


Optimisez votre déploiement en créant des conteneurs avec Docker

 openclassrooms.com/fr/courses/2035766-optimisez-votre-deploiement-en-creant-des-conteneurs-avec-docker/6211517-creez-votre-premier-dockerfile

Créez les instructions dans votre Dockerfile

La première chose que vous devez faire est de créer un fichier nommé "Dockerfile", puis de définir dans celui-ci **l'image que vous allez utiliser comme base**, grâce à l'instruction **FROM**. Dans notre cas, nous allons utiliser une image de base Debian 9.

```
FROM debian:9
```

L'instruction **FROM** n'est utilisable qu'une seule fois dans un Dockerfile.

Ensuite, utilisez l'instruction **RUN** pour exécuter une commande dans votre conteneur.

```
RUN apt-get update -yq \  
&& apt-get install curl gnupg -yq \  
&& curl -sL https://deb.nodesource.com/setup_10.x | bash \  
&& apt-get install nodejs -yq \  
&& apt-get clean -y
```

Limitez au maximum le nombre d'instructions **RUN**, afin de limiter le nombre de layers créées, et donc de réduire la taille de notre image Docker.

Puis, utilisez l'instruction **ADD** afin de copier ou de télécharger des fichiers dans l'image. Dans notre cas, nous l'utilisons pour ajouter les **sources de notre application** locale dans le dossier **/app/** de l'image.

```
ADD . /app/
```

Utilisez ensuite l'instruction **WORKDIR** qui permet de modifier le répertoire courant. La commande est équivalente à une commande **cd** en ligne de commande. L'ensemble des commandes qui suivront seront toutes exécutées depuis le répertoire défini.

```
WORKDIR /app
```

Puis, l'instruction **RUN** suivante permet d'installer le package du projet Node.js.

```
RUN npm install
```

Vous auriez pu aussi utiliser deux fois l'instruction **ADD**, et ainsi permettre à votre image d'ajouter une première fois le fichier **package.json**; puis, une fois le **npm install** réalisé, ajouter un second **ADD . /app/**. Ceci permettrait de réduire l'adhérence entre les dépendances présentes dans le fichier **package.json** et le code de l'application, ce qui permet d'économiser du temps lors du build de l'image.

Maintenant que le code source et les dépendances sont bien présents dans votre conteneur, nous devons indiquer à notre image quelques dernières informations.

```
EXPOSE 2368
VOLUME /app/logs
```

L'instruction **EXPOSE** permet d'indiquer le port sur lequel votre application écoute.
L'instruction **VOLUME** permet d'indiquer quel répertoire vous voulez partager avec votre host.

Les instructions **EXPOSE** et **VOLUME** ne sont pas nécessaires au bon fonctionnement de notre image Docker. Cependant, les ajouter permet une meilleure compréhension pour l'utilisateur des ports d'écoute attendus, ainsi que des volumes partageables.

Nous allons conclure par l'instruction qui doit toujours être présente, et la placer en dernière ligne pour plus de compréhension : **CMD** . Celle-ci permet à notre conteneur de savoir quelle commande il doit exécuter lors de son démarrage.

```
CMD npm run start
```

En résumé, voici notre Dockerfile une fois terminé :

```
FROM debian:9

RUN apt-get update -yq \
&& apt-get install curl gnupg -yq \
&& curl -sL https://deb.nodesource.com/setup_10.x | bash \
&& apt-get install nodejs -yq \
&& apt-get clean -y

ADD . /app/
WORKDIR /app
RUN npm install

EXPOSE 2368
VOLUME /app/logs

CMD npm run start
```

Créez votre fichier .dockerignore

Notre Dockerfile est maintenant prêt à fonctionner ! Cependant, il nous reste encore quelques petites modifications à faire.

Sur un projet Git, nous utilisons un fichier **.gitignore** ; sur Docker il existe le même type de fichier. Celui-ci permet de ne pas copier certains fichiers et/ou dossiers dans notre conteneur lors de l'exécution de l'instruction **ADD** .

À la racine de votre projet (soit à côté de votre fichier **Dockerfile**), vous devez créer un fichier **.dockerignore** qui contiendra les lignes suivantes :

```
node_modules
.git
```

Profitez de l'optimisation Docker

Une dernière petite chose : quand vous exécutez la commande `docker build`, **Docker va créer un conteneur pour chaque instruction**, et le résultat sera sauvegardé dans une layer. Le résultat final étant un ensemble de layers qui construisent une image Docker complète.

Mais cela apporte aussi de nombreux avantages. Si une layer ne bouge pas entre deux builds, **Docker ne la reconstruira pas**. Seules les layers situées après une layer qui se reconstruit seront elles aussi reconstruites.

Vous pouvez ainsi créer de nouvelles images très rapidement, sans devoir attendre indéfiniment le build de votre image.

Dans notre cas, si vous ajoutez une dépendance dans le fichier `package.json`, et que vous relancez un build de votre image, vous verrez qu'il n'y a que les layers situées après le `ADD package.json /app/` qui seront reconstruites ; l'installation de Node.js restera en cache.

Lancez votre conteneur personnalisé !

Vous pouvez maintenant créer votre première image Docker !

```
docker build -t ocr-docker-build .
```

L'argument `-t` permet de **donner un nom à votre image** Docker. Cela permet de retrouver plus facilement votre image par la suite.

Le `.` est le répertoire où se trouve le Dockerfile ; dans notre cas, à la racine de notre projet.

Maintenant, vous pouvez lancer votre conteneur avec la commande `docker run` :

```
docker run -d -p 2368:2368 ocr-docker-build
```

Vous retrouvez, dans le dossier `logs`, les logs de votre application, et vous pourrez y accéder sur le port `2368`, soit via l'URL <http://127.0.0.1:2368>.

En résumé

Pour créer une image Docker, vous savez utiliser les instructions suivantes :

- `FROM` qui vous permet de définir l'image **source** ;
- `RUN` qui vous permet d'exécuter des **commandes** dans votre conteneur ;
- `ADD` qui vous permet **d'ajouter des fichiers** dans votre conteneur ;

- **WORKDIR** qui vous permet de définir votre **répertoire de travail** ;
- **EXPOSE** qui permet de définir les **ports d'écoute** par défaut ;
- **VOLUME** qui permet de définir les **volumes utilisables** ;
- **CMD** qui permet de définir la **commande par défaut** lors de l'exécution de vos conteneurs Docker.

Dans le prochain chapitre, nous découvrirons comment utiliser des images grâce au partage sur le Docker Hub.

#