COMMENT OPTIMISER VOS IMAGES DOCKER ?

Introduction

Docker (d'ailleurs voici mon cours complet sur docker) est devenu un outil incontournable pour les développeurs et les ingénieurs DevOps. Il offre une flexibilité et une facilité d'utilisation qui simplifient la création, le déploiement et l'exécution d'applications dans des conteneurs.

Cependant, la taille et le temps de construction des images Docker peuvent rapidement devenir des problèmes. Dans cet article, nous explorerons **différentes méthodes et stratégies pour optimiser vos images Docker**.

Contexte

Les images Docker sont construites à partir de fichiers Dockerfile. Ces fichiers définissent à quoi devrait ressembler l'image, ainsi que le système d'exploitation et les commandes à exécuter.

De grandes images Docker peuvent allonger le temps nécessaire pour construire et partager des images entre des clusters et des fournisseurs de cloud. Il est donc utile d'optimiser les images Docker et les Dockerfiles pour aider les équipes à partager des **images Docker plus petites et plus performances** et à résoudre les problèmes.

Inspection des Images Docker

Avant de plonger dans les techniques d'optimisation, il est crucial de comprendre comment **inspecter la taille et le temps de construction de vos images Docker**. L'inspection vous donne une idée claire des étapes qui nécessitent une optimisation.

Historique de votre image Docker

La commande `docker image history img_name` vous permet d'examiner la création de votre image Docker couche par couche. Chaque couche représente une instruction dans votre Dockerfile et cette commande vous montre la taille de chaque couche.

Cette information est cruciale car elle vous permet de vous concentrer sur les couches qui ajoutent le plus de poids à votre image. En optimisant ces couches, vous pouvez obtenir la plus grande réduction de taille.

Supposons que vous ayez exécuté la commande `docker image history img_name` et que la sortie ressemble à ceci :

```
IMAGE       CREATED        CREATED BY                                      SIZE
b91d4548528d 34 seconds ago /bin/sh -c apt-get install -y python3 python…  140MB
f5b439869a1b 2 minutes ago  /bin/sh -c apt-get install -y wget               7.42MB
9667e45447f6 About an hour ago /bin/sh -c apt-get update                    27.1MB
a2a15febcdf3 3 weeks ago    /bin/sh -c #(nop) CMD [“/bin/bash”]             0B
```

Dans cet exemple, nous pouvons voir que l'installation de Python 3 a ajouté 140 Mo à l'image, ce qui est assez important. L'installation de `wget` a ajouté 7,42 Mo, et la mise à jour de apt a ajouté 27,1 Mo.

Si votre application n'a pas besoin de toutes les fonctionnalités de Python 3, vous pourriez envisager d'utiliser une version plus légère ou même un autre langage qui nécessite moins d'espace. De même, si `wget` n'est utilisé que pour une opération

spécifique qui pourrait être réalisée autrement, vous pourriez envisager de le supprimer pour réduire la taille de l'image.

En résumé, cette commande vous aide à identifier les couches qui ajoutent le plus de poids à votre image. Vous pouvez alors vous concentrer sur l'**optimisation de ces couches spécifiques Docker** pour réduire la taille globale de l'image.

Suivi du temps

Si vous souhaitez mesurer le temps que prend chaque étape de votre Dockerfile, vous pouvez simplement utiliser la sortie standard de la commande `docker build`. Cette méthode est intégrée et ne nécessite pas d'outils externes.

Exécutez la commande suivante pour construire votre image Docker et observer le temps pris par chaque étape :

```
docker build -t my-image .
```

Supposons que la sortie ressemble à ceci :

```
[+] Building 64.5s (6/6) FINISHED
 => [internal] load metadata for docker.io/library/ubuntu:latest 3.0s
 => [1/2] FROM docker.io/library/ubuntu@sha256:xxx 27.0s
 => [2/2] RUN apt-get update && apt-get install -y curl && rm -rf /var/lib/apt/lists/
 => exporting to image 0.2s
```

Dans cet exemple, le chargement des métadonnées pour l'image de base Ubuntu a pris 3,0 secondes, l'étape de construction à partir de cette image a pris 27,0 secondes, et l'installation de curl a pris 34,1 secondes.

Si certaines étapes prennent un temps considérable, vous pouvez vous concentrer sur ces étapes pour les optimiser. Par exemple, si l'installation de certains packages prend 34,1 secondes, vous pourriez envisager de créer une image de base qui inclut

ces packages, ce qui réduirait ce temps à presque zéro dans les constructions futures.

Réduire la taille de l'image

Choisir une image de base légère

L'image de base que vous choisissez pour votre Dockerfile a un impact significatif sur la taille finale de l'image Docker. Les images de base contiennent le système d'exploitation et les bibliothèques essentielles sur lesquelles votre application sera construite. Par conséquent, une image de base plus légère peut souvent **réduire considérablement la taille globale de l'image**.

Par exemple, considérez que vous utilisez l'image Ubuntu 18.04 comme base, qui a une taille d'environ 64,2 Mo.

```
FROM ubuntu:18.04 # 64.2MB
```

Cette image inclut de nombreux outils et bibliothèques qui peuvent ne pas être nécessaires pour votre application. Si votre application n'a pas besoin de ces fonctionnalités supplémentaires, cette taille est inutilement grande.

Une alternative serait d'utiliser une image plus légère comme Alpine Linux, qui a une taille d'environ 5,58 Mo.

```
FROM alpine:3 # 5.58MB
```

Information

Alpine Linux est conçu pour être petit, simple et sécurisé. Il est utilisé pour fournir les fonctionnalités essentielles et laisse de côté les fonctionnalités qui sont généralement inutiles pour les conteneurs Docker. Cela le rend idéal pour les applications qui n'ont pas besoin d'un système d'exploitation complet ou qui nécessitent une image aussi petite que possible.

Attention

Il est important de noter que toutes les applications ne fonctionneront pas sur Alpine. Par conséquent, il est crucial de tester votre application pour vous assurer qu'elle fonctionne comme prévu sur une image plus légère.

Nettoyage après installation

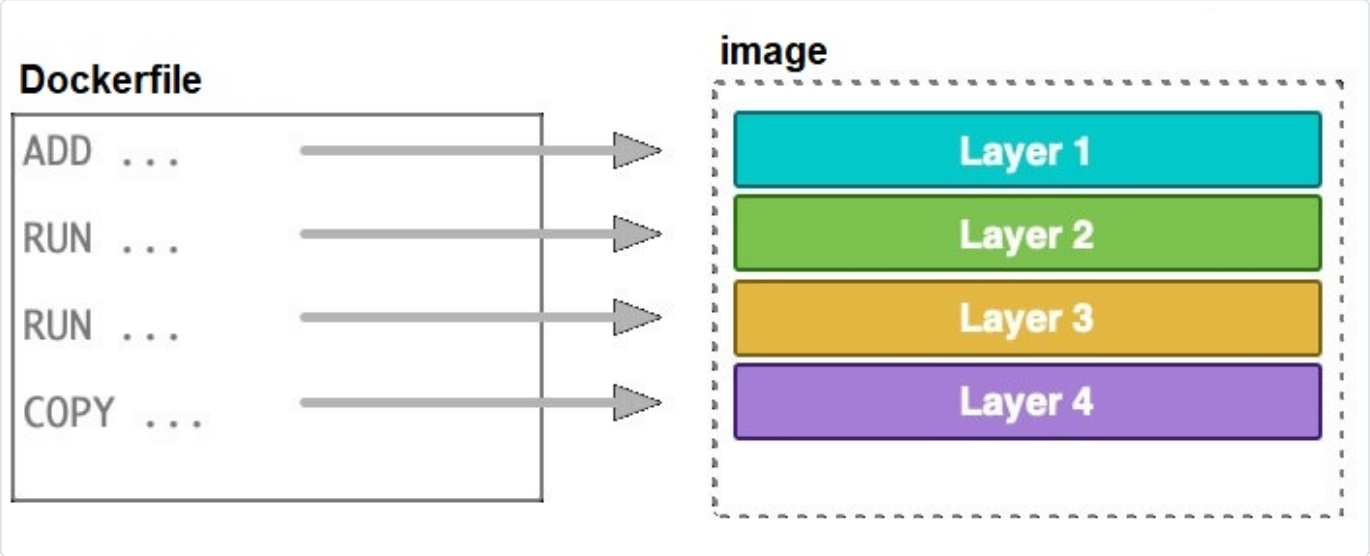
Lors de l'installation de packages avec des gestionnaires de paquets comme `apt` sous Ubuntu, des fichiers temporaires et des caches sont souvent créés. Ces fichiers sont utiles pour accélérer les installations futures, mais dans le contexte d'une image Docker, ils ne font qu'ajouter un poids inutile.

Par conséquent, il est recommandé de **nettoyer ces fichiers temporaires immédiatement** après l'installation des packages nécessaires. Voici comment vous pouvez le faire :

```
RUN apt-get update && \
apt-get install -y some-package && \
apt-get clean && \
rm -rf /var/lib/apt/lists/*
```

Dans cet exemple, la commande `apt-get clean` supprime les fichiers de package qui ont été téléchargés pendant l'installation. La commande `rm -rf /var/lib/apt/lists/*` supprime les métadonnées du package qui ne sont plus nécessaires une fois que les packages ont été installés.

Il est crucial que ces commandes de nettoyage soient dans la même instruction `RUN` que l'installation du package. En Docker, chaque instruction `RUN` crée une nouvelle couche, et si le nettoyage est effectué dans une couche différente, les fichiers temporaires resteront dans la couche précédente, annulant ainsi tout bénéfice en termes de réduction de la taille de l'image.



En suivant cette approche, vous pouvez réduire significativement la taille de votre image Docker sans compromettre les fonctionnalités de votre application.

Utilisation de builds statiques

Les builds statiques sont des compilations de code qui incluent toutes les dépendances nécessaires au sein du fichier exécutable lui-même. Cela contraste avec les builds dynamiques, qui nécessitent des bibliothèques partagées séparées.

L'avantage des builds statiques est qu'ils peuvent réduire considérablement la taille de votre image Docker, car vous n'avez pas besoin d'inclure des bibliothèques partagées supplémentaires. De plus, cela peut également **améliorer la sécurité de votre image Docker** en réduisant le nombre de composants dépendants.

```
# Exemple avec un build statique
RUN wget -q https://johnvansickle.com/ffmpeg/builds/ffmpeg-git-amd64-static.tar.xz &&
tar xf ffmpeg-git-amd64-static.tar.xz && \
mv ./ffmpeg-git-20190902-amd64-static/ffmpeg /usr/bin/ffmpeg && \
rm -rfd ./ffmpeg-git-20190902-amd64-static && \
rm -f ./ffmpeg-git-amd64-static.tar.xz  # Taille finale : 74.9MB
```

Dans cet exemple, nous téléchargeons une version statique de FFmpeg, ce qui nous permet d'économiser de l'espace par rapport à l'installation de FFmpeg via un gestionnaire de paquets qui utiliserait une version dynamique.

```
# Exemple avec un build dynamique
RUN apt-get install -y ffmpeg  # Taille finale : 270MB
```

Comme vous pouvez le voir, l'utilisation d'un build statique dans cet exemple a permis de réduire la taille de l'image de près de 200 Mo. C'est une économie significative, surtout si vous prévoyez de déployer cette image sur plusieurs machines ou dans un environnement de cloud.

Attention

Il est important de noter que tous les projets ne bénéficieront pas de builds statiques et que certains peuvent nécessiter des bibliothèques dynamiques spécifiques. Assurez-vous donc de tester soigneusement votre application dans l'environnement cible.

Installation des dépendances nécessaires

Lors de l'installation de paquets ou de bibliothèques, il est courant que le gestionnaire de paquets suggère également d'installer des dépendances "recommandées". Bien que ces dépendances puissent être utiles dans certains cas, elles peuvent également augmenter inutilement la taille de votre image Docker.

Pour éviter cela, vous pouvez utiliser des drapeaux spécifiques lors de l'installation de paquets pour ignorer ces dépendances recommandées. Par exemple, avec `apt-get`, vous pouvez utiliser le drapeau `--no-install-recommends`.

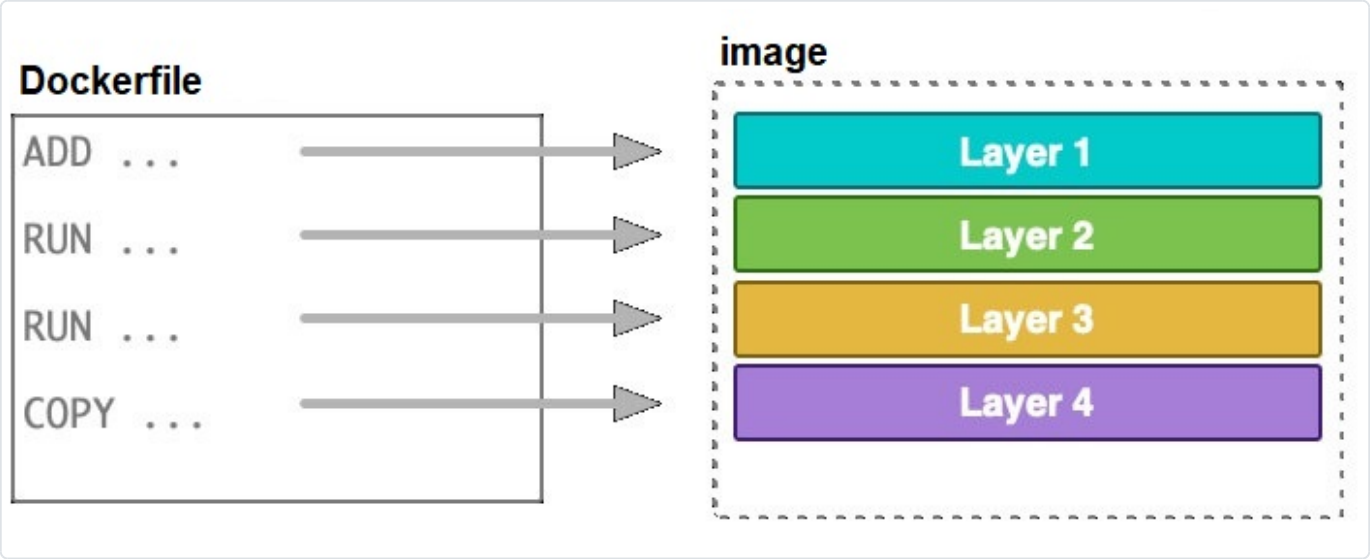
```
# Sans --no-install-recommends
RUN apt-get install -y python3-dev  # Taille finale : 144MB

# Avec --no-install-recommends
RUN apt-get install --no-install-recommends -y python3-dev  # Taille finale : 138MB
```

Dans cet exemple, l'utilisation du drapeau `--no-install-recommends` a permis de réduire la taille de l'image de 6 Mo. Cela peut sembler minime, mais ces petites économies peuvent s'accumuler, surtout si vous installez de nombreux paquets.

Il est important de noter que cette approche nécessite une certaine prudence. Assurez-vous de bien comprendre les dépendances dont votre application a réellement besoin pour fonctionner correctement, car l'omission de paquets importants peut entraîner des problèmes.

Optimisation de la mise en cache des dépendances

Lorsque vous construisez une image Docker, chaque instruction dans votre Dockerfile crée une "couche" qui est **mise en cache par Docker**. Cette mise en cache est utile car si rien n'a changé dans une couche particulière, Docker la réutilisera plutôt que de la reconstruire, ce qui accélère le processus de construction.



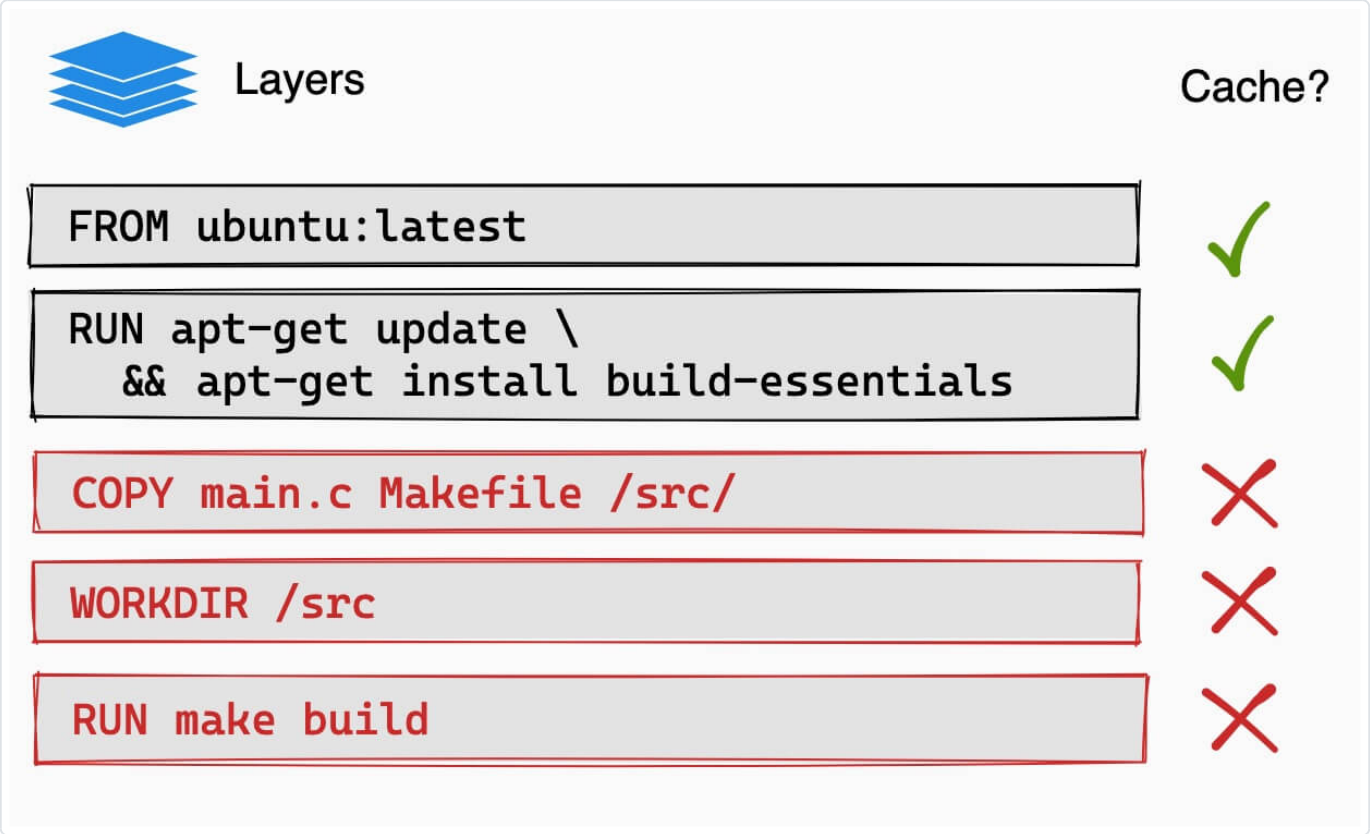
Imaginons que vous ayez une application qui dépend de plusieurs bibliothèques ou "dépendances". Chaque fois que vous modifiez une ligne de code dans votre application, vous ne voulez pas attendre que toutes ces dépendances soient téléchargées et installées à nouveau, n'est-ce pas ? C'est là que la mise en cache des dépendances entre en jeu.

L'idée est de copier et d'installer vos dépendances en premier, avant de copier le reste de votre code source. Ainsi, à moins que vos dépendances ne changent (ce qui est rare par rapport aux changements de code), cette étape sera mise en cache et ne sera pas exécutée à chaque construction.

```
# Exemple avec un projet Node.js
# Copier seulement le fichier package.json pour installer les dépendances
COPY package.json ./
RUN npm install

# Copier le reste du code source
COPY . .
```

Dans cet exemple, si vous modifiez une ligne de code dans votre application mais que vous ne modifiez pas le fichier `package.json`, Docker sautera l'étape `RUN npm install` et utilisera la version mise en cache. Cela peut réduire le temps de construction de plusieurs minutes à quelques secondes.

Pour illustrer, disons que l'installation des dépendances prend 5 minutes et que la copie de votre code prend 10 secondes. Sans mise en cache, chaque construction prendrait environ 5 minutes et 10 secondes. Avec la mise en cache, la première construction prendrait 5 minutes et 10 secondes, mais les constructions suivantes ne prendraient que 10 secondes si les dépendances ne changent pas.



C'est une énorme économie de temps, surtout si vous construisez souvent votre image, comme dans un environnement de développement continu.

Constructions Multi-stages

Prenons un exemple concret pour mieux comprendre. Imaginons que vous ayez une application web qui utilise Node.js pour le back-end et Nginx pour servir les fichiers statiques. Dans un scénario typique, vous auriez besoin de Node.js pour construire votre application, mais une fois que l'application est construite, vous

n'avez plus besoin de Node.js pour la faire fonctionner, vous avez seulement besoin des fichiers statiques générés et de Nginx.

Pour mieux comprendre l'efficacité des constructions multi-étapes avec un exemple concret, imaginons que vous construisiez une application web simple avec Node.js et que vous souhaitiez la déployer avec Nginx.

Dans une approche traditionnelle, votre Dockerfile pourrait ressembler à ceci :

```
FROM nginx:alpine
WORKDIR /app
COPY package*.json ./
RUN apk add --no-cache nodejs npm
RUN npm install
COPY . .
RUN npm run build
COPY ./dist /usr/share/nginx/html
```

Avec cette approche, votre image finale pourrait avoir une taille d'environ 200MB (c'est une estimation pour l'exemple).

Maintenant, utilisons une construction multi-étapes :

```
# Étape de construction
FROM node:14 AS build-stage
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

# Étape de production
FROM nginx:alpine AS production-stage
COPY --from=build-stage /app/dist /usr/share/nginx/html
```

Dans cette première étape, nous utilisons une image Node.js pour construire notre application. Le `RUN npm run build` génère des fichiers statiques à partir de votre code source. Ces fichiers sont généralement placés dans un répertoire comme `/app/dist` après la construction.

```
# Étape de production
FROM nginx:alpine AS production-stage
COPY --from=build-stage /app/dist /usr/share/nginx/html
```

Dans la deuxième étape, nous utilisons une image Nginx pour servir ces fichiers statiques. Le `COPY --from=build-stage /app/dist /usr/share/nginx/html` copie uniquement les fichiers statiques nécessaires pour exécuter l'application depuis l'étape de construction vers l'étape de production.

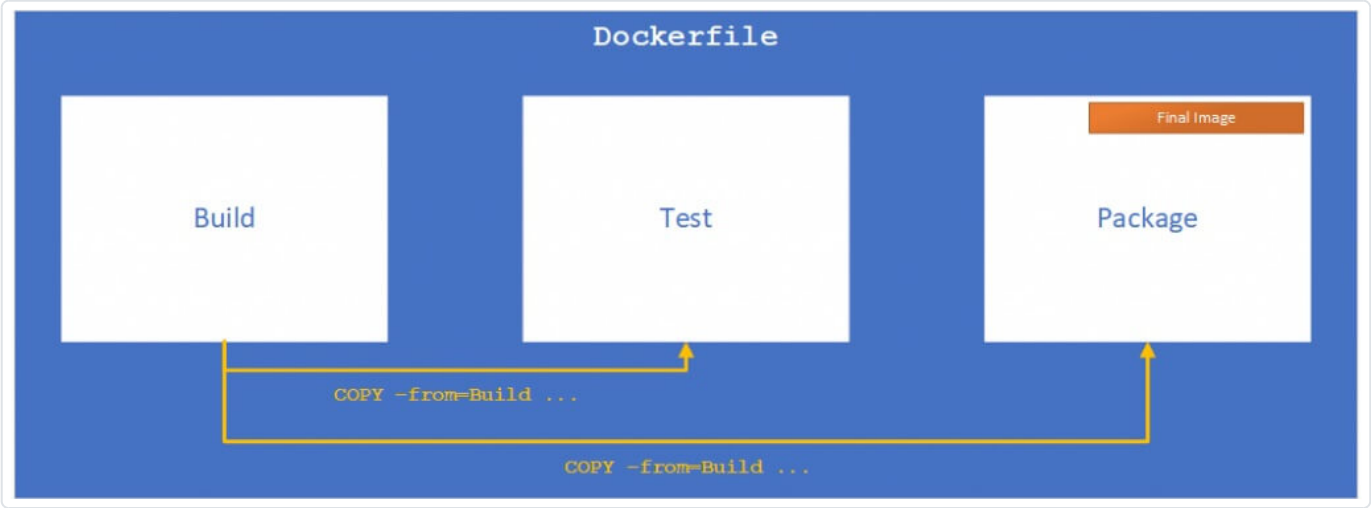
Pour les novices de NodeJS, Le répertoire **/app/dist** contient ici les fichiers statiques générés par la commande `npm run build`. Ce sont les fichiers qui sont effectivement nécessaires pour exécuter votre application. Tout le reste, comme les fichiers source, les dépendances de développement et les outils de construction, n'est pas nécessaire pour exécuter l'application et est donc omis, ce qui réduit la taille de l'image finale.

Avec cette approche, votre image finale pourrait avoir une taille d'environ 30MB. Voici une mini comparaison des deux approches:

Approche	Taille de l'image finale	Temps de construction
Traditionnelle	200MB	5 minutes
Multi-étapes	30MB	3 minutes

Pour arriver à ce résultat incroyable, dans l'approche multi-étapes, nous avons séparé le processus de construction de l'application du processus de création de l'image de production. Cela nous permet de n'inclure que les fichiers strictement nécessaires pour exécuter l'application dans l'image finale, ce qui réduit considérablement sa taille.

De plus, comme nous n'avons pas à installer Node.js et npm dans l'image de production (nous utilisons une image de construction séparée pour cela), le temps de construction est également réduit.



Comme vous pouvez le voir, **l'approche multi-étapes nous permet de réduire significativement la taille de l'image Docker** finale et le temps de construction. Cela rend le déploiement plus rapide et plus efficace.

Autres optimisations

Utilisation du .dockerignore

Le fichier **.dockerignore** joue un rôle similaire à celui de **.gitignore** dans les projets Git. Il permet d'**exclure certains fichiers et dossiers du processus de construction de l'image Docker**.

L'exclusion de fichiers inutiles accélère le processus de construction et réduit la taille de l'image finale. Cela peut être particulièrement utile pour exclure des fichiers temporaires, des logs ou des fichiers de développement.

Par exemple, supposons que vous ayez un dossier **node_modules** de 200MB et quelques fichiers de log de 50MB. Sans un fichier **.dockerignore**, ces fichiers seraient envoyés au démon Docker, ce qui prendrait du temps et augmenterait la taille de l'image.

```
# Exemple de .dockerignore
node_modules/
*.log
```

Dans cet exemple, le dossier `node_modules` et tous les fichiers avec l'extension `.log` seront ignorés lors de la construction de l'image Docker.

Optimisation des variables d'environnement

Les variables d'environnement peuvent être utilisées pour stocker des configurations qui varient entre les environnements. Cela vous permet d'utiliser la même image Docker dans différents contextes, ce qui est plus efficace que de créer une nouvelle image pour chaque environnement.

Au lieu de hardcoder ces valeurs dans votre Dockerfile ou votre code source, vous pouvez les passer comme variables d'environnement. Cela rend votre image plus flexible et réutilisable.

```
# Utilisation de variables d'environnement dans Dockerfile
ENV NODE_ENV=production
```

Dans cet exemple, la variable d'environnement `NODE_ENV` est définie comme `production`. Vous pouvez facilement la remplacer par `development` ou `test` en fonction de votre besoin.

Cela permet donc de construire des images différentes pour vos environnements souhaitées, ce qui pourrait vous faire **gagner plusieurs minutes de temps de construction et de réduire l'espace disque**.

Optimisation des couches de l'image

Chaque instruction dans un Dockerfile crée une nouvelle couche, et ces couches sont mises en cache pour les constructions futures. En optimisant la manière dont ces couches sont créées, vous pouvez accélérer les constructions subséquentes.

Par exemple, si l'installation d'un package prend 120 secondes et que ce package est rarement mis à jour, vous pouvez optimiser votre Dockerfile pour que cette étape soit mise en cache.

```
# Exemple d'optimisation des couches
RUN apt-get update && \
apt-get install -y package1 package2 && \
apt-get clean
```

Ici, nous avons regroupé la mise à jour du système, l'installation des paquets et le nettoyage en une seule couche pour optimiser l'utilisation du cache.

En regroupant ces commandes en une seule couche, vous assurez que cette couche est mise en cache tant que les packages ne sont pas mis à jour. Dans notre exemple, cela peut réduire le temps de construction de 120 secondes pour chaque construction future.

Conclusion

Optimiser vos images Docker est une étape cruciale pour **rendre vos déploiements plus efficaces** en termes de temps et de ressources. Comme nous l'avons vu, plusieurs techniques peuvent être employées pour réduire la taille de l'image et accélérer le temps de construction. Chaque application est unique, donc il est important de tester différentes méthodes pour trouver celles qui sont les plus efficaces pour votre cas spécifique.

En résumé, l'optimisation des images Docker n'est pas seulement une bonne pratique, mais aussi une nécessité dans les environnements de production modernes où l'efficacité et la rapidité sont essentielles. En investissant un peu de temps pour optimiser vos Dockerfiles, vous pouvez réaliser des économies significatives en termes de coûts de stockage et de temps de déploiement, tout en améliorant la performance et la fiabilité de vos applications.

N'oubliez pas que l'optimisation est un processus continu. À mesure que votre application évolue, il est bon de revenir et de réévaluer vos images Docker pour des opportunités d'optimisation supplémentaires.