# Formation Docker


KI '023

Ecole des Ponts Paristech

17 mars 2022

# Sommaire

# Qu'est-ce que c'est Docker ?


Concept clé : **le conteneur**.

- Bloc indépendant des autres blocs
- Je peux installer ce que je veux dans un conteneur
- Un conteneur peut interagir avec le reste du système

Différent des machines virtuelles : plus léger.

# À quoi ça sert ?


- Facilité d'installation d'un projet (sorte de "kit auto-déployé")
- Réplicabilité des conteneurs
- Isolement du système principal → non_invasif
  + indépendant de la configuration de la machine

## Exemple

***uPont :*** construction automatique d'un conteneur comportant le code et toutes les installation nécessaires pour le fonctionnement du site.

# Comment créer un conteneur ?


Au lancement du conteneur → on utilise une **image**, qui est comme une screenshot d'un conteneur.
Une **image** contient tout le contexte pour créer un conteneur :

- Disposition des fichiers
- Librairies, programmes à installer
- Ports à binder
- Commandes à exécuter au démarrage
- etc...

# Comment créer une image ?

On utilise un fichier **Dockerfile**.

*Exemple*

```
FROM nginx:1.21-alpine

RUN rm /etc/nginx/conf.d/default.conf
COPY nginx.conf /etc/nginx/conf.d
```

→ Met à disposition un ensemble de commandes pour construire une image.

Références sur le Dockerfile :
`https://docs.docker.com/engine/reference/builder/`

# Comment créer une image ?

| | |
|---|---|
| FROM | Importe une image pré-existante (https ://-hub.docker.com/). |
| COPY | Copie des fichiers depuis le système d'exploitation vers le conteneur. |
| WORKDIR | Pour se déplacer dans le conteneur. |
| RUN | Exécuter une commande *bash*. |
| ENV | Donne une valeur à une variable d'environnement dans le conteneur. |
| ENTRYPOINT | Pour préciser des commandes à exécuter au démarrage. |
| CMD | Pour préciser des commandes à exécuter au démarrage. |

Table – Principales commandes dans un Dockerfile

# Chemins dans le système et dans le conteneur.


Pour COPY : Chemin dans le système en premier argument, chemin dans le conteneur en second argument.

```
# On va dans le dossier /etc du conteneur
WORKDIR /etc
# On copie le fichier nginx.conf dans le dossier
# /etc/nginx du conteneur et on renomme le fichier
COPY nginx.conf ./nginx/conf.d
```

# Bonus : staged build

→ Créer plusieurs images dans le même Dockerfile.

```
FROM python as base
# Do things here

FROM base as production
# Do things here

FROM base as development
# Do things here
```

Ici on crée 3 images : base, production et development.

- Met en cache certaines parties du build (plus rapide lors de modifications du Dockerfile)
- Images différentes pour la production et le développement

# Comment gérer plusieurs conteneurs ?

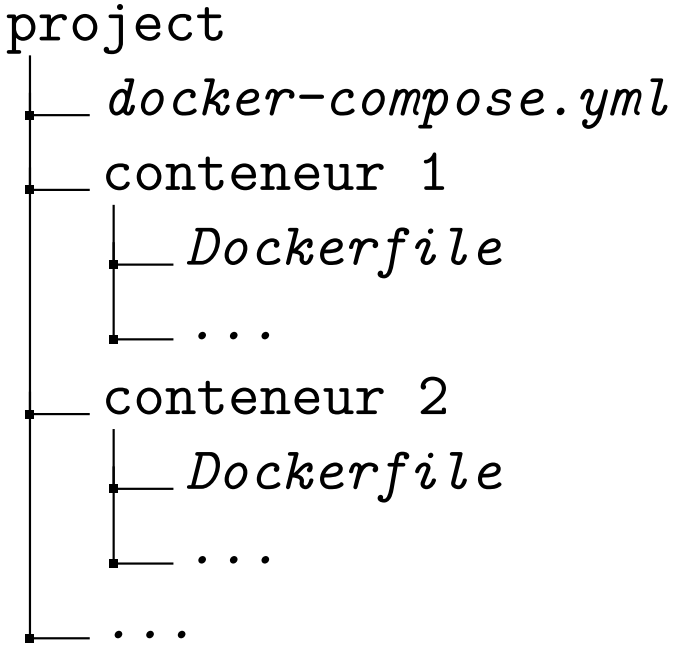

Exemple sur *uPont* :

- Base de données
- Serveur Python
- Compilateur pour créer des bundles javascript
- Serveur web et proxy

→ On ne veut pas avoir à lancer tous ces conteneurs un par un.

**Solution** : docker-compose (`sudo apt install docker-compose`)

# Structure des dossiers



# Fichier docker-compose.yml


```yaml
services:
  db:
    image: postgres
    volumes:
      - "./database:/var/lib/postgresql/data"
    restart: always
    environment:
      POSTGRES_DB: $DB_NAME
      POSTGRES_USER: $DB_USER
      POSTGRES_PASSWORD: $DB_PASSWORD
    networks:
      - db_link
  back:
    build:
      context: back
      dockerfile: Dockerfile
      target: production
    command: gunicorn upont.wsgi:application --bind 0.0.0.0:8000
    expose:
      - 8000
    depends_on:
      - db
      - webinstaller
    volumes:
      - static_volume:/src/static
      - media_volume:/src/media
      - bundles:/src/upont/static/react
```

# Fichier docker-compose.yml


```
    restart: always
    environment:
      DB_HOST: $DB_HOST
      DB_PORT: $DB_PORT
      DB_NAME: $DB_NAME
      DB_USER: $DB_USER
      DB_PASSWORD: $DB_PASSWORD
      ...
    networks:
      - ping
      - db_link
      - nginx_link
volumes:
  static_volume:
  media_volume:
  database:
  bundles:
    driver: local
networks:
  ping:
    driver: bridge
  db_link:
    driver: bridge
  nginx_link:
        driver: bridge
```

# Fichier docker-compose.yml

| | |
|---|---|
| services | Liste des conteneurs à créer. |
| image | Image à utiliser pour un conteneur (pas de fichier Dockerfile dans ce cas). |
| build | Instructions pour le création de l'image. |
| context | Dossier dans lequel se trouve les fichiers liés à un conteneur. |
| target | Image à utiliser quand plusieurs images sont définies dans le Dockerfile. |
| environment | Variables d'environnement créées au lancement. |

Table – Principales directives dans un docker-compose.yml

# Fichier docker-compose.yml

| | |
|---|---|
| ports | Permet de lier des ports du système à des ports internes aux conteneurs. |
| expose | Permet de rendre un port du conteneur accessible aux autres conteneurs. |
| depends_on | Conteneurs qui seront démarrés avant le conteneur en question. |

Table – Principales directives dans un docker-compose.yml

Documentation complète : `https://docs.docker.com/compose/compose-file/compose-file-v3/`

# Commandes usuelles


Construire les images des conteneurs :

```
docker-compose build
```

Lancer les conteneurs (-d = en mode détaché) :

```
docker-compose up -d
```

Voir les logs :

```
docker-compose logs --follow
```

# Fichier .env

```
DB_HOST=db
DB_PORT=5432
DB_USER=upont
DB_NAME=upont
DB_PASSWORD=upont
...
```

- Variables utilisées au lancement des conteneurs
- Valeurs souvent secrètes : commit des valeurs de test
- Toutes les variables sont centralisées dans le même fichier

# Les volumes


- Échanger des données entre les conteneurs
- Assurer une persistance des données
- Modifier le code sans reconstruire l'image (en développement)

Présentation

Dockerfile, image et conteneur

docker-compose

TP

# À vous de jouer !

On va faire tourner un petit script python dans un conteneur, en utilisant Docker et docker-compose.

Architecture de fichiers :

```
passa
├── docker-compose.yml
└── python
    ├── Dockerfile
    ├── run.py
    └── requirements.txt
```

# À vous de jouer !


1. Créez l'architecture de fichiers.
2. Définissez un conteneur python.
3. Dans requirements.txt, mettez la liste des packages à installer (par exemple, `numpy`).
4. Dans le Dockefile :
   - 4.1 Basez-vous sur une image de python (`https://hub.docker.com/_/python`).
   - 4.2 Créez un dossier `/src` dans le conteneur et placez-y le contenu du dossier python.
   - 4.3 Installez les packages python :
     ```
     pip install -r requirements.txt
     ```
   - 4.4 Définissez comme commande d'entrée du conteneur :
     ```
     python run.py
     ```
5. Faites faire n'importe quoi à votre script run.py, puis testez votre projet.

# BISOUS

Le KI '023