vous permettra de configurer un nouveau serveur où vous pourrez ensuite déployer votre application Docker.

En suivant ces étapes, vous serez en mesure de configurer une machine distante via AWS, d'y installer Docker, et d'exécuter votre conteneur d'application, rendant ainsi votre application accessible sur le web.

11.6.2. Lancement d'une Instance EC2 et Connexion via SSH

Lorsque vous lancez une nouvelle instance **EC2** sur **AWS**, un assistant de configuration s'ouvre pour vous guider dans la création de votre serveur cloud.

Voici les étapes pour configurer et se connecter à **votre instance EC2** :

Images d'applications et de systèmes d'exploitation (Amazon Machine Image) :

1. **Sélectionnez l'AMI d'Amazon Linux :** L'AMI (Amazon Machine Image) est l'image du système d'exploitation qui sera installée sur votre serveur. Choisissez l'option **Amazon Linux AMI en version 64 bits (x86)**. Cliquez sur "Select" pour valider ce choix.

2. Choisir une **Instance Éligible à l'Offre Gratuite** : Assurez-vous de sélectionner une option éligible à l'offre gratuite, comme l'instance **t3.micro** disponible au moment où je rédige ce cours. Cette option est idéale pour les tests et les démonstrations, car elle est gratuite pour commencer et offre suffisamment de ressources pour des tâches basiques.

▼ **Images d'applications et de systèmes d'exploitation (Amazon Machine Image)**

[Informations](#)

Une AMI est un modèle contenant la configuration logicielle (système d'exploitation, serveur d'applications et applications) requise pour lancer votre instance. Parcourez ou recherchez des AMI si vous ne trouvez pas ce que vous recherchez ci-dessous.

Effectuer une recherche dans notre catalogue complet, qui comprend des milliers d'images d'applications et de sy

Démarrage rapide

Amazon Linux 2023 (Éligible à l'offre gratuite)

macOS Ubuntu Windows Red Hat SUSE Li

Y compris les AMI d'AWS, de Marketplace et de la communauté

Amazon Machine Image (AMI)

AMI Amazon Linux 2023
ami-090abff6ae1141d7d (64 bits (x86), uefi-preferred) / ami-0a44bcbb8b18f238d (64 bits (Arm), uefi)
Virtualisation: hvm ENA activé: true Type de périphérique racine: ebs

Description

Amazon Linux 2023 est un système d'exploitation moderne basé sur Linux, à usage général et offrant cinq ans de support garanti. Optimisé pour AWS, il est conçu afin de fournir un environnement d'exécution sécurisé, stable et à hautes performances pour le développement et l'exécution de vos applications cloud.

Architecture: 64 bits (x86)
Mode de démarrage: uefi-preferred
ID AMI: ami-090abff6ae1141d7d
Fournisseur vérifié

Type d'instance :

1. **Vérifier la Création d'un VPC par Défaut** : Un VPC (Virtual Private Cloud) par défaut doit être sélectionné. Si ce n'est pas le cas, créez un nouveau VPC. Laissez tous les autres paramètres par défaut à moins que vous ne sachiez ce que vous faites.

Type d'instance [Informations](#) | [Obtenez des conseils](#)

Type d'instance

t3.micro	Éligible à l'offre gratuite		
Famille: t3	2 vCPU	1 Gio Mémoire	Génération actuelle: true
À la demande RHEL base tarification:	0.0396 USD par heure		
À la demande SUSE base tarification:	0.0108 USD par heure		
À la demande Linux base tarification:	0.0108 USD par heure		
À la demande Windows base tarification:	0.02 USD par heure		

Des frais supplémentaires s'appliquent pour les AMI avec un logiciel préinstallé

Toutes les générations

[Comparer les types d'instance](#)

Configurer les Détails de l'Instance et paramètres réseau :

Paramètres réseau [Informations](#)

Réseau [Informations](#) vpc-003d531ccb3b158e9

Sous-réseau [Informations](#)

Aucune préférence (sous-réseau par défaut dans n'importe quelle zone de disponibilité)

Attribuer automatiquement l'adresse IP publique [Informations](#)

Activer

Des frais supplémentaires s'appliquent en cas de dépassement de la [limite de l'offre gratuite](#)

Pare-feu (groupes de sécurité) [Informations](#)

Un groupe de sécurité est un ensemble de règles de pare-feu qui contrôlent le trafic de votre instance. Ajoutez des règles pour autoriser un trafic spécifique à atteindre votre instance.

Créer un groupe de sécurité Sélectionner un groupe de sécurité existant

Nous allons créer un nouveau groupe de sécurité appelé « **launch-wizard-1** » avec les règles suivantes :

Autoriser le trafic SSH depuis N'importe où 0.0.0.0/0

Autoriser le trafic HTTPS depuis l'Internet

Autoriser le trafic HTTP depuis l'Internet

⚠️ Les règles avec la source 0.0.0.0/0 autorisent toutes les adresses IP à accéder à votre instance. Nous vous recommandons de définir des règles de groupe de sécurité pour autoriser l'accès à partir d'adresses IP connues uniquement.

Nombre d'instances [Informations](#) 1

Image logicielle (AMI)

Amazon Linux 2 Kernel 5.10 AMI...en savoir plus
ami-0768a36c1787a447e

Type de serveur virtuel (type d'instance)

t3.micro

Pare-feu (groupe de sécurité)

Nouveau groupe de sécurité

Stockage (volumes)

1 volume(s) - 8 Gio

Offre gratuite : La première année inclut 750 heures d'utilisation mensuelle des instances t2.micro (ou t3.micro dans les régions où t2.micro n'est pas disponible) sur les AMI de l'offre gratuite, 750 heures d'utilisation d'adresses IPv4 publiques par mois, 30 Go de stockage EBS, 2 millions d'I/O, 1 Go d'instantanés et 100 Go de bande passante vers Internet

Annulez Lancer l'instance Examiner les commandes

Créer une Clé SSH pour la Connexion :

Paire de clés (connexion) [Informations](#)

Vous pouvez utiliser une paire de clés pour vous connecter en toute sécurité à votre instance. Assurez-vous d'avoir accès à la paire de clés sélectionnée avant de lancer l'instance.

Nom de la paire de clés - *obligatoire*

Sélectionnez Créer une paire de clés

Créer une Nouvelle Paire de Clés :

Une paire de clés **SSH** est nécessaire pour se connecter à votre instance **EC2** de manière sécurisée. Sélectionnez "Créer une paire de clés", donnez-lui un nom (par exemple, "example-1"), sélectionnez une paire de type **ED25519** (plus sûre) au format **.ppk**, puisque nous utiliserons **PUTTY**. puis cliquez sur "Créer une paire de clés".

Il est essentiel de télécharger cette clé, car elle ne sera disponible qu'une seule fois. Si vous perdez cette clé, **vous devrez arrêter votre instance et en créer une nouvelle, parce que vous ne pourrez plus vous y connecter.**



Ne partagez pas cette clé avec d'autres personnes, car quiconque possède ce fichier pourra se connecter à votre machine distante.

Cliquez sur "Lancer l'instance" pour démarrer votre instance virtuelle.

Ensuite, cliquez sur "View Instances" pour afficher l'état de vos instances. Le lancement de l'instance peut prendre quelques minutes.

Liste des instances lancées :

Instances (1) Informations		Last updated 1 minute ago	G	Se connecter	État de l'instance ▾	Actions ▾	Lancer des instances ▾
	Rechercher Instance par attribut ou identification (case-sensitive)	Tous les états ▾	< 1 > ⏪				
<input type="checkbox"/> Name ↴	ID d'instance ↴	État de l'insta... ↴	Type d'insta... ↴	Contrôle des statuts	Statut d'alarme Zone de dispon... DNS IPv4 public Adresse IPv4... IP élastique	Afficher les alarme eu-north-1b ec2-13-53-76-56.eu-no... 13.53.76.56 -	
<input type="checkbox"/> Example-1	i-02957cde28702a50b	En cours d'... ↴	t3.micro	Initialisation en cours			

Connexion à l'Instance via SSH

- Utiliser SSH pour se Connecter** : Une fois l'instance en cours d'exécution, vous utiliserez **SSH (Secure Shell)** pour vous connecter. SSH est un protocole qui vous permet de vous connecter à une machine distante depuis votre machine locale via la ligne de commande.
 - Sur Linux et macOS** : Vous pouvez utiliser le terminal intégré pour exécuter des commandes SSH.
 - Sur Windows** : Vous pouvez soit installer **WSL 2 (Windows Subsystem for Linux)** pour exécuter Linux sur Windows, soit utiliser un client SSH comme **PUTTY**. **PUTTY** est une application que vous pouvez télécharger et installer pour exécuter des commandes **SSH** sur une machine Windows.
- Connexion à l'Instance** : Dans le tableau de bord EC2 d'AWS, cliquez sur "Instance", et

sélectionnez l'instance que nous avons créé puis cliquez sur le bouton "Se connecter" ou click droit + Se connecter

"Click droit" sur l'instance

Puis choisissez l'onglet "Client SSH". AWS vous fournira les étapes détaillées pour établir la connexion avec votre instance EC2.

Connectez-vous à l'instance Informations

Connectez-vous à votre instance à i-02957cde28702a30b (Example-1) l'aide de l'une de ces options

- [EC2 Instance Connect](#)
- [Session Manager](#)
- [Client SSH](#)
- [EC2 Serial Console](#)

ID d'instance

[i-02957cde28702a30b \(Example-1\)](#)

1. Ouvrez un client SSH.
2. Recherchez votre fichier de clé privée. La clé utilisée pour lancer cette instance est example-1.pem
3. Exécuter, si nécessaire, cette commande pour vous assurer que votre clé n'est pas visible publiquement.

DNS public copié ["example-1.pem"](#)

... connectez-vous à votre instance à l'aide de son DNS public :

[ec2-13-53-76-56.eu-north-1.compute.amazonaws.com](#)

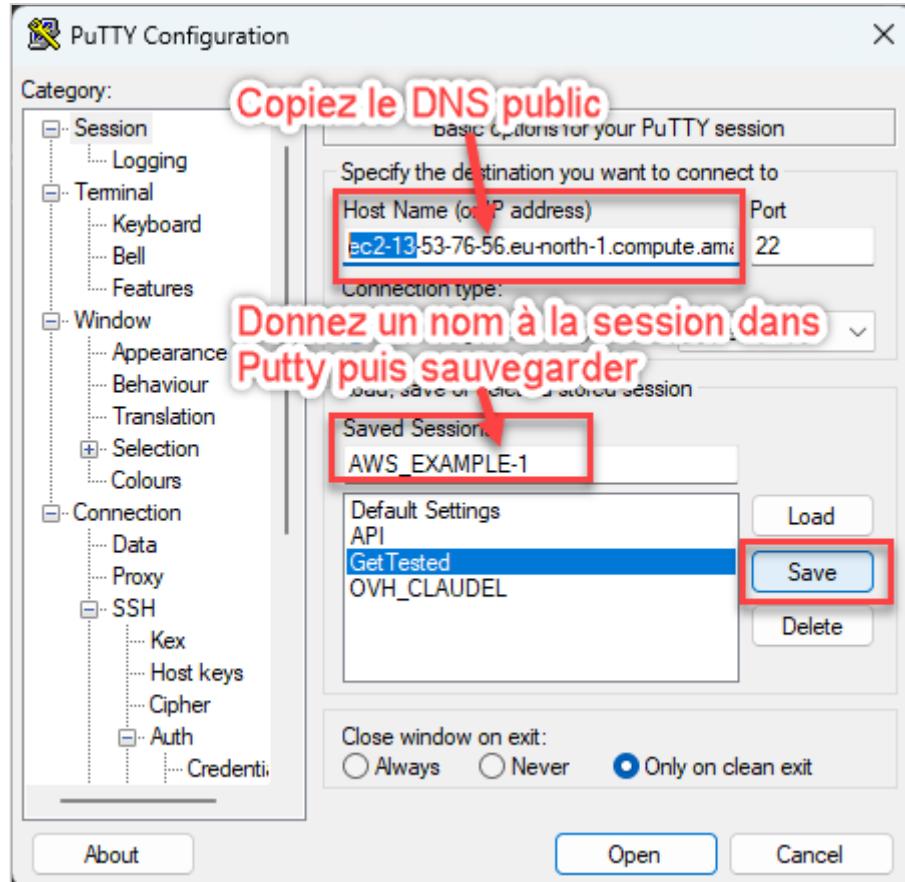
Exemple :

[ssh -i "example-1.pem" ec2-user@ec2-13-53-76-56.eu-north-1.compute.amazonaws.com](#)

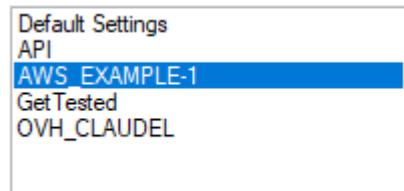
Remarque : Dans la plupart des cas, le nom d'utilisateur deviné est correct. Cependant, lisez les instructions d'utilisation de l'AMI pour vérifier si le propriétaire de l'AMI a modifié le nom d'utilisateur par défaut.

[Annulez](#)

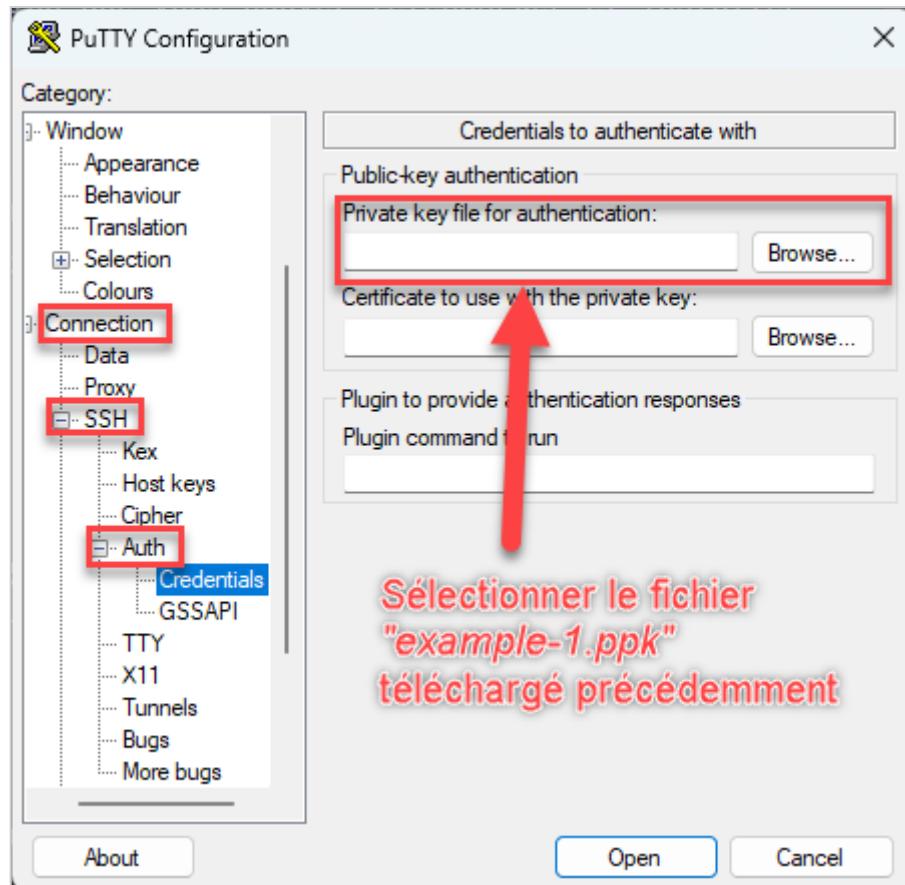
Ouvrez maintenant Putty :



Une fois la session sauvegardée, sélectionnez-la, mais ne cliquez pas encore sur "OPEN" :



Il faut ajouter dans PUTTY la clé SSH reçue :



Cliquez sur "OPEN", et saisissez le nom d'utilisateur fourni par AWS, pour ma part, il s'agit de :
ec2-user

Connectez-vous à l'instance Informations

Connectez-vous à votre instance à i-02957cde28702a30b (Example-1) l'aide de l'une de ces options

EC2 Instance Connect

Session Manager

Client SSH

EC2 Serial Console



Le port 22 (SSH) est ouvert à toutes les adresses IPv4

Le port 22 (SSH) est actuellement ouvert à toutes les adresses IPv4, indiquées par **0.0.0.0/0** dans la règle d'entrée de [votre groupe de sécurité](#). Pour une sécurité accrue, envisagez de restreindre l'accès aux seules adresses IP du service EC2 Instance Connect pour votre région : 13.48.4.200/30. [En savoir plus.](#)

ID d'instance

[i-02957cde28702a30b \(Example-1\)](#)

Type de connexion

Se connecter à l'aide d'EC2 Instance Connect

Connectez-vous à l'aide du client basé sur navigateur EC2 Instance Connect, avec une adresse IPv4 publique.

Se connecter à l'aide d'EC2 Instance Connect

Endpoint

Connectez-vous à l'aide du client basé sur navigateur EC2 Instance Connect, avec une adresse IPv4 privée et un point de terminaison VPC.

Adresse IP publique

[13.53.76.56](#)

Nom d'utilisateur pour le client SSH

Nom d'utilisateur

Saisissez le nom d'utilisateur défini dans l'AMI utilisée pour lancer l'instance. Si vous n'avez pas défini de nom d'utilisateur personnalisé, utilisez le nom d'utilisateur par défaut, ec2-user.

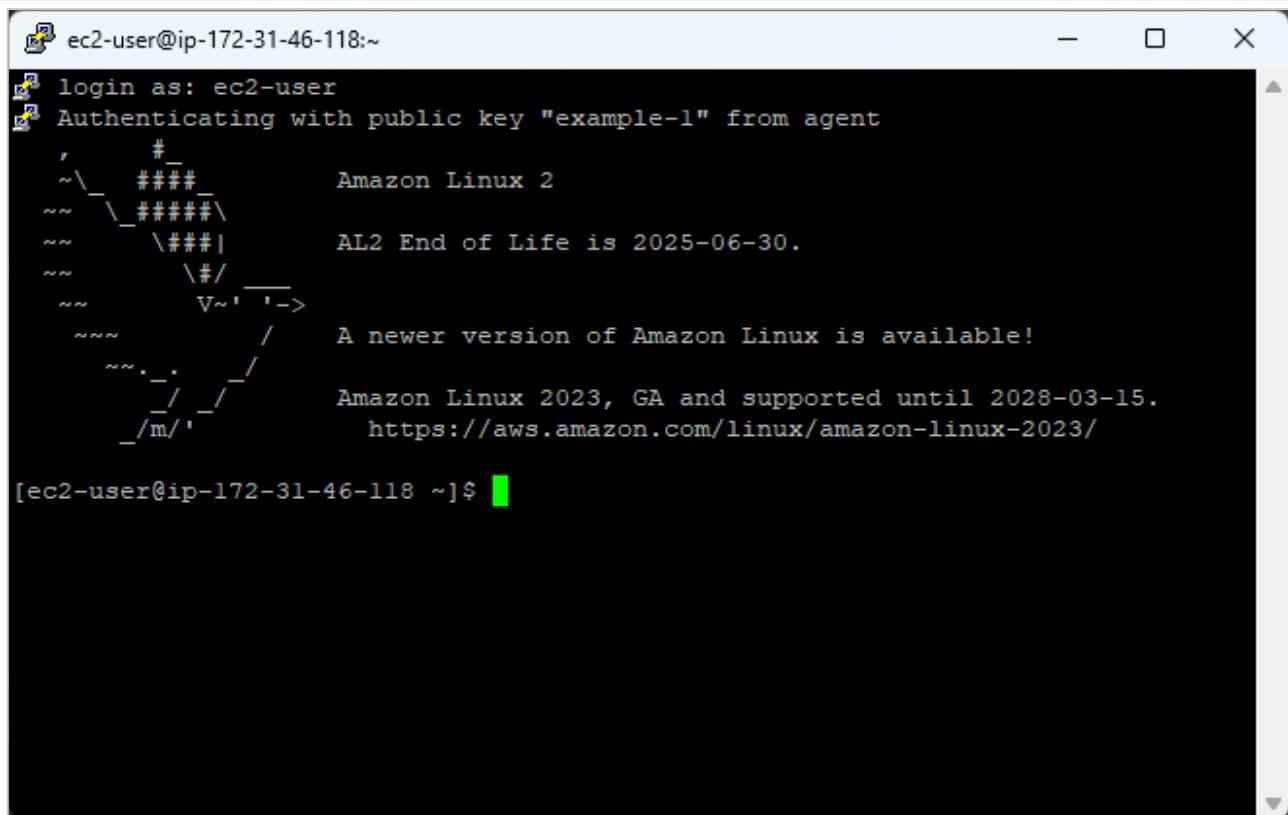
Remarque : dans la plupart des cas, le nom d'utilisateur par défaut, ec2-user, est correct. Cependant, lisez les instructions d'utilisation de l'AMI pour vérifier si le propriétaire de l'AMI a modifié le nom d'utilisateur par défaut.

Annulez

Se connecter

Vous devriez avoir cette fenêtre dans PUTTY qui s'affiche une fois authentifié :





The screenshot shows a terminal window titled "ec2-user@ip-172-31-46-118:~". The window displays the Amazon Linux 2 welcome screen, which includes a stylized logo composed of symbols like '#', '=', and 'v'. Text on the screen indicates the system is "Authenticating with public key 'example-1' from agent", the version is "Amazon Linux 2", the end-of-life date is "AL2 End of Life is 2025-06-30.", and a message about an "A newer version of Amazon Linux is available! Amazon Linux 2023, GA and supported until 2028-03-15." with a link "https://aws.amazon.com/linux/amazon-linux-2023/". The prompt "[ec2-user@ip-172-31-46-118 ~]\$" is visible at the bottom.

Installer Docker sur l'Instance :

+ Une fois connecté à votre instance **EC2 via SSH**, toutes les commandes que vous exécuterez le seront sur la machine distante. Vous pouvez maintenant procéder à l'installation de Docker, transférer votre image Docker sur l'instance et exécuter votre conteneur. Vous configurerez également les groupes de sécurité pour permettre l'accès approprié.

En suivant ces étapes, vous aurez configuré avec succès une instance EC2, créé une connexion sécurisée via SSH, et préparé votre environnement pour le déploiement d'applications conteneurisées.

11.6.3. Installation de Docker sur l'Instance EC2

Maintenant que nous sommes connectés à notre instance EC2 via SSH, nous allons procéder à l'installation de Docker. Voici les étapes à suivre pour configurer Docker sur votre machine distante :

- **Étape 1 : Mise à Jour des Paquets Essentiels**

Dans le terminal connecté PUTTY (qui exécute des commandes sur la machine distante), commencez par exécuter la commande suivante pour mettre à jour tous les paquets essentiels de la machine :

```
sudo yum update -y
```

Cette commande garantit que tous les paquets essentiels de la machine distante sont mis à jour et utilisent leur version la plus récente.

- **Étape 2 : Installation de Docker**

Ensuite, exécutez la commande suivante pour installer **Docker** :

```
sudo yum -y install docker
```

Cette commande installe Docker sur la machine distante. Bien que nous ayons installé Docker sur notre machine locale lors des premières sections du cours, cela ne nous aide pas si nous voulons exécuter un conteneur sur la machine distante. Il est donc nécessaire d'installer Docker sur l'instance EC2 également.

Confirmez l'installation en tapant **y** lorsque vous y êtes invité. Docker et tous les outils nécessaires à son fonctionnement seront installés sur la machine.

- **Étape 3 : Démarrer le Service Docker**

Une fois l'installation terminée, démarrez le service Docker en exécutant la commande suivante :

```
sudo service docker start
```

Puis :

```
sudo usermod -a -G docker ec2-user
```

Cette commande ajoute l'utilisateur **ec2-user** au groupe **docker**. Cela permet à l'utilisateur **ec2-user** d'exécuter des commandes Docker sans avoir besoin de préfixer chaque commande avec **sudo**. En d'autres termes, cela permet à **ec2-user** d'utiliser Docker avec des privilèges d'administration (root), mais sans avoir besoin d'accéder à root directement.

Il faut maintenant se déconnecter :

```
logout
```

Puis se reconnecter avec PUTTY et saisir la commande :

```
sudo systemctl enable docker
```

Vérifions si Docker est correctement lancé :

```
docker version
```

Vous devriez avoir cette réponse du prompt :

```
ec2-user@ip-172-31-46-118:~$ 
  _\ _###_      Amazon Linux 2
  ~ \_\####\     AL2 End of Life is 2025-06-30.
  ~ \#\#|      V~'__>
  ~ \#/       /   A newer version of Amazon Linux is available!
  ~ .-'      /   Amazon Linux 2023, GA and supported until 2028-03-15.
  _/_/      /   https://aws.amazon.com/linux/amazon-linux-2023/
/m/ '      / 

[ec2-user@ip-172-31-46-118 ~]$ sudo systemctl enable docker
Created symlink from /etc/systemd/system/multi-user.target.wants/docker.service
to /usr/lib/systemd/system/docker.service.
[ec2-user@ip-172-31-46-118 ~]$ docker version
Client:
Version:          25.0.5
API version:      1.44
Go version:       go1.22.5
Git commit:       5dc9bcc
Built:            Thu Aug 22 17:25:26 2024
OS/Arch:          linux/amd64
Context:          default

Server:
Engine:
Version:          25.0.6
API version:      1.44 (minimum version 1.24)
Go version:       go1.22.5
Git commit:       b08a51f
Built:            Thu Aug 22 17:26:01 2024
OS/Arch:          linux/amd64
Experimental:    false
containerd:
Version:          1.7.20
GitCommit:        8fc6bcff51318944179630522a095cc9dbf9f353
runc:
Version:          1.1.13
GitCommit:        58aa9203c123022138b22cf96540c284876a7910
docker-init:
Version:          0.19.0
GitCommit:        de40ad0
[ec2-user@ip-172-31-46-118 ~]$ 
```

11.6.4. Déploiement de l'Image Docker sur une Machine Distante

Maintenant que Docker est installé et fonctionne sur la machine distante, nous devons transférer notre image Docker locale vers cette machine. Pour ce faire, nous avons deux options principales :

- 1. Option 1 : Déployer le Code Source** : Cette approche consiste à copier notre code source complet (tout ce qui se trouve dans le dossier du projet, y compris le fichier Dockerfile) sur la machine distante et à y construire l'image Docker directement. Cela implique d'utiliser les commandes docker build pour créer l'image et docker run pour exécuter le conteneur sur la machine distante.
- 2. Option 2 : Construire l'Image en Local et la Déployer** : L'autre approche consiste à construire l'image Docker en local, sur notre propre machine, puis à déployer cette image pré-construite sur la machine distante. Cela simplifie le processus, car l'image est déjà prête et nous n'avons

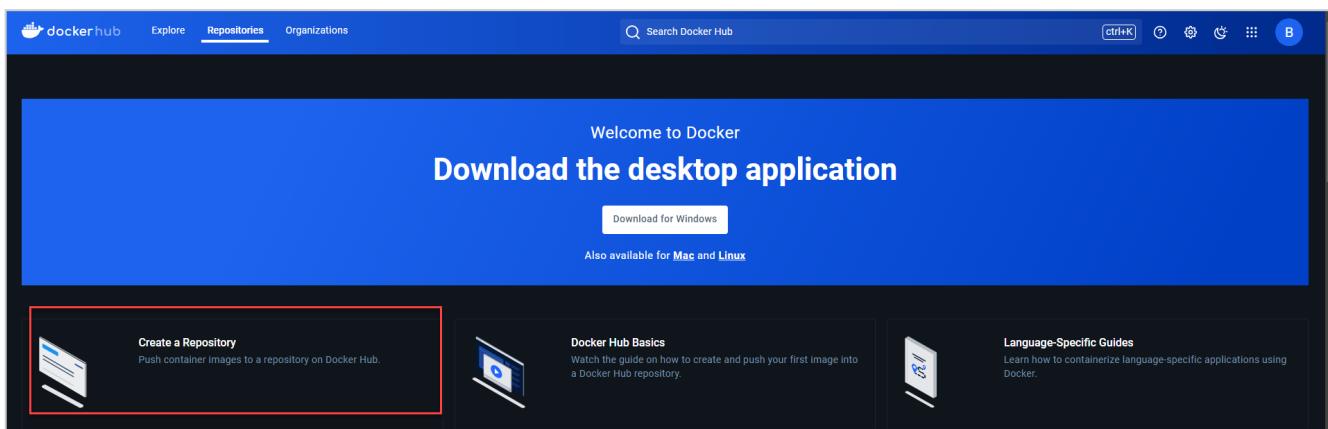
pas besoin de refaire tout le travail de construction sur la machine distante.

Choisir la Meilleure Option

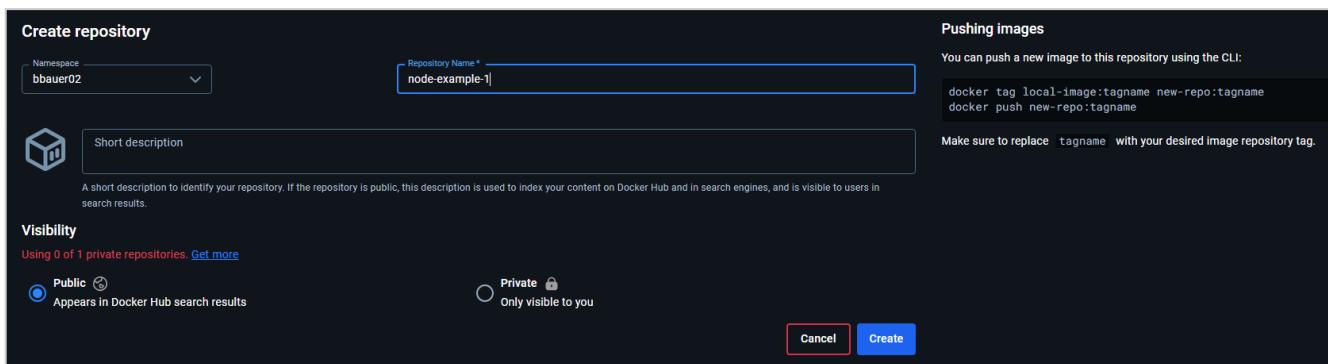
La première option ajoute une complexité inutile, car elle nécessite de reconstruire l'image sur la machine distante sans réelle valeur ajoutée. Il est plus efficace de préparer l'image en local, puis de la transférer sur la machine distante. Nous pouvons utiliser **Docker Hub** pour cela : construire l'image en local, la pousser sur Docker Hub, ensuite la récupérer depuis le serveur distant pour l'exécuter.

Étapes pour Déployer avec Docker Hub

- **Créer un Compte sur Docker Hub:** Connectez-vous à votre compte Docker Hub. Si vous avez suivi les étapes précédentes du cours, vous avez déjà configuré ce compte.
- **Créer un Nouveau dépôt :** Sur Docker Hub, cliquez sur "*Create Repository*"

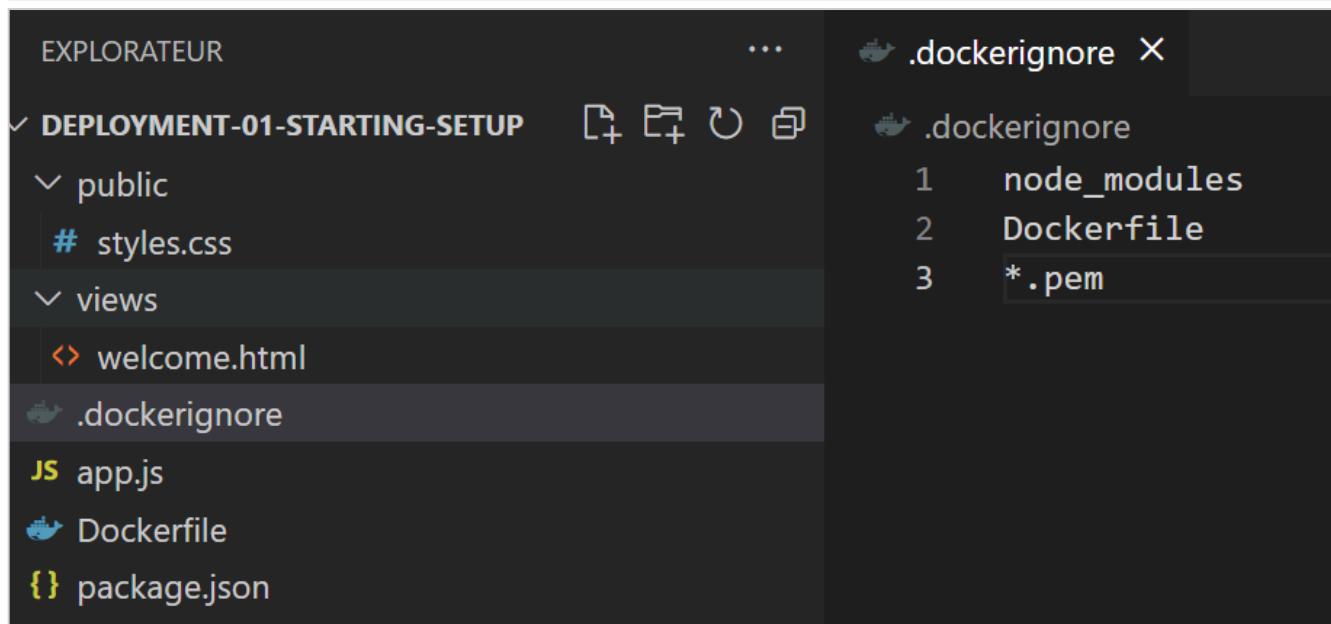


Et nommez votre nouveau dépôt (par exemple, *node-example-1*). Pour des raisons de coût et de simplicité, vous pouvez définir le dépôt comme **public**.



- **Construire l'Image en Local**

Ouvrez un terminal local (non connecté à votre serveur distant) et assurez-vous d'être dans le dossier de votre projet. Avant de construire l'image, créez un fichier `.dockerignore` pour exclure les fichiers et dossiers que vous ne voulez pas inclure dans l'image, tels que `node_modules`, `Dockerfile`, et toute clé `.pem` sensible.



```
EXPLORATEUR ... .dockerignore X
✓ DEPLOYMENT-01-STARTING-SETUP ⌂ ⌃ ⌚ ⌚
  ✓ public
    # styles.css
  ✓ views
    < welcome.html
  .dockerignore
  JS app.js
  Dockerfile
  {} package.json
```

Ensuite, utilisez la commande suivante pour construire l'image :

```
docker build -t node-dep-example-1 .
```

- **Taguer et Pousser l'Image vers Docker Hub**

Une fois l'image construite, taguez-la avec le nom de votre dépôt sur **Docker Hub** :

Exemple avec l'utilisateur Docker Hub : bbauer02

```
docker tag node-dep-example-1 bbauer02/node-example-1
```



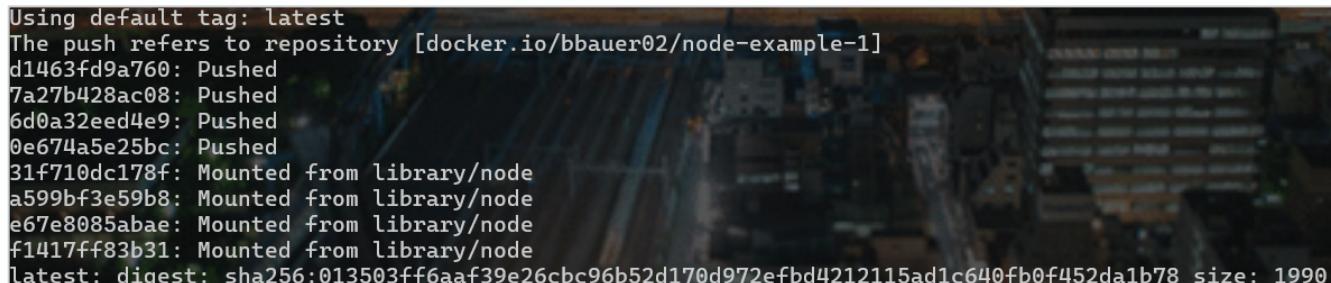
Remplacez **bbauer02** par votre propre nom d'utilisateur **Docker Hub**.

Puis, connectez-vous à Docker Hub (si ce n'est pas déjà fait) :

```
docker login
```

Poussez l'image vers Docker Hub :

```
docker push bbauer02/node-example-1
```



```
Using default tag: latest
The push refers to repository [docker.io/bbauer02/node-example-1]
d1463fd9a760: Pushed
7a27b428ac08: Pushed
6d0a32eed4e9: Pushed
0e674a5e25bc: Pushed
31f710dc178f: Mounted from library/node
a599bf3e59b8: Mounted from library/node
e67e8085abae: Mounted from library/node
f1417ff83b31: Mounted from library/node
latest: digest: sha256:013503ff6aaf39e26cbc96b52d170d972efbd4212115ad1c640fb0f452da1b78 size: 1990
```



Assurez-vous d'utiliser votre nom de dépôt correct. Cela transférera l'image construite localement sur Docker Hub, où elle sera stockée.

• Récupérer et Exécuter l'Image sur la Machine Distante

Avec l'image maintenant stockée sur Docker Hub, vous pouvez vous connecter à votre machine distante via SSH et récupérer l'image depuis Docker Hub :

```
[ec2-user@ip-172-31-46-118 ~]$ docker pull bbauer02/node-example-1
Using default tag: latest
latest: Pulling from bbauer02/node-example-1
f56be85fc22e: Pull complete
8f665685b215: Pull complete
e5fca6c395a6: Pull complete
561cb69653d5: Pull complete
f2b3cfca7b44f: Pull complete
eb2c2a416ca5: Pull complete
a2c78fa3560e: Pull complete
ea10e1a6d03d: Pull complete
Digest: sha256:013503ff6aaaf39e26cbc96b52d170d972efbd4212115ad1c640fb0f452da1b78
Status: Downloaded newer image for bbauer02/node-example-1:latest
docker.io/bbauer02/node-example-1:latest
```

Depuis le serveur distant nous récupérons l'image de notre application

Une fois l'image téléchargée sur la machine distante, vous pouvez l'exécuter en tant que conteneur.

11.6.5. Conclusion

En suivant ces étapes, vous avez appris comment construire une image Docker localement, la déployer sur un serveur distant via Docker Hub, et l'exécuter sur cette machine distante. Cela simplifie le processus de déploiement et permet de s'assurer que l'image utilisée en production est exactement la même que celle testée localement.

11.7. Exécution de l'Image Docker Poussée sur la Machine Distante (Instance EC2)

Maintenant que notre image Docker est poussée sur Docker Hub, il est temps de l'exécuter sur notre machine distante, c'est-à-dire notre instance EC2. Voici les étapes à suivre pour lancer l'image sur cette instance et tester l'application.

Connexion à l'Instance EC2 et Exécution du Conteneur

- Reconnexion à l'Instance EC2** : Revenez à votre terminal connecté via SSH à l'instance EC2. Si votre connexion s'est interrompue, reconnectez-vous simplement en utilisant la même commande SSH.
- Exécuter le Conteneur** : Une fois connecté, utilisez la commande suivante pour exécuter l'image Docker que vous avez poussée sur Docker Hub :

```
docker run -d -p 80:80 bbauer02/node-example-1
```

```
[ec2-user@ip-172-31-46-118 ~]$ docker run -d -p 80:80 bbauer02/node-example-1
d8d1ee7fd15822a5b677clc79c3f0d04a790c2ee612d584ccdf5e20d49309c7
[ec2-user@ip-172-31-46-118 ~]$
```

- **-d** exécute le conteneur en mode détaché (en arrière-plan).
- **--rm** supprime automatiquement le conteneur lorsqu'il s'arrête.
- **-p 80:80** mappe le port 80 de l'instance EC2 au port **80** du conteneur, ce qui est essentiel pour permettre aux utilisateurs d'accéder à l'application via **HTTP**.
- Remplacez **bbauer02** par votre propre nom d'utilisateur **Docker Hub**.

Cela téléchargera l'image depuis Docker Hub sur la machine distante et démarrera un conteneur basé sur cette image.

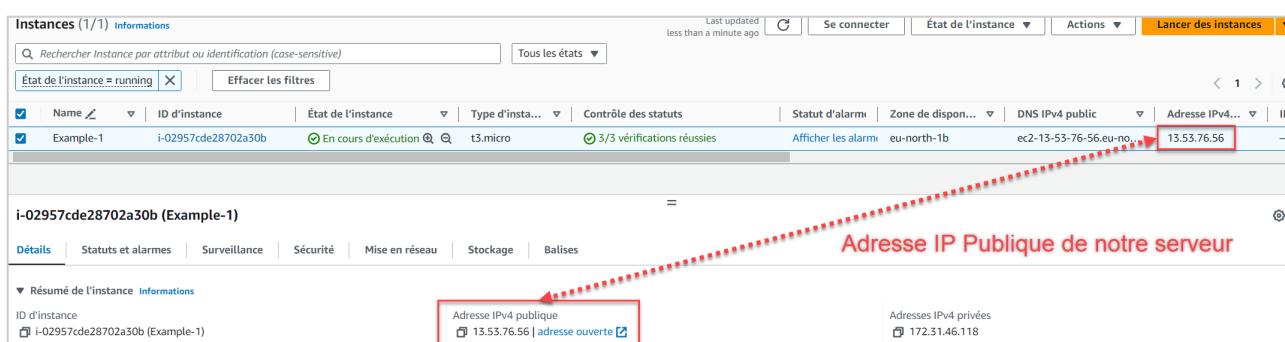
- Vérifier que le Conteneur Fonctionne** : Pour vérifier que le conteneur fonctionne correctement, exécutez la commande suivante :

```
sudo docker ps
```

Cette commande affichera une liste des conteneurs en cours d'exécution. Si votre conteneur apparaît dans la liste, cela signifie que votre application conteneurisée est en cours d'exécution sur l'instance distante.

Tester l'Application

- Obtenir l'Adresse IP Publique** : Retournez à la console AWS, accédez à votre instance EC2 en cours d'exécution, et notez l'adresse IP publique (**IPv4**) de l'instance.



Toutefois l'application n'est pas encore disponible, car il faudra modifier les groupes de sécurité.

- Configurer les Groupes de Sécurité** : Par défaut, votre instance EC2 bloque tout le trafic entrant, sauf les connexions SSH via le port 22.

Adresses IP I...	Surveillance	Nom du groupe de sécurité	Nom de clé	Heure de lancement
-	disabled	launch-wizard-1	example-1	2024/09/04 13:36 C

Notre instance est associé à un groupe de sécurité nommé ici par défaut : **launch-wizard-1**.

Dans le menu de gauche, cherchez la catégorie : "**Réseau et sécurité**" et cliquez sur "**Groupes**

de sécurité" :

Groupes de sécurité (2) Informations						
Nom du groupe de sécurité		ID de VPC	Description	Propriétaire	Actions	
sg-02b8f5b8f94427922	default	vpc-003d551ccb3b158e9	default VPC security group	640168428717	Actions	Créer un groupe de sécurité
sg-0532f5500f9d1ef0	launch-wizard-1	vpc-003d551ccb3b158e9	launch-wizard-1 created 2024-09-04T...	640168428717	Actions	Créer un groupe de sécurité

Par défaut, votre instance EC2 bloque tout le trafic entrant, sauf les connexions SSH via le port 22.

Règles entrantes (1)						
Gérer les balises Modifier les règles entrantes						
				Protocole	Plage de ports	Source
				TCP	22	0.0.0.0/0

Pour permettre le trafic HTTP (port 80), vous devez modifier les règles du groupe de sécurité :

Règles entrantes (1)						
Cliquez sur "Modifier les règles entrantes" pour ajouter de nouvelles règles de filtrage.						
				Protocole	Plage de ports	Source
				TCP	22	0.0.0.0/0

Puis saisissez les paramètres comme ci-dessous sans oublier d'enregistrer les modifications :

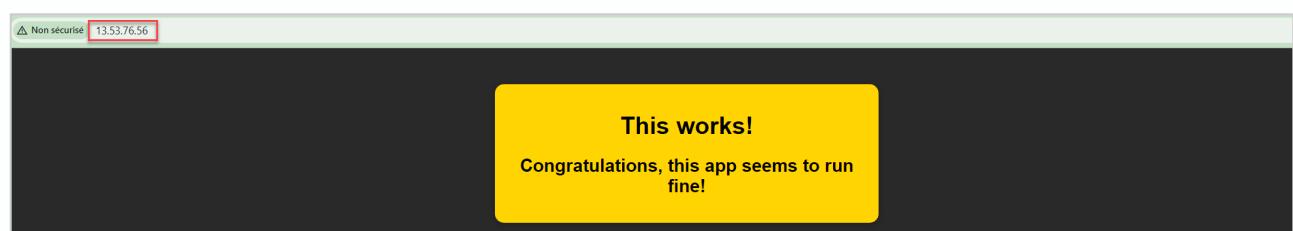
Modifier les règles entrantes Informations

Les règles entrantes contrôlent le trafic entrant qui est autorisé à atteindre l'instance.

On ajoute une autorisation sur les connexions entrantes HTTP provenant de toutes sources IP						
ID de règle de groupe de sécurité	Type	Protocole	Plage de ports	Source	Description - facultatif	
sgr-09981160072756bc1	SSH	TCP	22	Personnalisé(e) <input type="text" value="0.0.0.0/0"/> <input type="button" value="Supprimer"/>	<input type="button" value="Ajouter une règle"/>	
-	HTTP	TCP	80	N'importe où... <input type="text" value="0.0.0.0/0"/> <input type="button" value="Supprimer"/>	<input type="button" value="Enregistrer les modifications"/>	

⚠ Rules from source of 0.0.0.0/0 or ::/0 allow all IP addresses to access your instance. We recommend setting security group rules to allow access from known IP addresses only.

3. Accéder à l'Application via le Navigateur : Avec les règles de sécurité mises à jour, entrez l'adresse IP publique de votre instance EC2 dans votre navigateur web. Vous devriez voir un message indiquant :



Cela confirme que votre application Node.js conteneurisé fonctionne correctement sur une machine distante sans avoir besoin d'installer ou de configurer un environnement **Node.js** sur cette machine. Tout ce dont vous avez eu besoin était **Docker**.



Contrôle Complet : Vous avez maintenant un contrôle total sur cette instance EC2, ce qui signifie que vous pouvez exécuter n'importe quelle commande Docker, y

compris celles pour les applications multi-conteneurs avec Docker Compose.

Sécurité et Configuration : Bien que cette démonstration soit une introduction de base, pour un projet réel, vous devez approfondir la documentation AWS **pour sécuriser votre instance EC2 et comprendre comment gérer les règles de sécurité et les configurations réseau.**

Déploiement d'Applications Multi-conteneurs : Vous pouvez également utiliser Docker Compose pour déployer des applications plus complexes sur des machines distantes, bien que cela puisse nécessiter des ajustements dans la manière dont vous gérez les images et les configurations.

En suivant ces étapes, vous avez réussi à déployer et exécuter une application conteneurisée sur une instance EC2, démontrant ainsi le pouvoir de Docker pour le déploiement d'applications dans le cloud.

11.8. Gestion et mise à jour du conteneur/image

11.8.1. Mise à jour de l'image et de l'application

Nous venons de voir un premier exemple de déploiement d'une application avec Docker sur une instance EC2. Il est possible de faire bien plus, comme connecter un domaine personnalisé, mais pour cela, vous pouvez vous référer à la documentation officielle d'AWS. Cet exemple ne sera pas la méthode finale de déploiement que nous verrons dans ce module, car les applications plus complexes et les besoins spécifiques peuvent nécessiter des approches différentes. Néanmoins, ce premier exemple montre clairement pourquoi Docker est si utile, à la fois pour le développement et le déploiement.

Mettre à Jour le Code de l'Application

Pour terminer cet exemple, voyons comment mettre à jour le code sur le serveur distant et comment arrêter ou éteindre tout si nécessaire.

- Modification du Code Localement :** Faites un changement simple dans votre fichier HTML, par exemple, ajoutez des points d'exclamation après "This works!" dans le fichier `welcome.html`. Ce changement n'est pas automatiquement reflété sur le serveur distant, car il n'a été fait que localement.

`welcome.html`

```
...
<h1>This works !!!!!!!</h1>
...
```

- Reconstruction et "push" de l'Image :** Pour mettre à jour le serveur distant, il suffit de reconstruire l'image Docker localement, de la pousser à nouveau sur **Docker Hub**, puis d'utiliser cette image mise à jour sur le serveur distant.

Sur votre machine locale, ouvrez un terminal et reconstruisez l'image :

```
sudo docker build -t node-dep-example-1 .
```

Taggez l'image correctement et poussez-la sur Docker Hub :

```
sudo docker tag node-dep-example-1 bbauer02/node-example-1
sudo docker push bbauer02/node-example-1
```



Seules les couches modifiées seront poussées, ce qui optimise le processus de mise à jour.

3. Arrêter et Redémarrer le Conteneur sur le Serveur Distant :

- Revenez au terminal connecté à l'instance EC2.
- Listez tous les conteneurs en cours d'exécution et arrêtez le conteneur actuel :

```
sudo docker ps
sudo docker stop <nom_du_conteneur>
```

```
[ec2-user@ip-172-31-46-118 ~]$ sudo docker stop elated_mcclintock
[ec2-user@ip-172-31-46-118 ~]$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
[ec2-user@ip-172-31-46-118 ~]$ Le conteneur est arrêté
[ec2-user@ip-172-31-46-118 ~]$
```

- Pour utiliser la **version mise à jour**, monter l'image à nouveau vers **Docker Hub** :

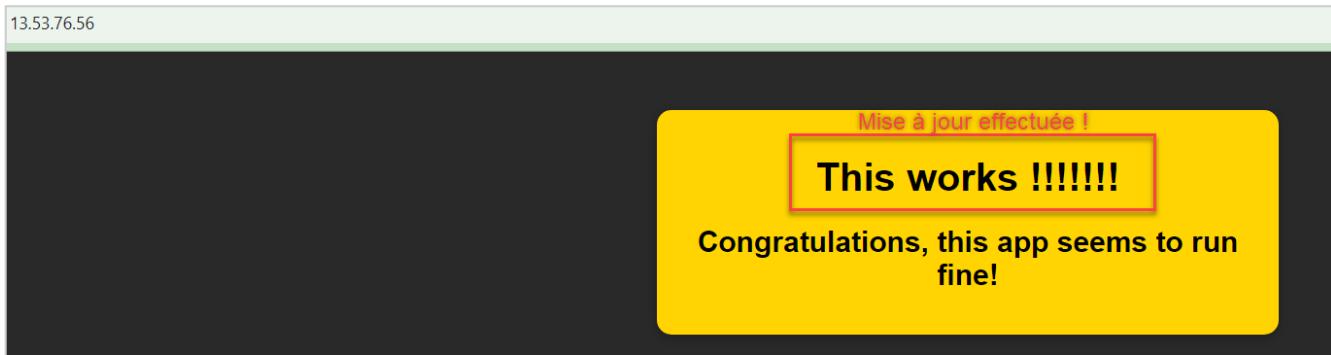
```
sudo docker pull bbauer02/node-example-1
```

```
[ec2-user@ip-172-31-46-118 ~]$ sudo docker pull bbauer02/node-example-1
Using default tag: latest
latest: Pulling from bbauer02/node-example-1
f56be85fc22e: Already exists
8f665685b215: Already exists
e5fcfa6c395a6: Already exists
56lcb69653d5: Already exists
f2b3cfab7b44f: Already exists
eb2c2a416ca5: Already exists
a2c78fa3560e: Already exists
9260194dea98: Pull complete
Digest: sha256:754abl0f2fe58d871eb2613efbc669fccd3d5e8ab2f6e9e49147158cf08eaa2e
Status: Downloaded newer image for bbauer02/node-example-1:latest
docker.io/bbauer02/node-example-1:latest
[ec2-user@ip-172-31-46-118 ~]$
```

- Ensuite, relancez le conteneur avec la commande :

```
sudo docker run -d --rm -p 80:80 bbauer02/node-example-1
```

Si vous rechargez la page dans votre navigateur, vous verrez les changements (par exemple, les points d'exclamation ajoutés).



11.8.2. Arrêter et Supprimer l'Instance EC2

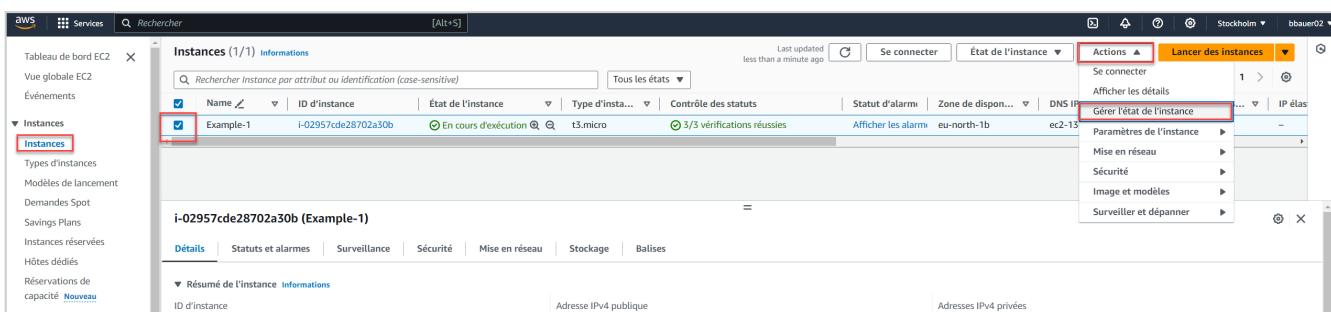
Si vous souhaitez arrêter temporairement l'application ou éteindre complètement l'instance EC2, voici comment procéder :

- Arrêter le Conteneur** : Utilisez `docker stop` pour arrêter le conteneur. Une fois le conteneur arrêté, **la page web ne sera plus accessible**, car il n'y aura plus de serveur web en cours d'exécution sur l'instance EC2.

```
sudo docker stop <nom_du_conteneur>
```

```
[ec2-user@ip-172-31-46-118:~] $ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
955ff97f385c5 bbauder02/node-example-1 "docker-entrypoint.s..." 3 minutes ago Up 3 minutes 0.0.0.0:80->80/tcp, :::80->80/tcp cranky_newton
[ec2-user@ip-172-31-46-118:~] $ docker stop cranky_newton
[ec2-user@ip-172-31-46-118:~] $
```

- Terminer l'Instance EC2** : Si vous souhaitez arrêter définitivement l'**instance EC2**, retournez dans le tableau de bord EC2, sélectionnez votre instance, allez dans le menu "Actions", choisissez "Gérer l'état de l'instance".



Sélectionner "Terminate(Résilier)" et cliquez sur "Modifier l'état".

Gérer l'état de l'instance

Instance details

i-02957cde28702a30b (Example-1) running

Paramètres d'état de l'instance

Début

Disponible lorsque l'instance est arrêtée

Stop (Arrêter)

Mettre en veille prolongée

Arrêt était absent pour cette instance – La mise en veille prolongée était activée lors du lancement.

Redémarrer

Terminate (Résilier)



Notez que lorsque vos instances sont résiliées :

Sur une instance basée sur EBS, l'action par défaut concerne la suppression du volume EBS racine lorsque l'instance est mise hors service. Le stockage sur les éventuels disques locaux sera perdu.

Annulez

Modifier l'état

Terminate (delete) instance ?

X

Sur une instance basée sur EBS, l'action par défaut concerne la suppression du volume EBS racine lorsque l'instance est mise hors service. Le stockage sur les éventuels disques locaux sera perdu.

Voulez-vous vraiment résilier ces instances ?

ID d'instance



i-02957cde28702a30b (Example-1)

Protection de la résiliation



Désactivé

To confirm that you want to delete the instances, choose the terminate button below. Instances with termination protection enabled will not be terminated. Terminating the instance cannot be undone.

Annulez

Terminate (delete)



Cette action est irréversible pour cette instance particulière.

11.8.3. Conclusion

Cet exemple de déploiement simple montre comment utiliser Docker pour exécuter une application sur un serveur distant, ainsi que la facilité avec laquelle vous pouvez mettre à jour l'application et gérer son cycle de vie. Bien que cette méthode soit efficace pour des applications simples, elle présente quelques **inconvénients**, surtout pour **les applications plus complexes ou lorsque des exigences spécifiques en matière de déploiement sont nécessaires**. C'est pourquoi nous examinerons d'autres méthodes de déploiement plus robustes et adaptées à des scénarios variés dans les prochaines sections de ce chapitre.

11.9. Inconvénients de l'approche actuelle

Au cours des dernières leçons, je vous ai montré comment déployer une application conteneurisée sur une instance **AWS EC2** et comment exécuter votre application dans un conteneur sur un hôte distant sans installer d'autres environnements ou logiciels supplémentaires sur cet hôte, à part Docker.

Docker est génial !

Il est nécessaire d'installer que Docker (et pas le runtime ou d'autres outils)

Uploader notre code est très simple

Nous avons exactement la même application et son environnement en production comme en développement

C'est l'une des raisons pour lesquelles Docker est si puissant : une fois Docker installé, il n'y a besoin daucun autre outil. Comme vous l'avez vu, déployer notre code était très simple, car nous avons simplement tiré la dernière image, garantissant que l'application et l'environnement sur la machine locale étaient identiques à ceux de la machine distante. Tout est encapsulé dans le conteneur.

Limites de l'Approche Manuelle

Bien que cette méthode de déploiement présente des avantages, elle comporte aussi des inconvénients. Je l'appelle l'approche, "faites-le vous-même", car nous avons dû créer l'**instance manuellement**, la **configurer manuellement**, nous y **connecter manuellement**, **installer Docker manuellement**, et effectuer de nombreuses étapes par **nous-mêmes**, même si l'exécution réelle de l'application restait relativement simple.

L'approche « faites-le vous-même » - inconvénients

Nous avons le contrôle total de la machine distante

=> **Nous sommes pleinement responsables d'elle et de sa sécurité**

Maintenir les applications critiques à jour

Gérer le réseau et les groupes de sécurité/firewall

Contrôle de la machine distante avec SSH en ligne de commande à saisir sois même => Fastidieux !

La chose la plus importante à retenir si vous choisissez cette approche (*c'est-à-dire utiliser votre propre serveur distant créé via un fournisseur d'hébergement cloud comme une instance EC2*) est que vous êtes entièrement responsable de cette machine distante.

Cela inclut non seulement sa configuration, mais aussi sa sécurité. Vous devez vous assurer que la machine est suffisamment puissante pour gérer le trafic, la remplacer par une machine plus performante si nécessaire, et maintenir tous les logiciels système à jour. Même si vous n'avez pas besoin d'installer des logiciels spécifiques à l'application, vous devez maintenir le système d'exploitation à jour.

Vous êtes également responsable de la gestion du réseau, des groupes de sécurité et du pare-feu. Si vous ne savez pas ce que vous faites, il est facile de créer une instance EC2 non sécurisée, ce qui la rend vulnérable aux piratages ou à d'autres abus.

Besoin d'une Approche Plus Gérée

En plus des responsabilités de sécurité, la connexion à la machine via SSH et l'installation manuelle de Docker, bien que fonctionnelles, peuvent être fastidieuses. Vous pourriez préférer un flux de travail de déploiement où vous n'avez pas besoin de vous connecter manuellement à la machine distante.

Idéalement, vous pourriez simplement exécuter une ou plusieurs commandes depuis votre machine locale, ce qui déplacerait automatiquement votre image Docker vers un hôte distant. Cet hôte distant pourrait également être géré pour s'assurer que vous n'avez pas à vous soucier de la mise à jour du système d'exploitation ou d'autres tâches de maintenance.

Je recommande d'opter pour une approche plus gérée, car elle vous permet de vous concentrer sur l'écriture de votre code source et la création de votre application conteneurisée, sans avoir à gérer les serveurs, les pare-feu et les réseaux, ce qui nécessite différentes compétences de celles d'un développeur web ou logiciel typique.

Conclusion et Prochaines Étapes

L'utilisation d'une instance EC2 a été un bon premier exemple pour démontrer comment vous pouvez installer Docker sur une machine distante et utiliser vos conteneurs comme vous le feriez localement. Cependant, pour des applications réelles, plus grandes, ou utilisant plusieurs conteneurs, cette approche peut ne pas être idéale, à moins que vous ne maîtrisiez parfaitement la gestion des serveurs. Dans certains cas, avoir un contrôle total peut être un avantage, mais pour de nombreux projets, ce ne sera pas le cas.

Pour cette raison, dans le reste de ce chapitre, nous n'utiliserons pas **EC2**. Je vous montrerai plutôt d'autres services gérés qui simplifient le déploiement en prenant en charge une grande partie de la gestion, vous permettant de vous concentrer davantage sur le développement de votre application.

11.10. Du déploiement manuel aux services managés

L'approche, "faites-le vous-même" ou "Do it Yourself en anglais" (DIY) présente certains inconvénients potentiels, comme mentionné plus tôt dans ce module lors de la discussion sur les compromis à faire. Parfois, vous devez choisir entre avoir un contrôle total, avec toutes les responsabilités qui en découlent, ou accepter de sacrifier un peu de contrôle pour bénéficier de moins de responsabilités.

Du développement à la mise en prod : les choses à garder en tête !

Ne pas utiliser de « Bind Mounts »

Certaines applications conteneurisées nécessitent une étape de compilation

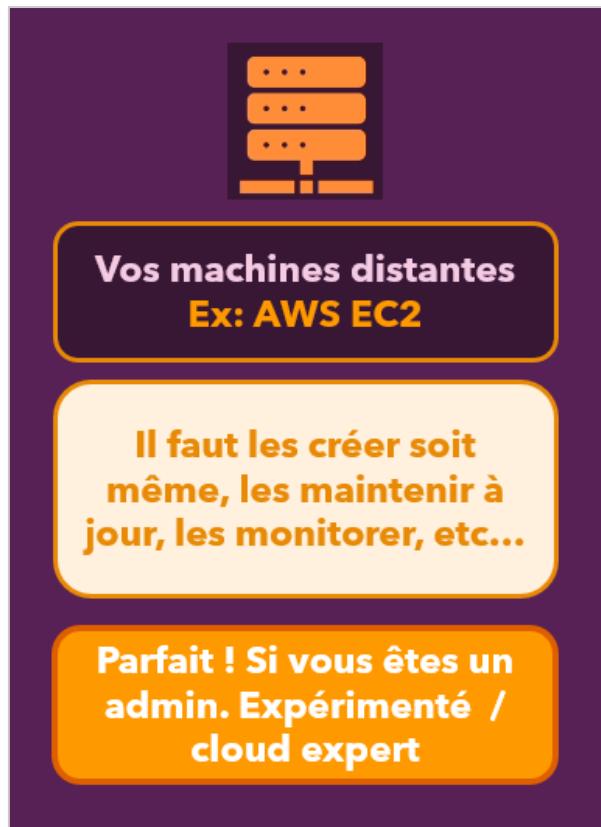
Les projets multi-conteneurs ont souvent besoin d'être séparé vers plusieurs machines (hôtes) distantes

Le compromis entre le contrôle et la responsabilité pourrait en valoir la peine.

Passer d'un Service Autogéré à un Service Géré

Dans ce contexte, cela peut signifier, passer d'un service entièrement autogéré, comme gérer soi-même des instances EC2, à une solution gérée par un tiers. En d'autres termes, au lieu d'exécuter nos propres machines et de les gérer de bout en bout, nous pourrions envisager d'utiliser un service géré.

Quand vous gérez vos propres machines ou hôtes distants, vous devez non seulement les créer et les maintenir vous-même, mais aussi vous assurer qu'ils sont à jour, surveillés, et capables de s'adapter à une augmentation du trafic. Si votre application croît et attire plus de trafic, **c'est à vous de vous assurer que le système peut supporter cette charge sans tomber en panne.**



Cela n'est pas forcément une mauvaise chose si vous avez les compétences nécessaires, surtout si vous êtes **un administrateur expérimenté ou un expert en cloud**. Cependant, sans cette expertise, il est facile de mettre en place **un système instable et potentiellement vulnérable**. C'est pourquoi utiliser un service géré pour les hôtes distants, comme **AWS ECS (Elastic Container Service)**, peut être avantageux.



ECS est un service d'Amazon qui simplifie la gestion des conteneurs, leur déploiement, leur exécution, leur surveillance, et bien plus encore.

Avantages des Services Gérés

L'avantage d'un service géré comme ECS est que la création, la gestion, la mise à jour, la surveillance et la mise à l'échelle sont simplifiées. C'est le fournisseur de services, **comme AWS**, qui se charge de ces aspects techniques, vous laissant simplement la tâche de spécifier comment vous voulez que ces tâches soient gérées.



Cela signifie moins de contrôle direct sur les détails, mais aussi **moins de responsabilité et moins de risques d'erreurs**.

Toutefois, il est important de noter qu'utiliser un service géré signifie que **vous ne vous contentez plus d'utiliser Docker**. Vous utilisez maintenant un service fourni par un fournisseur de cloud, et vous devez suivre les règles de ce service.



Cela signifie que le déploiement et l'exécution des conteneurs ne se feront plus via des commandes **Docker directes**, car vous ne gérez plus l'installation de Docker sur une machine.

Vous utiliserez plutôt les outils et interfaces fournis par **le service cloud spécifique**.

Exemple de l'Elastic Container Service (ECS) d'AWS

Figure 7. Documentation du service : <https://aws.amazon.com/fr/ecs/>

Dans les prochaines leçons, je vais vous montrer comment utiliser **AWS ECS** comme exemple de service géré pour le déploiement de conteneurs. ECS est un exemple parmi d'autres, et les étapes

que je vous montrerai fonctionneront **spécifiquement** pour ce service.

Bien sûr, il existe de nombreux autres services gérés par d'autres fournisseurs de cloud comme **Microsoft Azure**, qui propose également des services pour la gestion des conteneurs.



Ce que vous **apprendrez dans ces leçons vous aidera** non seulement à travailler avec ce service spécifique, mais aussi avec tout autre service de tout autre fournisseur de cloud, car chacun aura sa propre documentation et ses propres règles à suivre. Il est essentiel de rester **flexible** et d'être capable de s'**adapter** à ces **différentes règles**.

Conclusion

Le compromis consiste donc à décider si vous souhaitez tout gérer vous-même—*installer et configurer Docker sur une machine distante, exécuter toutes les commandes Docker que vous connaissez déjà*—ou si vous préférez utiliser un service géré, comme vous le verrez dans les prochaines leçons, où vous devrez suivre les directives et les outils fournis par le fournisseur.



Néanmoins, vous continuerez à travailler avec des conteneurs, et tout ce que vous avez appris jusqu'à présent s'appliquera encore, bien que certaines commandes soient différentes.

11.11. IMPORTANT : Avertissement concernant l'utilisation de AWS ECS (Elastic Container Service)





Dans les prochaines leçons, nous allons utiliser un service appelé **AWS ECS (Elastic Container Service)**. Contrairement à EC2, ECS n'est pas couvert par l'**offre gratuite d'AWS**.

Vous pouvez vérifier ce qui est inclus dans l'offre gratuite en consultant la page suivante : [AWS Free Tier](#).

Par conséquent, vous ne devriez suivre activement ces leçons que si vous êtes d'accord pour engager quelques coûts. En suivant les instructions données, vous ne serez facturé que quelques dollars, à condition de supprimer toutes les ressources une fois que vous avez terminé.

Pour en savoir plus sur les prix d'AWS, veuillez consulter leur page de tarification : [AWS Pricing](#).



Important : Il est essentiel de vérifier et de supprimer toutes les ressources créées (par exemple, les équilibreurs de charge, les passerelles NAT, etc.) une fois que vous avez terminé, sinon les coûts mensuels pourraient être beaucoup plus élevés !

11.12. Déploiement d'une Application avec AWS ECS

Dans cette section, nous allons répéter l'exemple de déploiement précédent, mais cette fois-ci en utilisant AWS ECS (**Elastic Container Service**) au lieu d'AWS EC2. L'approche avec ECS nous évite de gérer manuellement l'infrastructure, ce qui simplifie le déploiement de conteneurs.

11.12.1. Avertissement

Contrairement à EC2, les services ECS ne sont pas couverts par l'offre gratuite d'AWS.

Pour éviter des frais imprévus, il est recommandé de suivre passivement ces leçons si vous souhaitez éviter des coûts, ou de consulter les pages de tarification AWS pour connaître les coûts potentiels :



- [Page de l'offre gratuite d'AWS](#)
- [Page de tarification d'AWS](#)

Notez que même en utilisant un ensemble minimal de fonctionnalités, des frais peuvent s'appliquer.

Assurez-vous de supprimer toutes les ressources créées (comme les équilibreurs de charge et les passerelles NAT) pour éviter des coûts élevés.

11.12.2. Configuration de ECS

1. **Accès à ECS** : Accédez au service ECS en cliquant sur le menu des services d'AWS et en recherchant "ECS" (Elastic Container Service).
2. **Débuter avec ECS** : Vous verrez un écran de démarrage avec un bouton "Get Started". Cliquez dessus pour lancer l'assistant de création de conteneur avec ECS.

11.12.3. Déploiement de Conteneurs

11.12.3.1. Choix du Conteneur

- Le conteneur que nous allons déployer est celui nommé `node-example-1`, précédemment poussé sur Docker Hub. Si vous n'avez pas suivi les leçons précédentes, assurez-vous de construire une image à partir du projet d'exemple et de la pousser sur votre compte Docker Hub.

11.12.3.2. Configuration du Conteneur

- Nom du Conteneur** : Donnez un nom au conteneur, par exemple `node-demo`.
- Image du Conteneur** : Entrez le nom du référentiel Docker Hub, par exemple `votrenomutilisateur/node-example-1`. Si votre image est sur un autre registre de conteneurs, utilisez le domaine complet du registre suivi du nom du référentiel.
- Limites de Mémoire** : Laissez les paramètres par défaut.
- Mappage de Ports** : Exposez le port 80 pour que l'application soit accessible via HTTP.

11.12.3.3. Options Avancées

- Ignorer les paramètres de santé spécifiques pour l'instant. Ceux-ci permettent à AWS de vérifier l'état du conteneur.
- Les variables d'environnement et les points de montage peuvent être définis ici, de la même manière qu'avec la commande `docker run`.

11.12.3.4. Configuration des Tâches

Les tâches définissent le modèle pour votre application, spécifiant comment AWS doit exécuter vos conteneurs. Une tâche peut contenir un ou plusieurs conteneurs.

- Type de Lancement** : Utiliser FARGATE pour lancer le conteneur en mode "serverless". AWS gère l'infrastructure, démarrant le conteneur uniquement lorsque nécessaire.

11.12.4. Configuration du Service et du Cluster

- Service** : Définit comment la tâche doit être exécutée. Vous pouvez ajouter un équilibrEUR de charge, mais pour cet exemple, nous commencerons sans.
- Cluster** : Groupe de services où vos tâches s'exécutent. Un cluster peut regrouper plusieurs conteneurs logiquement.

11.12.5. Finaliser le Déploiement

- Création du Cluster** : Cliquez sur "Next" pour créer le cluster. AWS configurera toutes les ressources nécessaires pour exécuter le conteneur.
- Accès à l'Application** : Accédez à votre application en utilisant l'IP publique fournie par AWS sous la section "Tasks".

11.12.6. Mise à Jour des Conteneurs

Pour mettre à jour le conteneur ou l'image, reconstruisez l'image localement, poussez-la sur Docker Hub, et mettez à jour la configuration du conteneur dans ECS.

11.12.7. Conclusion

Avec AWS ECS, nous avons pu déployer une application sans gérer manuellement l'infrastructure sous-jacente. Ce processus simplifie le déploiement et la gestion des conteneurs en nous déchargeant des tâches de gestion des serveurs. Dans les prochaines leçons, nous explorerons comment gérer des applications multi-conteneurs avec ECS.