

# Dockeriser une application Web NodeJS

Joseph AZAR – Oct 2022

Le but de cet exemple est de vous montrer comment obtenir une application Node.js dans un conteneur Docker. Le guide est destiné au développement et non à un déploiement en production. Le guide suppose également que vous disposez d'une installation Docker fonctionnelle et d'une compréhension de base de la structure d'une application Node.js.

Dans la première partie de ce guide, nous allons créer une application Web simple dans Node.js, puis nous allons créer une image Docker pour cette application, et enfin nous allons instancier un conteneur à partir de cette image.

Docker vous permet de regrouper une application avec son environnement et toutes ses dépendances dans une "boîte", appelée conteneur. En règle générale, un conteneur consiste en une application s'exécutant dans une version simplifiée d'un système d'exploitation Linux. Une image est le plan d'un conteneur, un conteneur est une instance en cours d'exécution d'une image.

## Créer l'application Node.js

- Créer un nouveau dossier pour votre projet (ex: docker-demo)
- Initialiser un projet NodeJS en utilisant "npm init"
  - `$ npm init --yes`
- Installer **express**
  - `$ npm install express`
- Mettez à jour le fichier **package.json** en définissant le point d'entrée principal comme **server.js** et ajoutez une nouvelle commande "**start**" sous "**scripts**".



```
Terminal: Local + v
(base) joseph@joseph-WP-EliteBook: /docker-demo$ npm init --y
Wrote to /docker-demo/package.json:

{
  "name": "docker-demo",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}

(base) joseph@joseph-WP-EliteBook: /docker-demo$ npm install express
added 57 packages, and audited 58 packages in 3s

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

found 0 vulnerabilities
```

```
package.json x
1 {
2   "name": "docker-demo",
3   "version": "1.0.0",
4   "description": "",
5   "main": "server.js",
6   "scripts": {
7     "start": "node server.js",
8     "test": "echo \"Error: no test specified\" && exit 1"
9   },
10  "keywords": [],
11  "author": "",
12  "license": "ISC",
13  "dependencies": {
14    "express": "^4.18.2"
15  }
16 }
```

Créez un fichier **server.js** qui définit une application Web à l'aide du framework **Express.js**:

```
$ touch server.js
```

```
const express = require('express');
```

```
// Constantes
```

```
const PORT = 3000;
```

```
const HOST = '0.0.0.0';
```

```
// App
```

```
const app = express();
```

```
app.get('/', (req, res) => {
```

```
  res.send('DOCKER APP DEMO: OK');
```

```
});
```

```
app.listen(PORT, HOST, () => {
```

```
  console.log(`Le serveur ecoute sur http://${HOST}:${PORT}`);
```

```
});
```

```
// server.js

const express = require('express');

// Constantes
const PORT = 3000;
const HOST = '0.0.0.0';

// App
const app = express();
app.get('/', (req, res) => {
  res.send('DOCKER APP DEMO: OK');
});

app.listen(PORT, HOST, () => {
  console.log(`Le serveur ecoute sur http://${HOST}:${PORT}`);
});
```

Dans les prochaines étapes, nous verrons comment exécuter cette application dans un conteneur Docker à l'aide de l'image Docker officielle. Tout d'abord, vous devez créer une image Docker de votre application.

## Création d'un Dockerfile

Docker peut créer des images portables afin que d'autres puissent exécuter notre logiciel. Il existe de nombreuses façons d'utiliser Docker, mais l'une des plus utiles consiste à créer des Dockerfiles. Ce sont des fichiers qui donnent essentiellement des instructions de construction à Docker lorsque vous créez une image de conteneur.

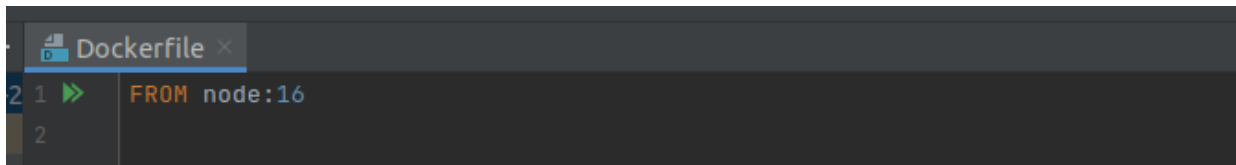
Créez un fichier vide appelé **Dockerfile** :

```
$ touch Dockerfile
```

Ouvrez le Dockerfile dans votre éditeur de texte.

La première chose que nous devons faire est de définir à partir de quelle image nous voulons construire. Ici nous utiliserons la version 16 du node disponible depuis le Docker Hub ([https://hub.docker.com/\\_/node](https://hub.docker.com/_/node)):

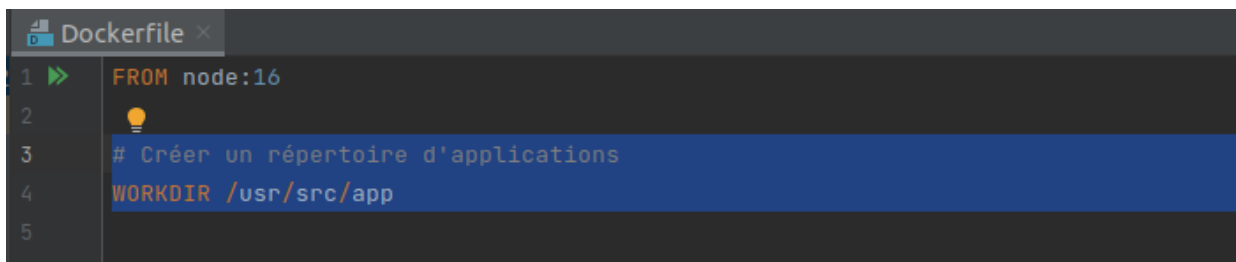
```
FROM node:16
```



```
Dockerfile x
2 1 >> FROM node:16
2
```

Ensuite, nous créons un répertoire pour contenir le code de l'application à l'intérieur de l'image, ce sera le répertoire de travail de votre application :

```
# Créer un répertoire d'applications
WORKDIR /usr/src/app
```

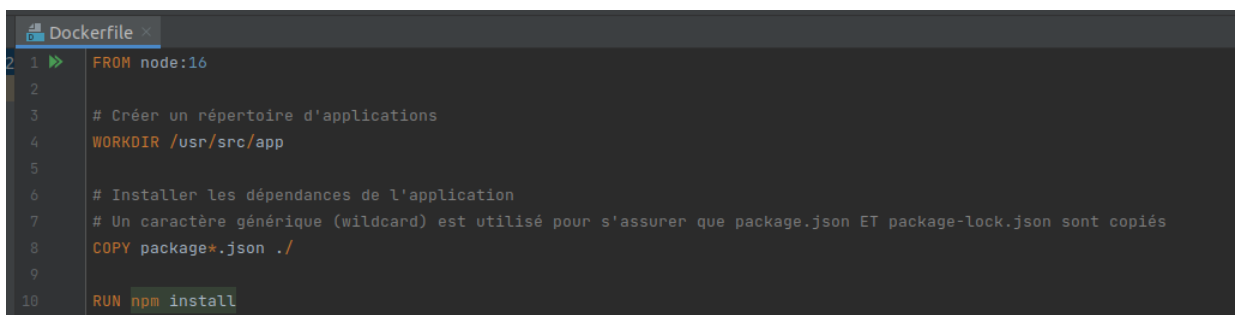


```
Dockerfile x
1 >> FROM node:16
2
3 # Créer un répertoire d'applications
4 WORKDIR /usr/src/app
5
```

Cette image est fournie avec Node.js et NPM déjà installés. La prochaine chose que nous devons faire est donc d'installer les dépendances de votre application à l'aide du binaire **npm**.

```
# Installer les dépendances de l'application
# Un caractère générique (wildcard) est utilisé pour s'assurer que package.json
# ET package-lock.json sont copiés
COPY package*.json ./
```

```
RUN npm install
```



```
Dockerfile x
2 1 >> FROM node:16
2
3 # Créer un répertoire d'applications
4 WORKDIR /usr/src/app
5
6 # Installer les dépendances de l'application
7 # Un caractère générique (wildcard) est utilisé pour s'assurer que package.json ET package-lock.json sont copiés
8 COPY package*.json ./
9
10 RUN npm install
```

Notez que, plutôt que de copier l'intégralité du répertoire de travail, nous ne copions que le fichier **package.json**. Cela nous permet de tirer parti des couches Docker mises en cache. Pour regrouper (Bundle) le code source de votre application dans l'image Docker, utilisez l'instruction **COPY** :

```
# Regroupez la source de l'application
COPY . .
```

```

1  FROM node:16
2
3  # Créer un répertoire d'applications
4  WORKDIR /usr/src/app
5
6  # Installer les dépendances de l'application
7  # Un caractère générique (wildcard) est utilisé pour s'assurer que package.json ET package-lock.json sont copiés
8  COPY package*.json ./
9
10 RUN npm install
11
12 # Regroupez la source de l'application
13 COPY . .

```

Votre application se lie au port 3000, vous utiliserez donc l'instruction **EXPOSE** pour qu'elle soit mappée par docker daemon:

```
EXPOSE 3000
```

```

1  FROM node:16
2
3  # Créer un répertoire d'applications
4  WORKDIR /usr/src/app
5
6  # Installer les dépendances de l'application
7  # Un caractère générique (wildcard) est utilisé pour s'assurer que package.json ET package-lock.json sont copiés
8  COPY package*.json ./
9
10 RUN npm install
11
12 # Regroupez la source de l'application
13 COPY . .
14
15 EXPOSE 3000

```

Définissez la commande pour exécuter votre application à l'aide de CMD qui définit votre runtime. Ici, nous allons utiliser **node server.js** pour démarrer votre serveur :

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

```

1  FROM node:16
2
3  # Créer un répertoire d'applications
4  WORKDIR /usr/src/app
5
6  # Installer les dépendances de l'application
7  # Un caractère générique (wildcard) est utilisé pour s'assurer que package.json ET package-lock.json sont copiés
8  COPY package*.json ./
9
10 RUN npm install
11
12 # Regroupez la source de l'application
13 COPY . .
14
15 EXPOSE 3000
16
17 CMD [ "node", "server.js" ]

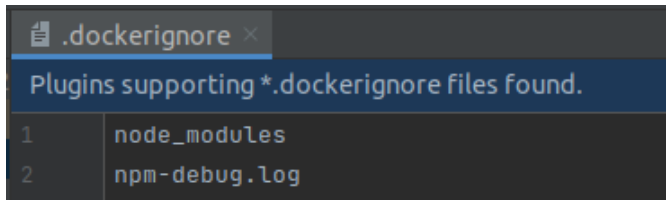
```

# Fichier .dockerignore

Créez un fichier **.dockerignore** dans le même répertoire que votre **Dockerfile** avec le contenu suivant :

```
$ touch .dockerignore
```

```
node_modules
npm-debug.log
```

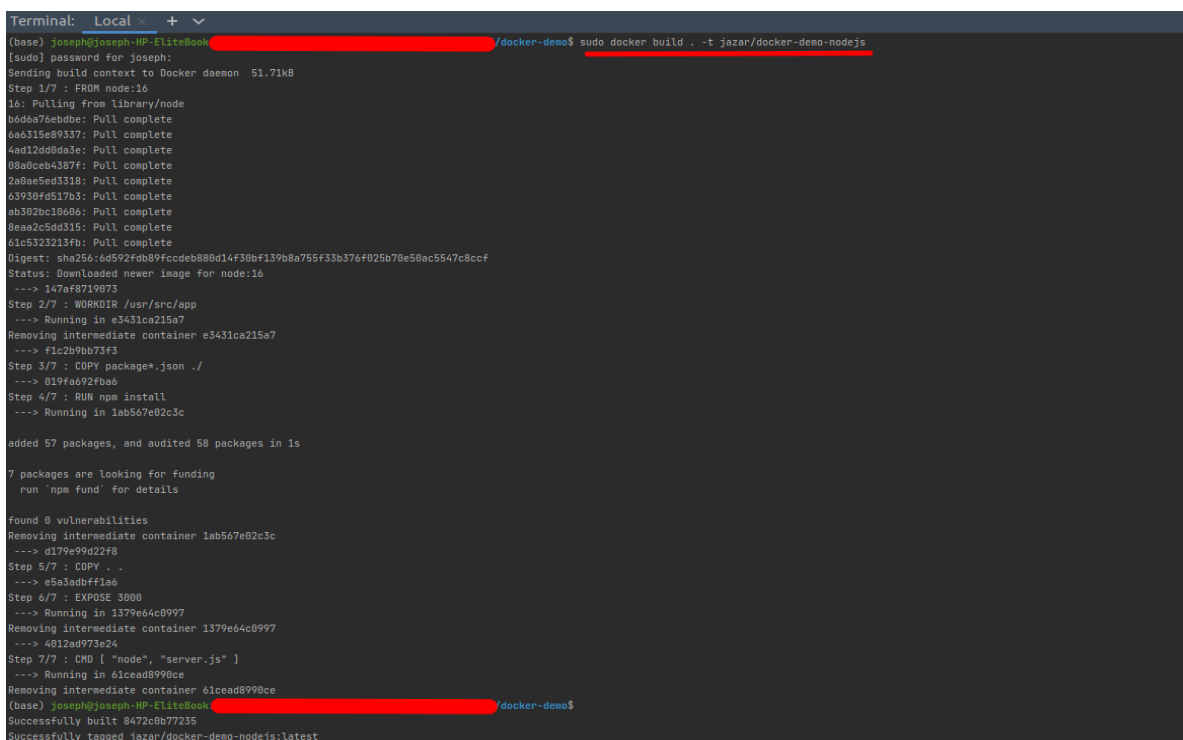


Cela empêchera vos modules locaux et vos journaux de débogage d'être copiés sur votre image Docker et éventuellement d'écraser les modules installés dans votre image.

## Construire votre image

Accédez au répertoire contenant votre **Dockerfile** et exécutez la commande suivante pour créer l'image Docker. L'indicateur **-t** vous permet de baliser votre image afin qu'elle soit plus facile à retrouver ultérieurement à l'aide de la commande “**docker images**” :

```
$ docker build . -t <votre username>/docker-demo-nodejs
```



Votre image sera désormais répertoriée par Docker :

```
$ docker images
```

```
Terminal: Local x + v
(base) joseph@joseph-HP-EliteBook: /docker-demo$ sudo docker images
REPOSITORY          TAG         IMAGE ID      CREATED       SIZE
jazar/docker-demo-nodejs  latest     8472c0b77235  17 minutes ago  916MB
```

## Exécutez l'image

L'exécution de votre image avec `-d` exécute le conteneur en mode détaché, laissant le conteneur s'exécuter en arrière-plan. L'indicateur `-p` redirige un port public vers un port privé à l'intérieur du conteneur. Exécutez l'image que vous avez précédemment créée :

```
$ docker run -p 49160:3000 -d <votre username>/docker-demo-nodejs
```

```
(base) joseph@joseph-HP-EliteBook: /docker-demo$ sudo docker run -p 49160:3000 -d jazar/docker-demo-nodejs
9b9478340c4723dfa67ed29a6cc1531d9d6daeb01892ea2aadb7ec24a27c047c
```

Imprimez le résultat de votre application :

```
$ docker ps
```

```
$ docker logs <CONTAINER ID>
```

```
(base) joseph@joseph-HP-EliteBook: /docker-demo$ sudo docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS                               NAMES
9b9478340c47   jazar/docker-demo-nodejs  "docker-entrypoint.s..." 3 minutes ago  Up 3 minutes  0.0.0.0:49160->3000/tcp, :::49160->3000/tcp  compassionate_nightingale
(base) joseph@joseph-HP-EliteBook: /docker-demo$ sudo docker logs 9b9478340c47
Le serveur écoute sur http://0.0.0.0:3000
```

Si vous devez entrer dans le conteneur, vous pouvez utiliser la commande **exec** :

```
$ docker exec -it <CONTAINER ID> /bin/bash
```

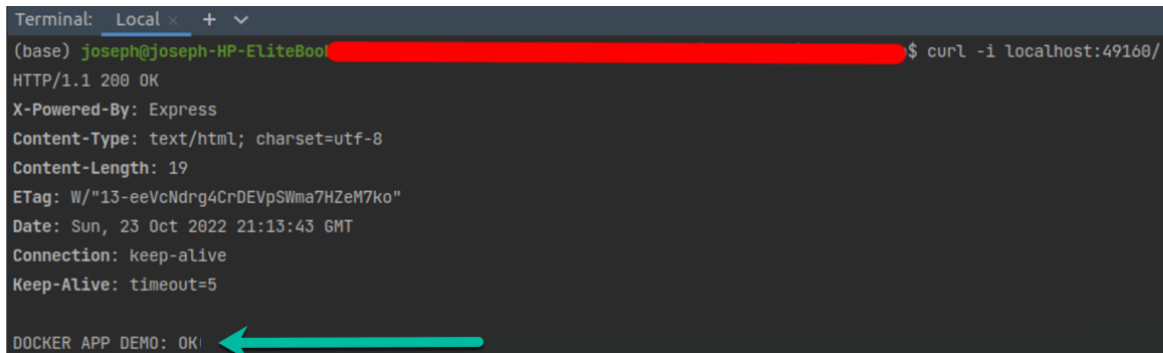
```
(base) joseph@joseph-HP-EliteBook: /docker-demo$ sudo docker exec -it 9b9478340c47 /bin/bash
root@9b9478340c47:/usr/src/app# ls
Dockerfile  node_modules  package-lock.json  package.json  server.js
```

## Tester l'application

Pour tester votre application, obtenez le port de votre application mappé par Docker (utilisez "**docker ps**"). Dans l'exemple ci-dessus, Docker a mappé le port **3000** à l'intérieur du conteneur au port **49160** sur votre machine.

Vous pouvez maintenant appeler votre application en utilisant **curl** (installer si nécessaire via : **sudo apt-get install curl**) :

```
$ curl -i localhost:49160/
```



A terminal window titled "Terminal: Local" with a red bar at the top. The prompt is "(base) joseph@joseph-HP-EliteBook". The command executed is "\$ curl -i localhost:49160/". The output is an HTTP response from Express: "HTTP/1.1 200 OK", "X-Powered-By: Express", "Content-Type: text/html; charset=utf-8", "Content-Length: 19", "ETag: W/\"13-eeVcNdrG4CrDEVpSWma7HZem7ko\"", "Date: Sun, 23 Oct 2022 21:13:43 GMT", "Connection: keep-alive", "Keep-Alive: timeout=5". The final line is "DOCKER APP DEMO: OK!" with a green arrow pointing to it.

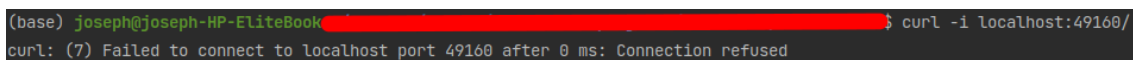
```
Terminal: Local x + v
(base) joseph@joseph-HP-EliteBook $ curl -i localhost:49160/
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: text/html; charset=utf-8
Content-Length: 19
ETag: W/"13-eeVcNdrG4CrDEVpSWma7HZem7ko"
Date: Sun, 23 Oct 2022 21:13:43 GMT
Connection: keep-alive
Keep-Alive: timeout=5
DOCKER APP DEMO: OK!
```

## Fermez l'image

Afin de fermer l'application que nous avons démarrée, nous exécutons la commande kill. Cela utilise l'ID du conteneur.

```
$ docker kill <CONTAINER ID>
```

Vous pouvez tester la commande **curl**, vous recevrez cette erreur car l'image est arrêtée :



A terminal window showing the prompt "(base) joseph@joseph-HP-EliteBook". The command executed is "\$ curl -i localhost:49160/". The output is an error message: "curl: (7) Failed to connect to localhost port 49160 after 0 ms: Connection refused".

```
(base) joseph@joseph-HP-EliteBook $ curl -i localhost:49160/
curl: (7) Failed to connect to localhost port 49160 after 0 ms: Connection refused
```