Créez et exécutez votre image

Temps de lecture estimé: 8 minutes

Orientation et configuration (https://docs.docker.com/get-started/part1)

Créez et exécutez votre image (https://docs.docker.com/get-started/part2)

Partager des images sur Docker Hub (https://docs.docker.com/get-started/part3)

Prérequis

Suivez l'orientation et la configuration de la partie 1 (https://docs.docker.com/get-started/) .

Introduction

Maintenant que nous avons configuré notre environnement de développement, grâce à Docker Desktop, nous pouvons commencer à développer des applications conteneurisées. En général, le flux de travail de développement ressemble à ceci:

1. Créez et testez des conteneurs individuels pour chaque composant de votre application en créant d'abord des images Docker.
2. Assemblez vos conteneurs et votre infrastructure de support dans une application complète.
3. Testez, partagez et déployez votre application conteneurisée complète.

Dans cette étape du didacticiel, concentrons-nous sur l'étape 1 de ce flux de travail: créer les images sur lesquelles nos conteneurs seront basés. N'oubliez pas qu'une image Docker capture le système de fichiers privé dans lequel nos processus conteneurisés s'exécuteront; nous devons créer une image qui contient exactement ce dont notre application a besoin pour fonctionner.

> **Les environnements de développement conteneurisés** sont plus faciles à configurer que les environnements de développement traditionnels, une fois que vous avez appris à créer des images, comme nous le verrons ci-dessous. En effet, un environnement de développement conteneurisé isolera toutes les dépendances dont votre application a besoin dans votre image Docker; il n'est pas nécessaire d'installer autre chose que Docker sur votre machine de développement. De cette façon, vous pouvez facilement développer des applications pour différentes piles sans rien changer sur votre machine de développement.

Configurer

Laissez-nous télécharger un exemple de projet à partir de la page des exemples Docker (https://github.com/dockersamples/node-bulletin-board) .

Git (/get-started/part2/#clonegit) Windows (sans Git) (/get-started/part2/#clonewin)

Mac ou Linux (sans Git) (/get-started/part2/#clonemac)

Git

Si vous utilisez Git, vous pouvez cloner l'exemple de projet à partir de GitHub:

```
git clone https://github.com/dockersamples/node-bulletin-board
cd node-bulletin-board/bulletin-board-app
```

Le `node-bulletin-board` projet est une simple application de tableau d'affichage, écrite en Node.js. Dans cet exemple, imaginons que vous avez écrit cette application et que vous essayez maintenant de la conteneuriser.

Définissez un conteneur avec Dockerfile

Jetez un œil au fichier appelé `Dockerfile` dans l'application babillard. Les Dockerfiles décrivent comment assembler un système de fichiers privé pour un conteneur, et peuvent également contenir des métadonnées décrivant comment exécuter un conteneur basé sur cette image. L'application de tableau d'affichage Dockerfile ressemble à ceci:

```
# Use the official image as a parent image
FROM node:current-slim

# Set the working directory
WORKDIR /usr/src/app

# Copy the file from your host to your current location
COPY package.json .

# Run the command inside your image filesystem
RUN npm install

# Inform Docker that the container is listening on the specified port at runtime.
EXPOSE 8080

# Run the specified command within the container.
CMD [ "npm", "start" ]

# Copy the rest of your app's source code from your host to your image filesystem.
COPY . .
```

L'écriture d'un Dockerfile est la première étape de la conteneurisation d'une application. Vous pouvez considérer ces commandes Dockerfile comme une recette étape par étape sur la façon de construire notre image. Celui-ci prend les mesures suivantes:

- Démarrez `FROM` l' `node:current-slim` image préexistante . Il s'agit d'une *image officielle* , construite par les fournisseurs de node.js et validée par Docker pour être une image de haute qualité contenant l'interpréteur Node.js Long Term Support (LTS) et les dépendances de base.
- Permet `WORKDIR` de spécifier que toutes les actions suivantes doivent être effectuées à partir du répertoire `/usr/src/app` *de votre système de fichiers image* (jamais le système de fichiers de l'hôte).
- `COPY` le fichier `package.json` de votre hôte à l'emplacement actuel (`.`) dans votre image (donc dans ce cas, à `/usr/src/app/package.json`)
- `RUN` la commande à l' `npm install` intérieur de votre système de fichiers image (qui lira `package.json` pour déterminer les dépendances des nœuds de votre application et les installer)
- `COPY` dans le reste du code source de votre application depuis votre hôte vers votre système de fichiers image.

Vous pouvez voir que ce sont les mêmes étapes que vous auriez pu prendre pour configurer et installer votre application sur votre hôte. Cependant, les capturer en tant que Dockerfile nous permet de faire la même chose dans une image Docker portable et isolée.

Les étapes ci-dessus ont constitué le système de fichiers de notre image, mais il y a d'autres lignes dans notre Dockerfile.

La `CMD` directive est notre premier exemple de spécification de certaines métadonnées dans notre image qui décrit comment exécuter un conteneur basé sur cette image. Dans ce cas, cela signifie que le processus conteneurisé que cette image est censée prendre en charge l'est `npm start` .

Le `EXPOSE 8080` informe Docker que le conteneur écoute sur le port 8080 lors de l'exécution.

Ce que vous voyez ci-dessus est un bon moyen d'organiser un Dockerfile simple; commencez toujours par une `FROM` commande, suivez-la avec les étapes pour créer votre système de fichiers privé et concluez avec toutes les spécifications de métadonnées. Il y a beaucoup plus de directives Dockerfile que juste quelques-unes que nous

voyons ci-dessus; pour une liste complète, voir la référence Dockerfile (https://docs.docker.com/engine/reference/builder/) .

Construisez et testez votre image

Maintenant que nous avons du code source et un Dockerfile, il est temps de construire notre première image et de nous assurer que les conteneurs lancés à partir de celui-ci fonctionnent comme prévu.

> **Utilisateurs Windows** : cet exemple utilise des conteneurs Linux. Assurez-vous que votre environnement exécute des conteneurs Linux en cliquant avec le bouton droit sur le logo Docker dans votre barre d'état système et en cliquant sur **Basculer vers des conteneurs Linux** si l'option apparaît. Ne vous inquiétez pas, toutes les commandes de ce didacticiel fonctionnent exactement de la même manière pour les conteneurs Windows.

Assurez-vous que vous êtes dans le répertoire `node-bulletin-board/bulletin-board-app` d'un terminal ou de PowerShell à l'aide de la `cd` commande. Construisons votre image de tableau d'affichage:

```
docker image build -t bulletinboard:1.0 .
```

Vous verrez Docker parcourir chaque instruction de votre Dockerfile, créant votre image au fur et à mesure. En cas de succès, le processus de génération doit se terminer par un message `Successfully tagged bulletinboard:1.0` .

> **Utilisateurs Windows:** vous pouvez recevoir un message intitulé «AVERTISSEMENT DE SÉCURITÉ» à cette étape, notant les autorisations de lecture, d'écriture et d'exécution définies pour les fichiers ajoutés à votre image. Nous ne traitons aucune information sensible dans cet exemple, alors n'hésitez pas à ignorer l'avertissement dans cet exemple.

Exécutez votre image comme un conteneur

1. Démarrez un conteneur basé sur votre nouvelle image:

   ```
   docker container run --publish 8000:8080 --detach --name bb bulletinboard:1.0
   ```

   Nous avons utilisé quelques drapeaux communs ici:

   - `--publish` demande à Docker de transférer le trafic entrant sur le port 8000 de l'hôte, vers le port 8080 du conteneur (les conteneurs ont leur propre ensemble privé de ports, donc si nous voulons en atteindre un à partir du réseau, nous devons le transférer de cette manière; sinon , les règles de pare-feu empêcheront tout le trafic réseau d'atteindre votre conteneur, par défaut).
   - `--detach` demande à Docker d'exécuter ce conteneur en arrière-plan.
   - `--name` nous permet de spécifier un nom avec lequel nous pouvons nous référer à notre conteneur dans les commandes suivantes, dans ce cas `bb` .

   Notez également que nous n'avons pas spécifié quel processus nous voulions que notre conteneur s'exécute. Nous n'avons pas eu à le faire, car nous avons utilisé la `CMD` directive lors de la construction de notre Dockerfile; grâce à cela, Docker sait exécuter automatiquement le processus à l' `npm start` intérieur de notre conteneur au démarrage.

2. Visitez votre application dans un navigateur à `localhost:8000` . Vous devriez voir votre application de tableau d'affichage opérationnelle. À cette étape, nous ferions normalement tout ce que nous pourrions pour que notre conteneur fonctionne comme nous l'espérions; il serait maintenant temps d'exécuter des tests unitaires, par exemple.

3. Une fois que vous êtes convaincu que votre conteneur de babillard fonctionne correctement, vous pouvez le supprimer:

```
docker container rm --force bb
```

L' `--force` option supprime le conteneur en cours d'exécution.

Conclusion

À ce stade, nous avons réussi à créer une image, effectué une simple conteneurisation d'une application et confirmé que notre application fonctionne correctement dans son conteneur. La prochaine étape sera de partager vos images sur Docker Hub (https://hub.docker.com/) , afin qu'elles puissent être facilement téléchargées et exécutées sur n'importe quelle machine de destination.

À la partie 3 >>
[https://docs.docker.com/get-started/part3/]

Références CLI

Une documentation supplémentaire pour toutes les commandes CLI utilisées dans cet article est disponible ici:

- image de docker (https://docs.docker.com/engine/reference/commandline/image/)
- conteneur docker (https://docs.docker.com/engine/reference/commandline/container/)
- Référence Dockerfile (https://docs.docker.com/engine/reference/builder/)

conteneurs (https://docs.docker.com/search/?q=containers) , images (https://docs.docker.com/search/?q=images) , dockerfiles (https://docs.docker.com/search/?q=dockerfiles) , noeud (https://docs.docker.com/search/?q=node) , code (https://docs.docker.com/search/?q=code) , codage (https://docs.docker.com/search/?q=coding) , build (https://docs.docker.com/search/?q=build) , push (https://docs.docker.com/search/?q=push) , run (https://docs.docker.com/search/?q=run)