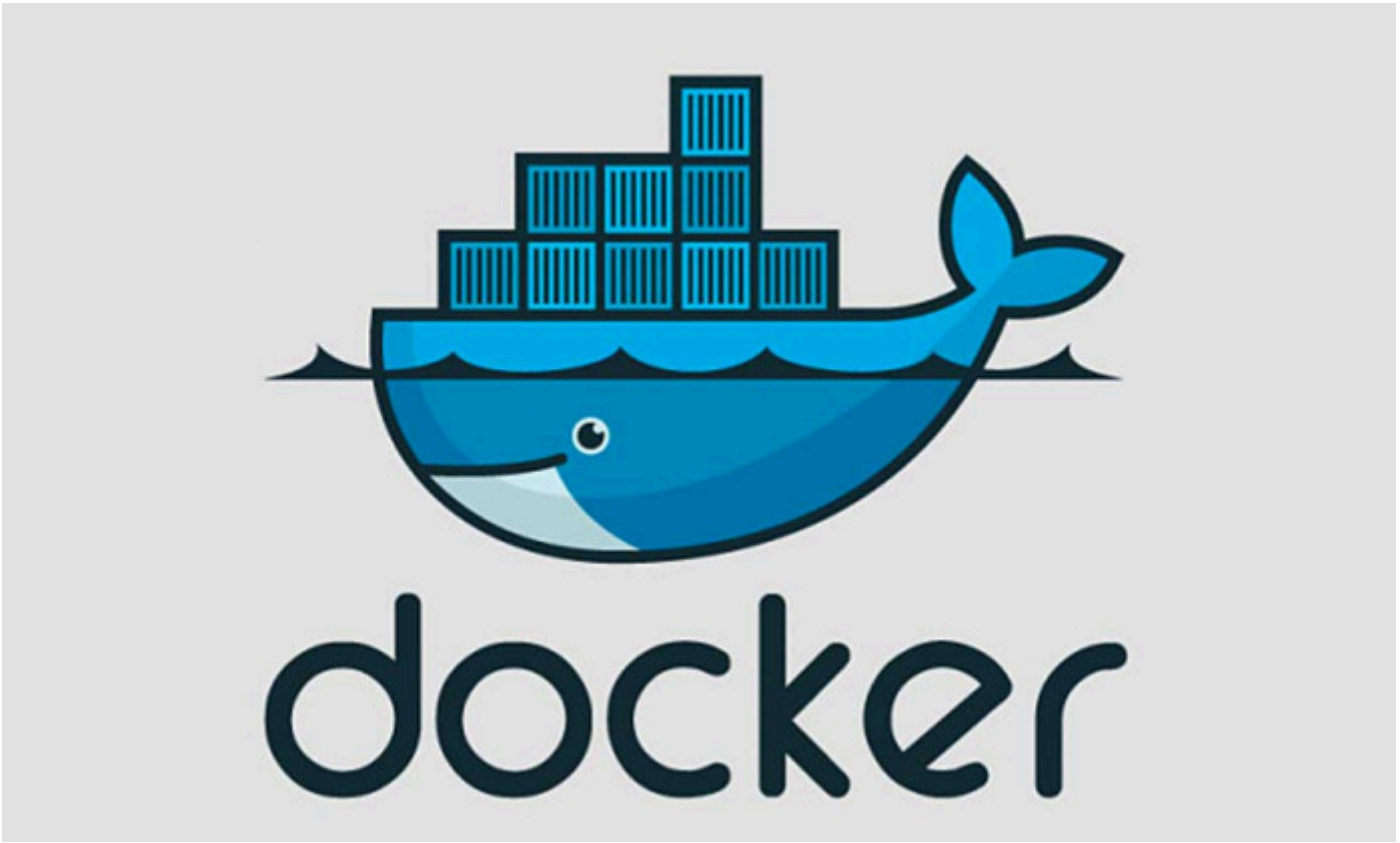
Introduction à Docker

Travaux pratiques

Concepteur et développeur d'applications



1. Création d'une image à l'aide d'un fichier Dockerfile

Au lieu de créer une image binaire statique, nous pouvons utiliser un fichier appelé Dockerfile pour créer une image. Le résultat final est essentiellement le même, mais avec un Dockerfile nous fournissons les instructions pour construire l'image, plutôt que seulement les fichiers binaires bruts. C'est utile car il devient beaucoup plus facile de gérer les changements, d'autant plus que vos images deviennent plus grandes et plus complexes.

Nous allons utiliser un exemple simple dans cette section et construire une application "hello world" dans Node.js.

Nous commencerons par créer un fichier dans lequel nous récupérerons le nom d'hôte et l'afficherons.

Pour cela, créer un répertoire dockerbis dans votre répertoire /home/user. Puis déplacez vous dedans.

Ensuite créer un fichier nommé index.js puis insérez les trois lignes suivantes :

```
var os = require("os");
var hostname = os.hostname();
console.log("hello from " + hostname);
```

Le fichier que nous venons de créer est le code javascript pour notre serveur. Comme vous pouvez probablement le devinez, Node.js affichera simplement un message « hello ». Nous allons utiliser alpine comme image de base du système d'exploitation, ajouter un runtime node.js, puis copier notre code source dans le conteneur.

Nous spécifions également la commande par défaut à exécuter lors de la création du conteneur.

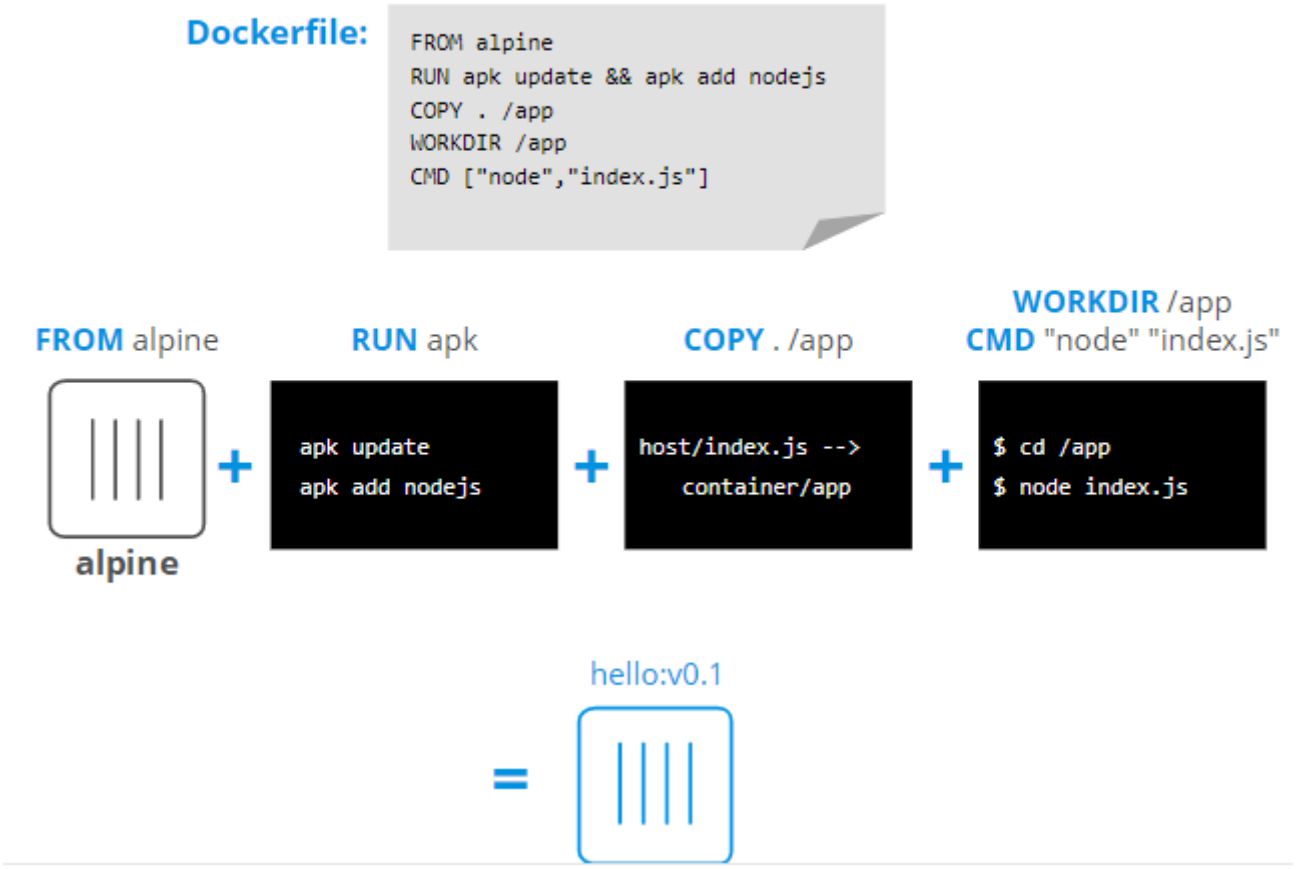
Créez un fichier nommé Dockerfile et copiez le contenu suivant dans celui-ci.

```
FROM alpine:3.11
RUN apk update && apk add nodejs
COPY . /app
WORKDIR /app
CMD ["node", "index.js"]
```

Ensuite construisons notre image à partir de ce fichier Dockerfile et nommez la hello:v1.0

```
docker image build -t hello:v1.0 .
```

Voici en image ce que réalise concrètement Docker lors de cette opération :



Ensuite, démarrez un conteneur pour vérifier que notre application s'exécute correctement :

```
docker container run hello:v1.0
```

Vous devez avoir un résultat similaire à celui-ci (évidemment l'ID du conteneur sera différent).

```
hello from 3a3a08e99341
```

Réellement que s'est-il passé ? Nous avons créé deux fichiers : le code d'application (index.js) qui contient un simple code javascript qui imprime un message. Puis un fichier Dockerfile contenant les instructions pour le moteur Docker afin de créer notre conteneur personnalisé. Ce Dockerfile fait ce qui suit :

- Spécifie une image de base à tirer depuis l'image alpine que nous avons utilisée dans les labos précédents.
- Ensuite, il exécute deux commandes (apk update et apk add) à l'intérieur de ce conteneur qui installe le serveur Node.js.
- Ensuite, nous lui avons dit de copier les fichiers de notre répertoire de travail dans le conteneur. Le seul fichier que nous avons en ce moment est notre index.js.

- Ensuite, nous spécifions WORKDIR - le répertoire que le conteneur doit utiliser lorsqu'il démarre.
- Enfin, nous avons donné à notre conteneur une commande (CMD) à exécuter lorsque le conteneur démarre.

2. Couches d’images

Il y a autre chose d'intéressant dans les images que nous construisons avec Docker. En cours d'exécution, ils semblent être un système d'exploitation unique et une application. Mais les images elles-mêmes sont en réalité composées de différentes couches.

Si vous remontez dans l’historique de votre terminal et regardez la sortie de votre commande docker image build, vous remarquerez qu'il y avait 5 étapes et chaque étape avait plusieurs tâches.

Les couches sont un concept important. Pour explorer cela, nous allons réaliser les opérations suivantes.

Tout d'abord, vérifiez l'image que vous avez créée précédemment en utilisant la commande history :

```
Docker image history <image id>
```

```
IMAGE               CREATED             CREATED BY                                      SIZE
COMMENT
8887688f668e        About a minute ago   /bin/sh -c #(nop)  CMD ["node" "index.js"]      0B
fec7c5259110        About a minute ago   /bin/sh -c #(nop) WORKDIR /app                  0B
d343cbfe870a        About a minute ago   /bin/sh -c #(nop) COPY dir:88d665d7baf244c64...   195B
6b6edaae8fcc        About a minute ago   /bin/sh -c apk update && apk add nodejs         33.4MB
e7d92cdc71fe        4 weeks ago          /bin/sh -c #(nop)  CMD ["/bin/sh"]              0B
<missing>           4 weeks ago          /bin/sh -c #(nop) ADD file:e69d441d729412d24...
5.59MB
```

Ce que vous voyez est la liste des images de conteneur intermédiaire qui ont été construites en même temps que la création de votre image finale de l'application Node.js. Certaines de ces images intermédiaires deviendront des couches dans votre image finale du conteneur.

Dans la sortie de la commande history, les calques Alpine d'origine sont en bas de la liste, puis chaque personnalisation que nous avons ajoutée dans notre Dockerfile est son propre pas dans la sortie. C'est un concept puissant car cela signifie que si nous devons apporter une modification à notre application, cela ne peut affecter qu'une seule couche ! Pour voir cela, nous allons modifier un peu notre application et créer une nouvelle image.

Tapez le texte suivant dans votre terminal :

```
echo "console.log(\"this is 2.0\");" >> index.js
```

Cela ajoutera une nouvelle ligne au bas de votre fichier index.js. Maintenant, nous allons construire une nouvelle image en utilisant notre code mis à jour. Nous allons également marquer notre nouvelle image pour la marquer comme une nouvelle version :

```
docker image build -t hello:v2.0 .
```

A l'issue de la construction de l'image vous devez avoir un résultat similaire à celui-ci :

```
roger@ubuntu-preprod:~/dockerbis$ docker build -t hello:v2.0 .
Sending build context to Docker daemon  3.072kB
Step 1/5 : FROM alpine:3.11
 ---> e7d92cdc71fe
Step 2/5 : RUN apk update && apk add nodejs
 ---> Using cache
 ---> 6b6edaae8fcc
Step 3/5 : COPY . /app
 ---> Using cache
 ---> bc2a4d56f329
Step 4/5 : WORKDIR /app
 ---> Using cache
 ---> d5fecb22ae5e
Step 5/5 : CMD ["node", "index.js"]
 ---> Using cache
 ---> cba0910a95d1
Successfully built cba0910a95d1
Successfully tagged hello:v2.0
```

Ensuite, démarrez un conteneur pour vérifier que notre application s'exécute correctement :

```
docker container run hello:v2.0
```

Vous devez avoir un résultat similaire à celui-ci :

```
hello from 975beef627bf
this is v2.0
```

3. Inspection d'une image

L'inspection d'image peut être pratique dans le cas où l'on souhaite connaître le contenu du conteneur ainsi que ses détails, les commandes qu'il peut exécuter, le système d'exploitation et plus encore.

L'image à inspecter doit être présente localement sinon, exécutez la commande suivante : docker image pull nom_de_image.

Ensuite examinons la composition de notre image :

```
docker image inspect alpine:3.11
```

Il y a pas mal d'informations :

- Les couches de l'image est composée de
- Le pilote utilisé pour stocker les calques
- L'architecture / le système d'exploitation pour lequel il a été créé
- Métadonnées de l'image
- ...

Nous n'entrerons pas dans tous les détails ici mais nous pouvons utiliser des filtres pour inspecter des détails particuliers sur l'image. De manière générale, les informations sur l'image sont au format JSON. Nous pouvons en profiter pour utiliser la commande inspect avec quelques informations de filtrage pour obtenir des données spécifiques de l'image.

Obtenons la liste des calques :

```
docker image inspect --format "{{ json .RootFS.Layers }}" alpine:3.11
```

Alpine est juste une petite image de base de l'OS donc il n'y a qu'une seule couche :

```
["sha256:5216338b40a7b96416b8b9858974bbe4acc3096ee60acbc4dfb1ee02aecceb10"]
```

Regardons notre image Hello personnalisée. Vous aurez besoin de l'identifiant de l'image :

```
docker image inspect --format "{{ json .RootFS.Layers }}" <image ID>
```

Notre image est un peu plus intéressante puisqu'elle comporte trois couches :

```
["sha256:5216338b40a7b96416b8b9858974bbe4acc3096ee60acbc4dfb1ee02aecceb10",
"sha256:3f9b90ceb06d0d28cc6e1abacac0302bd9d28a2ef2fd277485bd134dfb036166",
"sha256:905d5318efac73ac14bc43f240c53cab286f853719e4e617a7ffa1b5f408054b"]
```

3.1. Terminologie

- Calques - Une image Docker est construite à partir d'une série de calques. Chaque couche représente une instruction dans le fichier Docker de l'image. Chaque couche sauf la dernière est en lecture seule.
- Dockerfile - Un fichier texte qui contient toutes les commandes, dans l'ordre, nécessaires pour construire une image donnée. La page de référence Dockerfile répertorie les différentes commandes et les détails de format pour Dockerfiles.
- Volumes - Une couche de conteneur Docker spéciale qui permet aux données de persister et d'être partagées séparément du conteneur lui-même. Considérez les volumes comme un moyen d'abstraire et de gérer vos données persistantes séparément de l'application elle-même.

4. Docker-compose

Docker compose est un outil très intéressant de gestion de package docker. Cet outil va lancer vos conteneurs et leurs éventuels liens à partir d'un fichier de configuration écrit en yaml. Nous pourrions comparer cet outil à apt-get ou composer mais dans le but de packager des conteneurs docker. Cet outil vient simplifier la vie aux utilisateurs de docker.

4.1. Installation de Docker Compose

```
sudo apt install docker-compose
```

Pour vérifier que l'installation de docker-compose s'est bien déroulée, vous pouvez lancez la commande suivante :

```
docker-compose -v
```

Vous devez obtenir un résultat similaire à celui-ci :

```
roger@ubuntu-preprod:~/dockercompose$ docker-compose -v
docker-compose version 1.17.1, build unknown
```

https://hub.docker.com/r/bitnami/symfony

5. Exercice complémentaire

5.1. Création de l'image docker

D'abord, créons un dossier que nous appelons par exemple : basic-docker

```
mkdir basic-docker
```

Puis, créons à l'intérieur du dossier, un fichier vide que nous appelons Dockerfile.

```
touch Dockerfile
```

5.2. Le fichier Dockerfile

Il s'agit désormais d'ajouter le contenu du fichier Dockerfile. Nous commençons par réaliser l'image la plus simple possible.

Ajoutons le code suivant :

```
FROM php:7.3-alpine
CMD echo "Voici le résultat, une image très simple réalisée avec Docker"
```

La première ligne FROM indique l'image parente à laquelle nous faisons appel, nous utilisons une image de PHP pour pouvoir ensuite exécuter une simple commande.

Pour se construire, notre image fera appel à une autre image "officielle" que nous n'avons pas créée : php7.3-alpine, c'est-à-dire une image de la version 7.3 de PHP ("alpine" propose une version optimisée de PHP).

La seconde ligne CMD indique la commande qui sera exécutée par l'image : afficher (echo) le texte entre guillemets grâce à php.

5.3. Construction de l'image

Maintenant que le fichier Dockerfile est écrit, nous allons construire l'image Docker correspondante.

```
docker build -t basic .
```

Avec cet exemple, nous construisons une image avec la commande "docker build".

Puis nous lui donnons le nom (tag) "basic-docker" avec l'option "-t". Il est possible de ne pas utiliser de raccourci et d'écrire:

```
docker build --tag basic .
```

Enfin le point en fin de ligne est important, il indique que le “contexte” de la construction est le dossier dans lequel nous nous trouvons : la construction se fait depuis /basic-docker.

5.4. Exécution de l’image

L’image a été construite, elle existe mais elle n’est pas encore utilisée. Pour l’utiliser il faut passer par ce qu’on appelle un conteneur ou “container”.

Un container est créé lorsque nous utilisons la fonction run : docker run <nom de l’image>.

```
docker run basic
```

Notre conteneur ne fait pas grand chose : il écrit la ligne “Voici le résultat, une image très simple réalisée avec Docker” avant de se fermer.

5.5. Améliorons cette nouvelle image Docker PHP

5.5.1. Ajout du fichier .php

Nous allons maintenant ajouter un fichier php qui affichera toujours le même texte “Voici le résultat, une image très simple réalisée avec Docker” mais qui cette fois, sera lisible depuis un navigateur.

Nous créons donc directement à la racine de /basic-docker un nouveau fichier, cette fois il s’agit du fichier index.php dont le contenu très minimaliste se résume à :

```
<?php
echo "Voici le résultat, une image très simple réalisée avec Docker" ;
?>
```

5.5.2. Accès au fichier .php

Comment allons-nous réussir à utiliser notre fichier index.php? Pour y parvenir, nous avons besoin de passer trois étapes:

- Faire en sorte que docker puisse lire et accéder au fichier php
- Lancer un serveur php qui lira ensuite le fichier
- Accéder à l’url de ce serveur local à docker, hors du docker

5.5.3. Copie du fichier sur le container

Tout d’abord, nous devons permettre à docker d’accéder à notre fichier. Nous allons dire à docker de copier ce fichier : depuis notre disque jusqu’au disque du conteneur.

Nous allons utiliser l’instruction COPY:

```
COPY index.php /
```

Cette commande copie notre fichier pour le placer à la racine du docker /.

```
COPY <source> <dossier docker>
```

Pour éviter d'avoir à ajouter manuellement chaque fichier, nous pouvons utiliser un simple "." qui prendra en compte tout ce qui se trouve dans notre dossier /docker-basic .

```
COPY . /
```

Nous copions tous nos fichiers et dossiers "." à la racine du conteneur "/".

5.5.4. Lancement du serveur PHP depuis docker

La seconde étape consiste à interpréter le fichier index.php avec un serveur php.

Nous utilisons l'instruction "command" CMD pour faire exécuter le code suivant : php -S <host:port> <fichier php> comme nous le ferions sur notre machine locale.

```
CMD php -S 0.0.0.0:80 index.php
```

5.5.5. Dockerfile modifié

Récapitulons à présent pour voir à quoi ressemble maintenant notre Dockerfile.

Nous avons toujours notre instruction FROM à laquelle nous avons ajouté notre COPY et notre CMD.

```
FROM php:7.3-alpine

COPY . /

CMD php -S 0.0.0.0:80 index.php
```

L'host et le port renseignés sont ceux utilisés sur par docker, nous utilisons ici le localhost et le port 80 de docker, pas ceux de notre machine.

Notre image est prête à être exécutée par un conteneur.

5.5.6. Exécution de l'image

Pour finir, il nous reste plus qu'à lancer le conteneur après avoir re-buildé l'image:

```
docker build -t basic .
docker run --rm -p 82:80 basic
```

Cette fois-ci, le container ne se ferme pas automatiquement après avoir été lancé. Nous avons ajouté l'option --rm pour que le container soit automatiquement supprimé quand le container est arrêté.

Pour accéder à notre application, nous devons nous rendre sur l'adresse implicitement donnée : adresse_ip_vm:82

5.6. Création de docker-compose.yml

A la racine de notre projet, nous créons un fichier vide que nous nommons docker-compose.yml

```
touch docker-compose.yml
```

Ensuite nous ajoutons le contenu suivant au fichier

```
version: '3'
services:
  app:
    build:
      context: .
      dockerfile: Dockerfile
    image: basic
    ports:
    - 82:80
```

Passons en revue chaque partie du fichier :

Version : Il s'agit de la version de docker-compose que nous souhaitons utiliser, il est recommandé d'utiliser la version 3. Ce n'est pas la version de notre fichier.

Pour en savoir d'avantage, je vous renvoie vers la doc suivante :
https://docs.docker.com/compose/compose-file/compose-versioning/

Services

Pour l'instant nous n'avons qu'un seul "service", la base de notre application que nous avons appelé "app"; nous aurions pu choisir n'importe quel autre nom, par exemple "web" qui est parfois préféré, à vous de voir.

Build

La partie build comporte deux informations :

- context
- dockerfile

"dockerfile" indique, comme on peut le deviner, le chemin où se trouve le Dockerfile qui sera utilisé pour construire l'image du service associé (ici "app").

Le "context" : c'est le chemin à partir duquel nous exécutons nos commandes dans le docker.

Voir la doc officielle : https://docs.docker.com/compose/compose-file/#context

Image

Avec “image”, nous sommes invités à indiquer le nom de l’image qui sera construite pour notre service.

Elle sera construite à partir du fichier Dockerfile donné en paramètres de build comme indiqué plus haut.

Ports

Avec port nous mappons la correspondance entre le port utilisé dans docker et celui utilisé depuis notre machine : ici le port 82 de notre machine correspond au port 80 de docker.

5.6.1. Lancement de l’application docker-compose

Arrêtons si besoin les conteneurs qui seraient toujours actifs et dont nous désirons nous débarrasser et relançons notre commande de clean.

Nous pouvons alors lancer la commande qui va créer et exécuter notre application avec docker compose :

```
docker-compose up --build
```

Lorsque nous lançons la commande, nous pouvons bien accéder au contenu de notre application : nous voyons dans la console que le serveur PHP est actif.

Coupons notre conteneur et notre application (CTRL+C)

Nous pouvons lancer en mode détaché la même commande avec l’option -d afin de pouvoir fermer notre console tout en laissant fonctionner notre application.

5.6.2. Ajouter un volume

Certes, nous avons mis à jour notre code pour utiliser docker-compose.yml, mais notre objectif n’est toujours pas atteint : si nous modifions notre code php, les changements ne sont toujours pas visibles lorsque nous rechargeons le navigateur.

Pour répondre à notre objectif, nous allons ajouter un volume à notre fichier de configuration ce qui est très simple avec docker-compose, il nous suffit d’y ajouter la configuration “volumes:” comme ci-dessous:

```
version: '3'
services:
  app:
    build:
      context: .
      dockerfile: Dockerfile
    image: basic
    ports:
    - 82:80
    volumes:
    - .:/app/
```

Le paramètre que nous indiquons “.:/app/” correspond à un mapping entre <machine locale>:<docker>.

Ensuite ajouter la ligne ci-dessous dans le fichier Dockerfile :

```
WORKDIR /app
```

Avec notre exemple : nous lions les fichiers et dossiers se trouvant à la racine de notre application locale “.” (sur notre machine) avec un dossier app/ que nous créons sur notre environnement docker. Il y aura dans “app”, tout le contenu de notre dossier /basic-docker .

Si nous lançons à nouveau la commande docker-compose up --build nous pouvons voir que si nous modifions le code du fichier index.php et que nous rechargeons notre page http://adresse_ip_vm:82/, nous voyons les modifications prises en compte.

Pour fermer notre application, utiliser la commande :

```
docker-compose stop
```