

Les images Docker

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

Chapitre

Sommaire

1. Introduction aux images Docker	1
2. Images et Conteneurs : Quelle différence ?	2
3. Les images Docker pré-construites	4
3.1. Préambule	4
3.2. Utiliser une image existante	5
4. Les images Docker personnalisée	9
4.1. Préambule	9
4.2. TD : Créer une image pour une application Node.js	9
4.2.1. Objectif	9
4.2.2. Préparation	9
4.2.3. Présentation des fichiers	9
4.2.4. Lancez l'application Node.js en Local	11
4.2.5. Dockerfile : Création de notre propre image Docker	13
4.2.5.1. Instruction : FROM	14
4.2.5.2. Instruction : COPY	14
4.2.5.3. Instruction : RUN	15
4.2.5.4. Instruction : WORKDIR	15
4.2.5.5. Instruction : CMD	16
4.2.5.6. Instruction : EXPOSE	17
4.2.5.7. Construction de l'image	17
4.2.5.8. Mapping de ports	20
4.3. Précisions sur les images Docker	23
4.3.1. Lecture seule	23
4.3.2. Comprendre les Couches d'images	24
5. Manager les images et les conteneurs	26
6. Stopper et redémarrer les conteneurs	27
7. Comprendre les modes attaché et détaché	28
8. Entrer dans un conteneur en marche	29
9. Le mode interactif	30
10. Suppression des images et des conteneurs	31
11. Supprimer un conteneur arrêté automatiquement	32
12. Inspecter un conteneur	33
13. Copier des fichiers dans et depuis un conteneur	34
14. Nommer et tagger des conteneurs et images	35
15. Mise en pratique	36
16. Partager des images	37
17. Push d'images sur Dockerhub	38
18. Pull et utilisation d'images partagées	39

19. Résumé	40
------------------	----

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.

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.

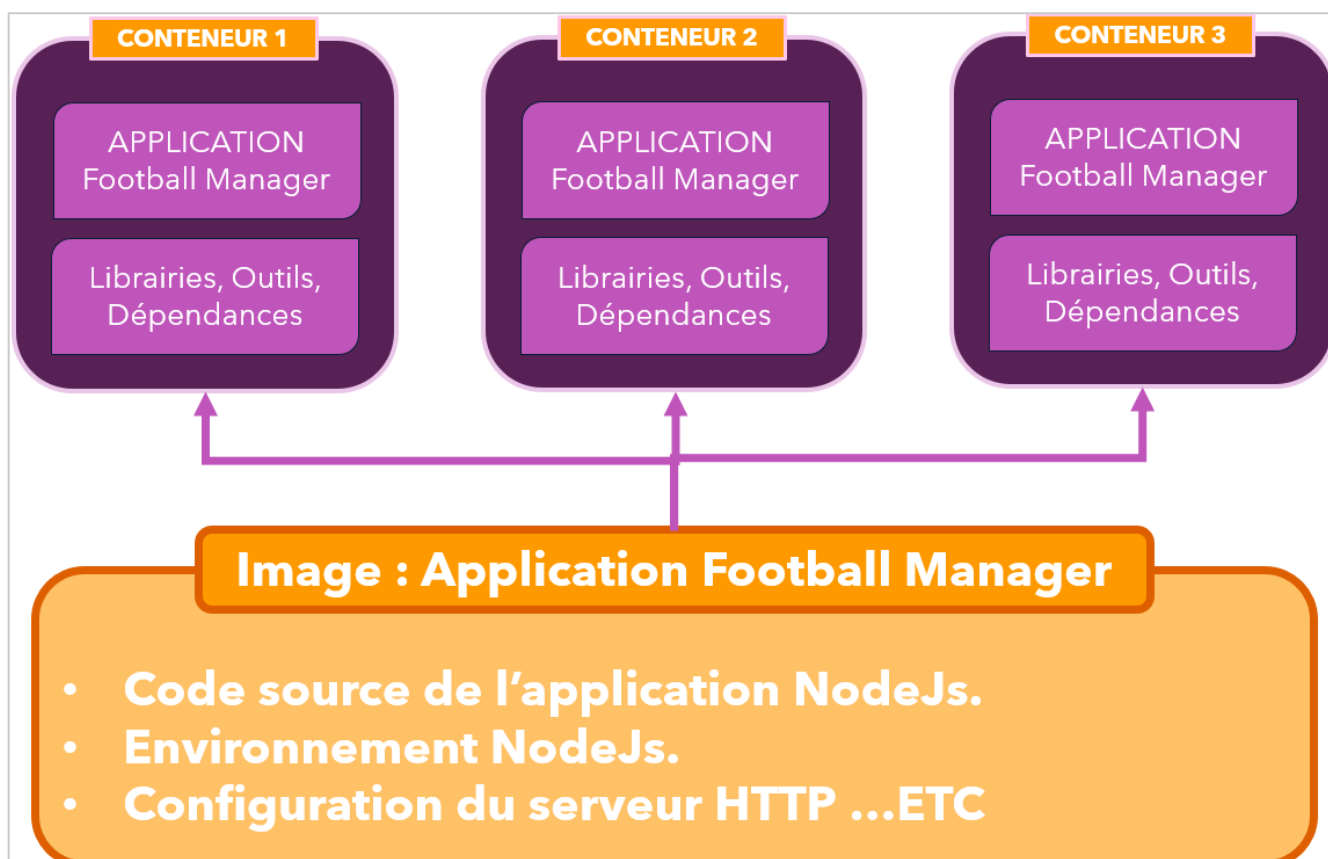
D'autre part, une image est un fichier contenant tout ce dont nous avons besoin pour créer un conteneur. Une image est un modèle, une sorte de gabarit qui nous permet de créer un conteneur. Elle contient le code source et les outils nécessaires pour exécuter une application.

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.

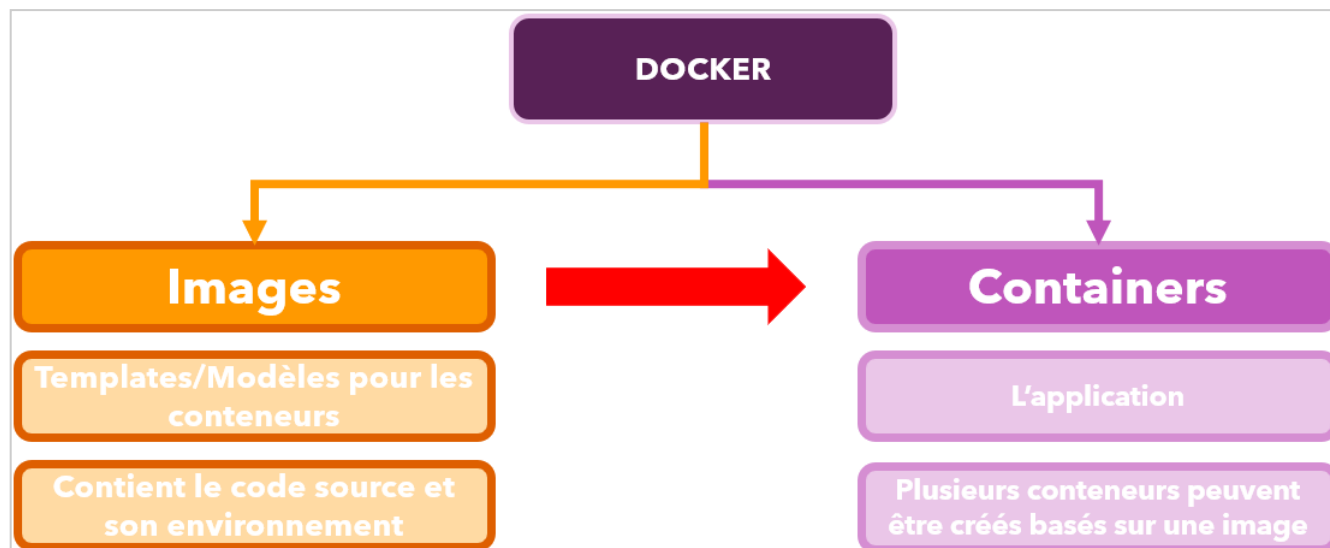


Cette image est un package partageable contenant toutes les instructions d'installation et de configuration de l'application. Le conteneur est une instance de cette image qui exécute l'ensemble des instructions.



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

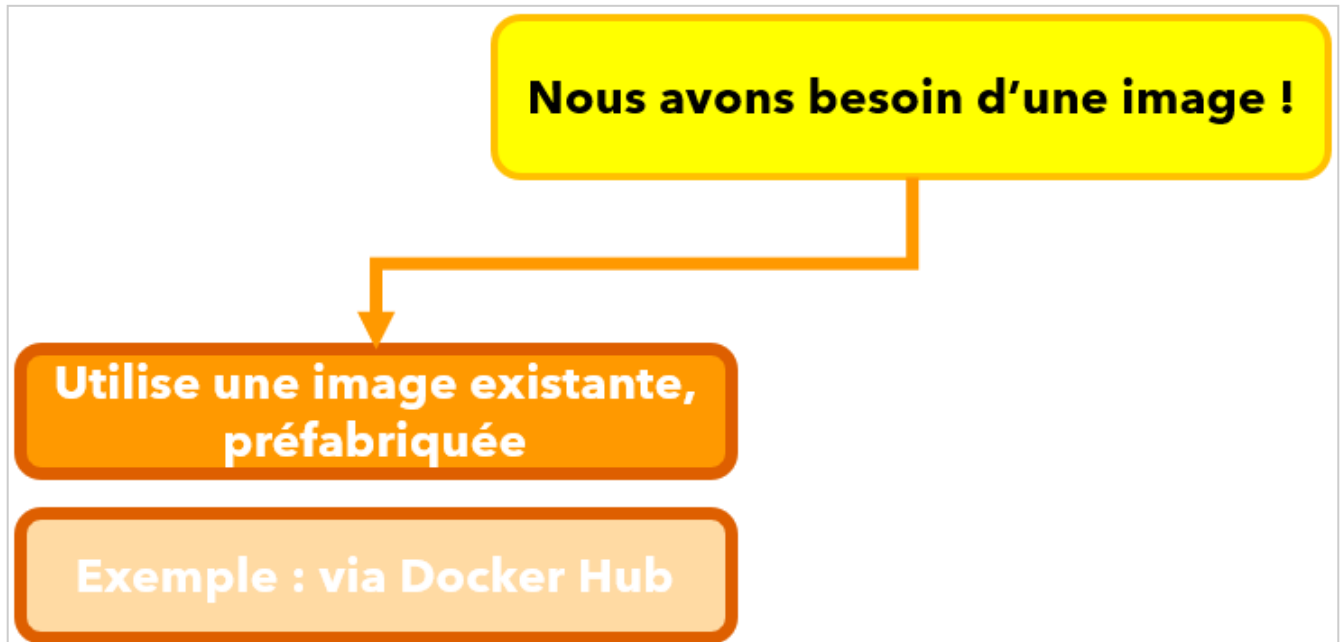
Cela deviendra encore plus clair lorsque nous commencerons à manipuler les images et les conteneurs.



3. Les images Docker pré-construites

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.



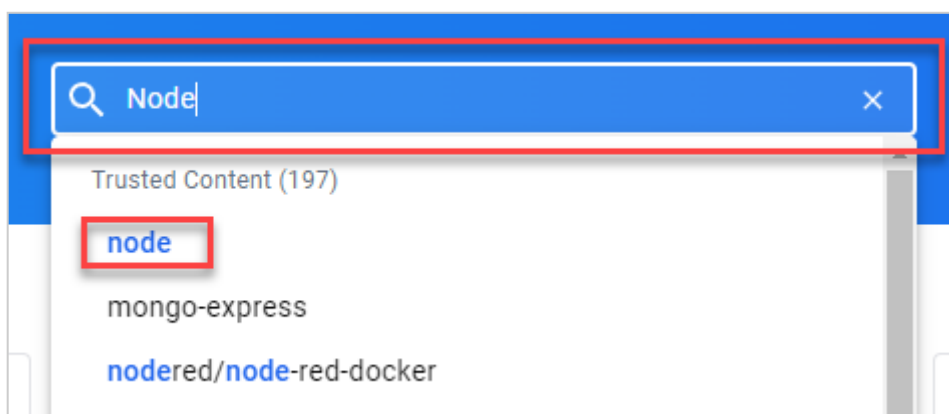
Utiliser des conteneurs existants : Ceux-ci sont créés par la communauté, nos collègues ou officiellement par les éditeurs de logiciels.

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 'node' entered and a 'docker pull node' button. Below this, the 'Tags' tab is selected, showing a list of tags. The first tag is 'lts-alpine3.19', which is highlighted. A red arrow points from the text 'Nom de l'image' to the 'node' part of the search bar. Another red arrow points from the same text to the 'lts-alpine3.19' tag. A green arrow points from the text 'Tag de l'image' to the 'lts-alpine3.19' tag. The table below shows the details for the 'lts-alpine3.19' tag, including its digest, OS/ARCH, vulnerabilities, and compressed size.

TAG	Digest	OS/ARCH	Vulnerabilities	Compressed Size
lts-alpine3.19				
Last pushed 5 days ago by dojanky				
	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.

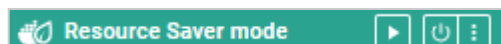
Vous pouvez sélectionner une image particulière contenant une version spécifique du service ou du système Linux de base. Pour ce faire, consultez l'onglet **Tag** et récupérez le nom de tag correspondant à la version souhaitée.

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

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

3.2. Utiliser une image existante

Assurez-vous d'avoir Docker Engine 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>` concernera 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 trouvée localement. Ainsi, elle sera automatiquement téléchargée 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
375735fcaa7a: Waiting
c12db77023cd: Waiting
ac50344c1606: Waiting
```

Message d'erreur indiquant que l'image "node" n'a pas été trouvée 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
375735fcaa7a: 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 ?



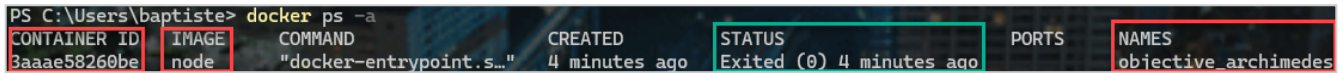
Il est essentiel de noter que par **défaut**, un conteneur est isolé de son environnement immédiat. Cependant, il est possible d'interagir avec lui via un

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 ne 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.

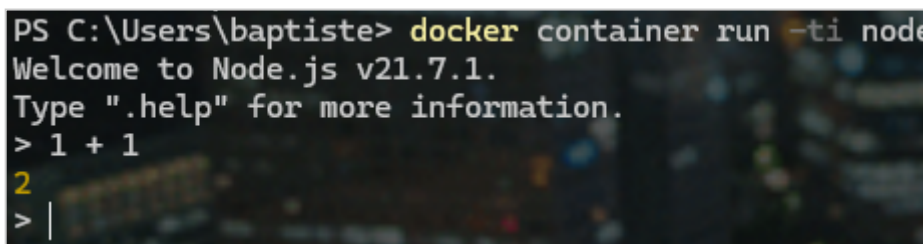
Attardons-nous sur le **STATUS** qui est **Exited**. Qui signifie que le conteneur a bien démarré une fois, puis s'est éteint.

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 : **-it**. Ce paramètre indique que nous souhaitons interagir avec le conteneur.

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



```
PS C:\Users\baptiste> docker container run -ti node
Welcome to Node.js v21.7.1.
Type ".help" for more information.
> 1 + 1
2
> |
```

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**, indépendamment 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.

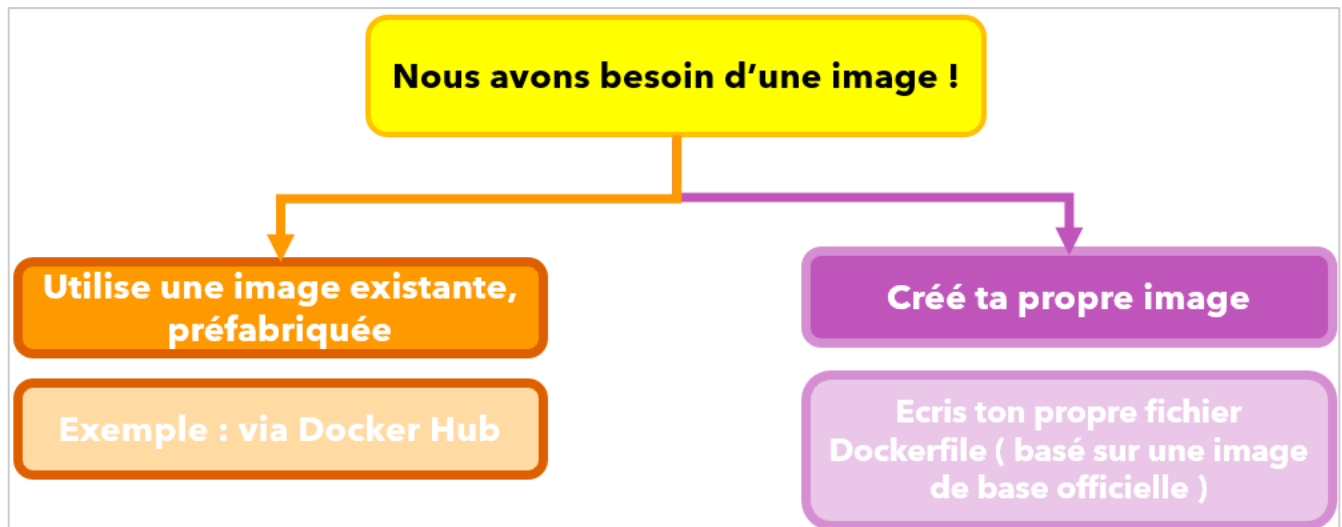
Nous verrons comment créer des images personnalisées dans le chapitre suivant.

4. Les images Docker personnalisée

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.



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

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.

4.2.2. Préparation



Pour réaliser ce TD, il n'est pas nécessaire de connaître Node.JS, 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.

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.

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**.

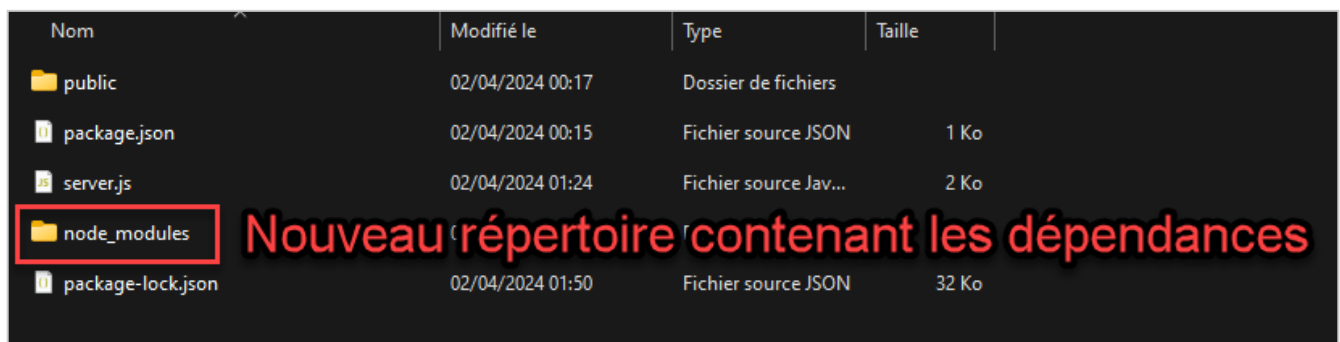
Installez les dépendances de l'application en exécutant la commande suivante :

```
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.
```

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			
package-lock.json	02/04/2024 01:50	Fichier source JSON	32 Ko

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 !

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**.

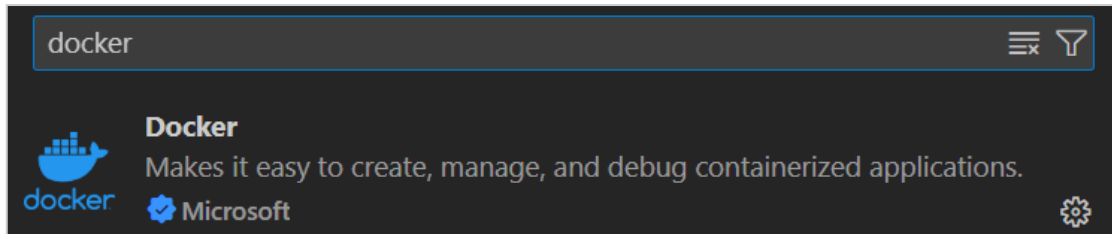


Figure 1. L'extension Docker pour Visual Studio Code

Ou

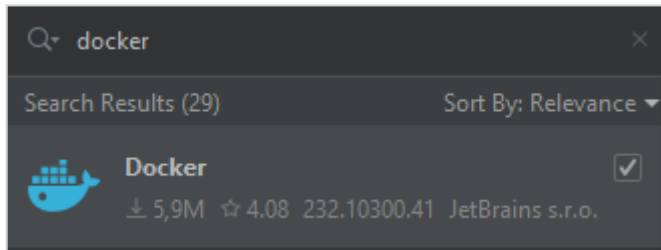


Figure 2. L'extension Docker pour PhpStorm

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**.

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.

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.

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.

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
```

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.

4.2.5.6. Instruction : EXPOSE

Comme nous l'avons mainte 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.

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 .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
```

La commande à saisir dans le répertoire du Dockerfile

Nous retrouvons les instructions du Dockerfile

Lorsque l'image est créée, nous recevons son identifiant unique

What's Next?
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.

```
PS C:\Users\baptiste> docker images
```

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.

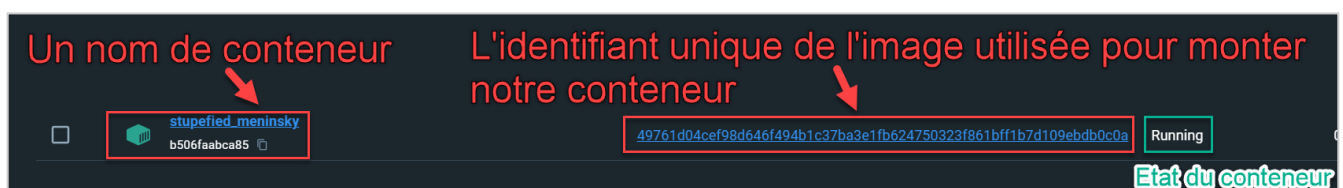
Grâce à son identifiant unique, nous pouvons maintenant lancer un conteneur basé sur cette image.

```
docker container run 49761d04cef98d646f494b1c37ba3e1fb624750323f861bfff1b7d109ebdb0c0a
```

En exécutant cette commande, nous constatons que nous n'avons plus la main sur la console.

Cela est normal, car l'instruction `CMD` de notre `Dockerfile` exécute le serveur Node.js en arrière-plan. Notre conteneur est donc en cours d'exécution avec le script `server.js` qui écoute sur le port 80.

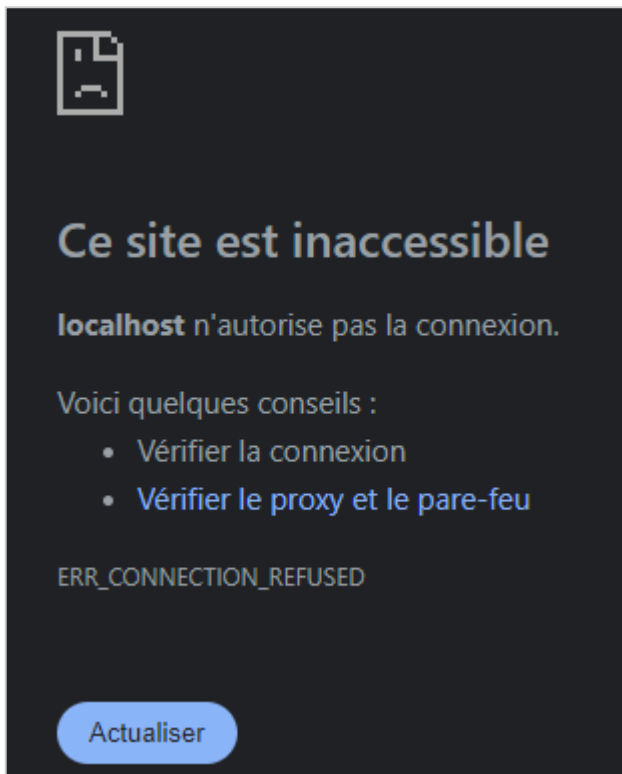
Pour vérifier, ouvrez "Docker Desktop", cliquez sur l'onglet "Containers" et vous verrez le conteneur en cours d'exécution.



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**.

```
PS C:\Users\baptiste> docker container ps
```

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.

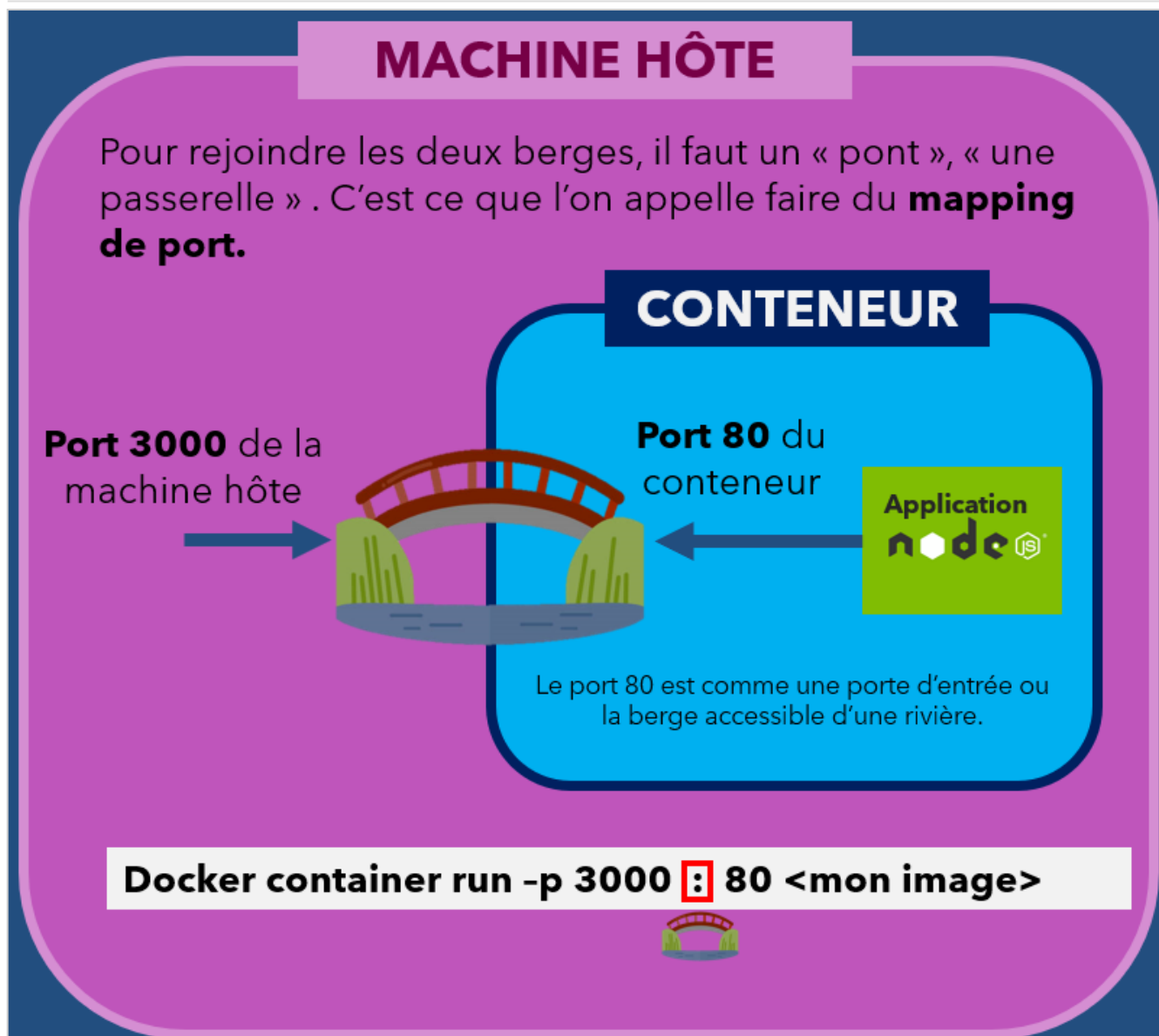
Mais concrètement, cela ne suffit pas ! Car l'instruction `EXPOSE` n'ouvre pas le port 80 de notre conteneur vers la machine hôte, elle est une facultative. Elle ne sert à rien d'autre **qu'à documenter le port** sur lequel notre application écoute. C'est une bonne pratique pour informer les autres utilisateurs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

4.2.5.8. Mapping de ports



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 c10bc7ec8382	49761d04cef98d646f494b1c37ba3e1fb624750323f861bf1b7d109ebdb0c0a	Running	0%	3000:80	1 minute ago	

Le voyant est au vert !

Le Status est "Running"

Mapping de ports OK.
Le lien est cliquable !

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.

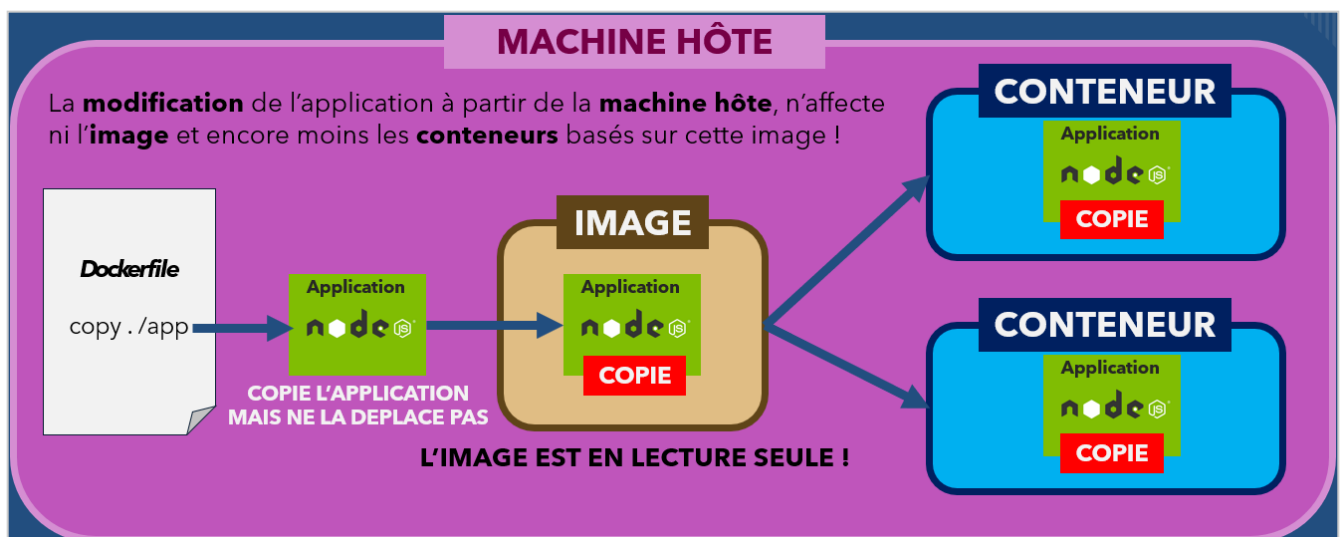
Cela s'applique à TOUTES LES COMMANDES DOCKER.

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.

4.3. Précisions sur les images Docker

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.

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                                docker:default
=> [internal] load build definition from Dockerfile             0.1s
=> => transferring dockerfile: 135B                             0.0s
=> [internal] load .dockerignore                               0.1s
=> => transferring context: 2B                                    0.0s
=> [internal] load metadata for docker.io/library/node:latest  0.0s
=> [1/4] FROM docker.io/library/node:latest                   0.3s
=> [internal] load build context                               0.1s
=> => transferring context: 2.16kB                               0.0s
=> [2/4] WORKDIR /app                                          0.0s
=> [3/4] COPY . /app                                           0.1s
=> [4/4] RUN npm install                                       2.7s
=> exporting to image                                         0.1s
=> => exporting layers                                         0.1s
=> => writing image sha256:49761d04cef98d646f494b1c37ba3e1fb624750323f861bffa1b7d109ebdb0c0a 0.0s
```

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
=> [3/4] COPY . /app
=> [4/4] RUN npm install
=> exporting to image
=> => exporting layers
=> => writing image sha256:71c81241a13ce95decdfb65486eac94f70fd2499abc1e33f3693752e7b947bf3
```

Si le code source de l'application est modifié, l'instruction `COPY` est rejouée, ainsi que toutes les instructions suivantes.

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.

5. Manager les images et les conteneurs

To do ...

6. Stopper et redémarrer les conteneurs

To do ...

7. Comprendre les modes attaché et détaché

To do ...

8. Entrer dans un conteneur en marche

To do ...

9. Le mode interactif

To do ...

10. Suppression des images et des conteneurs

To do ...

11. Supprimer un conteneur arrêté automatiquement

To do ...

12. Inspecter un conteneur

To do ...

13. Copier des fichiers dans et depuis un conteneur

To do ...

14. Nommer et tagger des conteneurs et images

To do ...

15. Mise en pratique

To do ...

16. Partager des images

To do ...

17. Push d'images sur Dockerhub

To do ...

18. Pull et utilisation d'images partagées

To do ...

19. Résumé

To do ...