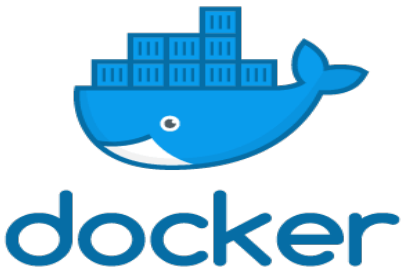
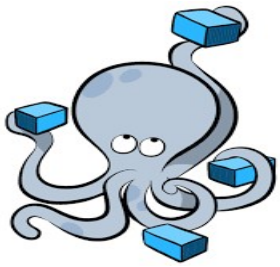
Approfondissement traitant des questions liées à la professionnalisation de l'utilisation de Docker

Containerized Python Development





Containerized Python Development – Part 1

Le développement de projets Python dans des environnements locaux peut devenir assez difficile si plusieurs projets sont en cours de développement en même temps. L'amorçage d'un projet peut prendre du temps car nous devons gérer les versions, configurer les dépendances et les configurations pour celui-ci. Avant, nous avions l'habitude d'installer toutes les exigences du projet directement dans notre environnement local, puis de nous concentrer sur l'écriture du code. Mais avoir plusieurs projets en cours dans le même environnement devient rapidement un problème car nous pouvons entrer dans des conflits de configuration ou de dépendance. De plus, lors du partage d'un projet avec des coéquipiers, nous devons également coordonner nos environnements. Pour cela, nous devons définir l'environnement de notre projet de manière à ce qu'il soit facilement partageable.

Un bon moyen d'y parvenir est de créer des environnements de développement isolés pour chaque projet. Cela peut être facilement fait en utilisant des conteneurs et Docker Compose pour les gérer.

Conteneuriser un service / outil Python et les meilleures pratiques pour celui-ci.

Requirements

Installez Docker puis Docker Compose

Présentation de travail

Nous montrons comment faire cela avec un service Flask simple de sorte que nous puissions l'exécuter de manière autonome sans avoir besoin de configurer d'autres composants.

```
#server.py
From flask import Flask
server = Flask(__name__)

@server.route("/")
def hello():
 return "Hello World!"

if __name__ == "__main__":
 server.run(host='0.0.0.0')
```

Pour exécuter ce programme, nous devons nous assurer que toutes les dépendances requises sont installées en premier. Une façon de gérer les dépendances consiste à utiliser un programme d'installation de package tel que **pip.** Pour cela, nous devons créer un fichier **requirements.txt** et y écrire les dépendances.

```
requirements.txt
Flask==1.1.1
```

Q. Réalisez l’arborescence ci-dessous

```
app
├── requirements.txt
└── src
    └── server.py
```

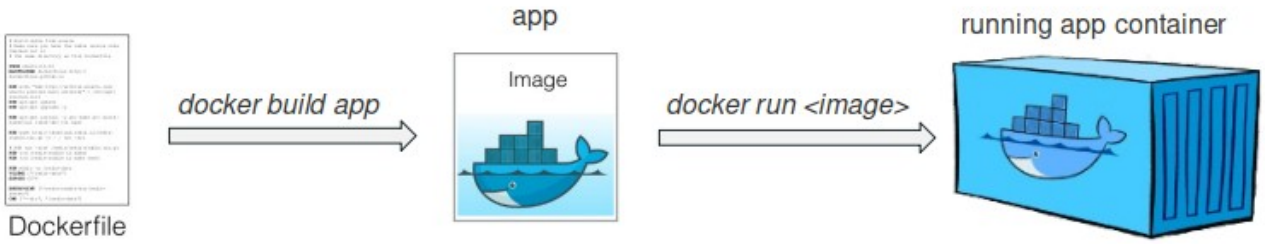
Nous créons un répertoire dédié pour le code source afin de l'isoler des autres fichiers de configuration.

Pour exécuter notre programme Python, il ne reste plus qu'à installer un interpréteur Python et à l'exécuter.

Nous pourrions exécuter ce programme localement. Mais cela va à l'encontre du but de la conteneurisation de notre développement qui est de conserver un environnement de développement standard propre qui nous permet de basculer facilement entre des projets avec des exigences contradictoires différentes.

Dockerfile

La façon d'exécuter notre code Python dans un conteneur consiste à le conditionner en tant qu'image Docker, puis à exécuter un conteneur basé sur celle-ci. Les étapes sont esquissées ci-dessous.



Analyse d'un Dockerfile

Voici un exemple de Dockerfile contenant des instructions pour assembler une image Docker pour notre service Python hello world:

```
# set base image (host OS)
FROM python:3.8
# set the working directory in the container
WORKDIR /code
# copy the dependencies file to the working directory
COPY requirements.txt .
# install dependencies
RUN pip install -r requirements.txt
# copy the content of the local src directory to the working directory
COPY src/ .
# command to run on container start
CMD [ "python", "./server.py" ]
```

Pour chaque instruction ou commande du Dockerfile, le Docker Builder génère un calque d'image et l'empile sur les précédents. Par conséquent, l'image Docker résultant du processus est simplement une pile en lecture seule de différentes couches.

Nous pouvons observer dans la sortie de la commande build que les instructions Dockerfile s’ exécutent comme des étapes.

Ensuite, nous pouvons vérifier que l'image est dans le store d'images local:

```
$docker images
REPOSITORY      TAG         IMAGE ID        CREATED          SIZE
myimage         latest      0a92e92f3b5     8 seconds ago    991MB
```

Meilleures pratiques de développement pour Dockerfiles

Image de base

La première instruction du Dockerfile spécifie l'image de base sur laquelle nous ajoutons de nouvelles couches pour notre application. Le choix de l'image de base est assez important car les fonctionnalités qu'elle embarque peuvent avoir un impact sur la qualité des couches construites dessus.

Lorsque cela est possible, nous devrions toujours utiliser des images officielles qui sont en général fréquemment mises à jour et peuvent avoir moins de problèmes de sécurité.

Le choix d'une image de base peut avoir un impact sur la taille de l'image finale. Si nous préférons la taille à d'autres considérations, nous pouvons utiliser certaines des images de base de très petite taille. Ces images sont généralement basées sur la distribution alpine et sont étiquetées en conséquence. Cependant, pour les applications Python, la variante slim de l'image officielle Docker Python fonctionne bien dans la plupart des cas (par exemple, python: 3.8-slim).

L'ordre des instructions est important pour tirer parti du cache de compilation.

Lors de la création fréquente d'une image, nous souhaitons absolument utiliser le mécanisme de cache du générateur pour accélérer les versions ultérieures.

Pour une utilisation efficace du mécanisme de mise en cache, nous devons placer les instructions pour les couches qui changent fréquemment après celles qui subissent le moins de changements.

```
...
# copy the dependencies file to the working directory
COPY requirements.txt .
# install dependencies
RUN pip install -r requirements.txt

# copy the content of the local src directory to the working directory
COPY src/ .
...
```

Au cours du développement, les dépendances de notre application changent moins fréquemment que le code Python. Pour cette raison, nous choisissons d'installer les dépendances dans une couche précédant celle de code. Par conséquent, nous copions le fichier de dépendances et les installons, puis nous copions le code source. C'est la raison principale pour laquelle nous avons isolé le code source dans un répertoire dédié dans la structure de notre projet.

Multi-stage builds

Bien que cela puisse ne pas être vraiment utile pendant le développement, il l'est une fois le développement terminé.

Ce que nous cherchons en utilisant des versions en plusieurs étapes est de dépouiller l'image finale de l'application de tous les fichiers et packages logiciels inutiles et de ne fournir que les fichiers nécessaires pour exécuter notre code Python. Voici un exemple rapide de fichier Dockerfile à plusieurs étapes pour notre exemple précédent:

```
# first stage
FROM python:3.8 AS builder
COPY requirements.txt .
# install dependencies to the local user directory (eg. /root/.local)
RUN pip install --user -r requirements.txt

# second unnamed stage
FROM python:3.8-slim
WORKDIR /code
# copy only the dependencies installation from the 1st stage image
COPY --from=builder /root/.local/bin /root/.local
COPY ./src .

# update PATH environment variable
ENV PATH=/root/.local:$PATH
CMD [ "python", "./server.py" ]
```

Notez que nous avons une construction en deux étapes où nous ne nommons que la première en tant que builder. Nous nommons une étape en ajoutant un AS <NAME> à l'instruction FROM et nous utilisons ce nom dans l'instruction COPY où nous voulons copier uniquement les fichiers nécessaires dans l'image finale.

Le résultat est une image finale plus mince pour notre application:

```
$ docker images
REPOSITORY    TAG      IMAGE ID       CREATED         SIZE
myimage       latest   0a92e92f3b5   2 hours ago     991MB
multistage    latest   e598271edefa   6 minutes ago   197MB
…
```

Dans cet exemple, nous nous sommes appuyés sur l'option --user de pip pour installer les dépendances dans le répertoire utilisateur local et copier ce répertoire dans l'image finale.

Exécuter le conteneur

```
$ docker images
REPOSITORY      TAG        IMAGE ID        CREATED         SIZE
myimage         latest     0a92e92f3b5     8 seconds ago   991MB
...

$ docker ps
CONTAINER ID   IMAGE   COMMAND   CREATED   STATUS   PORTS   NAMES
```

```
$ docker run -d -p 5000:5000 myimage
befb1477c1c7fc31e8e8bb8459fe05bcbdee2df417ae1d7c1d37f371b6fbf77f
```

Nous avons maintenant conteneurisé notre serveur hello world et nous pouvons interroger le port mappé sur localhost.

```
$ docker ps
CONTAINER     ID        IMAGE         COMMAND         PORTS
befb1477c1c7  myimage   "/bin/sh -c  'python ..."   0.0.0.0:5000->5000/tcp  ...

$ curl http://localhost:5000
"Hello World!"
```

Containerized Python Development – Part 2

Nous avons déjà montré comment conteneuriser un service Python et les meilleures pratiques pour celui-ci. Dans cette partie, nous expliquons comment configurer et câbler d'autres composants à un service Python en conteneur. Nous montrons un bon moyen d'organiser les fichiers et les données du projet et comment gérer la configuration globale du projet avec Docker Compose. Nous couvrons également les meilleures pratiques d'écriture de fichiers Compose pour accélérer notre processus de développement conteneurisé.

Gestion de la configuration du projet avec Docker Compose

Présentation de travail à faire

Prenons comme exemple une application pour laquelle nous séparons ses fonctionnalités en trois niveaux suivant une architecture de microservices. Il s'agit d'une architecture assez courante pour les applications multi-services. Notre exemple d'application se compose de:

- un niveau d'interface utilisateur - s'exécutant sur un service nginx
- un niveau logique - le composant Python sur lequel nous nous concentrons
- un niveau de données - nous utilisons une base de données mysql pour stocker certaines données dont nous avons besoin dans le niveau logique

La raison de la division d'une application en niveaux est que nous

pouvons facilement en modifier ou en ajouter de nouvelles fonctionnalités sans avoir à retravailler l'ensemble du projet.

Un bon moyen de structurer les fichiers de projet consiste à isoler le fichier et les configurations pour chaque service. Nous pouvons facilement le faire en ayant un répertoire dédié par service dans celui du projet. Ceci est très utile pour avoir une vue claire des composants et pour conteneuriser facilement chaque service. Cela aide également à manipuler les fichiers spécifiques au service sans avoir à craindre que nous puissions modifier par erreur d'autres fichiers de service.
Pour notre exemple d'application, nous avons les répertoires suivants:

```
Project
├─── web
├─── app
│  ├─── Dockerfile
│  ├─── requirements.txt
│  └─── src
│  └─── server.py
└─── db
```

Nous pourrions maintenant démarrer les conteneurs manuellement pour tous nos composants de projet conteneurisés. Cependant, pour les faire communiquer, nous devons gérer manuellement la création du réseau et y attacher les conteneurs. C'est assez compliqué et cela prendrait un temps de développement précieux si nous devons le faire fréquemment.

C'est ici que Docker Compose offre un moyen très simple de coordonner les conteneurs et de créer et supprimer des services dans notre environnement local. Pour cela, il suffit d'écrire un fichier Compose contenant la configuration des services de notre projet. Une fois que nous l'avons, nous pouvons lancer le projet avec une seule commande.

Composer un fichier

Voyons quelle est la structure des fichiers Compose et comment nous pouvons gérer les services de projet avec eux.

Voici un exemple de fichier pour notre projet. Comme vous pouvez le voir, nous définissons une liste de services. Dans la section db, nous spécifions l'image de base directement car nous n'avons aucune configuration particulière à lui appliquer. Pendant ce temps, notre service Web et d'application va avoir l'image construite à partir de leurs Dockerfiles. Selon l'endroit où nous pouvons obtenir l'image de service, nous pouvons définir la construction ou le champ d'image. Le champ de construction nécessite un chemin avec un Dockerfile à l'intérieur.

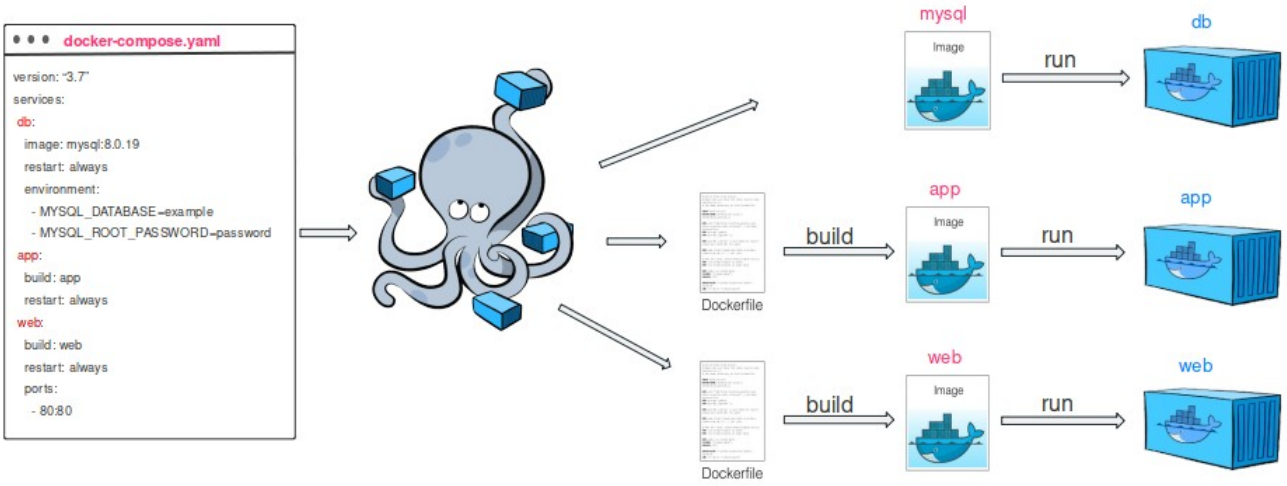
```
version: "3.7"
services:
  db:
    image: mysql:8.0.19
    command: '--default-authentication-plugin=mysql_native_password'
    restart: always
    environment:
      - MYSQL_DATABASE=example
      - MYSQL_ROOT_PASSWORD=password

  app:
    build: app
    restart: always

  web:
    build: web
```

```
    restart: always
    ports:
      - 80:80
```

Pour initialiser la base de données, nous pouvons passer des variables d'environnement avec le nom de la base de données et le mot de passe tandis que pour notre service Web, nous mappons le port du conteneur vers l'hôte local afin de pouvoir accéder à l'interface Web de notre projet.
Voyons comment déployer le projet avec Docker Compose.
Tout ce que nous devons faire maintenant est de placer le fichier docker-compose.yaml dans le répertoire racine du projet, puis d'émettre la commande de déploiement avec docker-compose.

```
Project
├─── docker-compose.yaml
├─── web
├─── app
└─── db
```

Docker Compose se chargera d'extraire l'image mysql de Docker Hub et de lancer le conteneur de base de données tandis que pour notre service Web et d'application, il crée les images localement, puis exécute les conteneurs à partir d'elles. Il s'occupe également de créer un réseau par défaut et d'y placer tous les conteneurs afin qu'ils puissent se rejoindre.



Tout cela est déclenché avec une seule commande.

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

Vérifiez les conteneurs en cours d'exécution:

```
$ docker-compose ps
   Name         Command                       State  Ports
-------------------------------------------------------------------------
project_app_1  /bin/sh -c python server.py    Up
project_db_1   docker-entrypoint.sh --def ... Up     3306/tcp, 33060/tcp
project_web_1  nginx -g daemon off;           Up     0.0.0.0:80→80/tcp
```

Pour arrêter et supprimer tous les conteneurs de projet, exécutez:

```
$ docker-compose down
Stopping project_db_1 ... done
Stopping project_web_1 ... done
Stopping project_app_1 ... done
Removing project_db_1 ... done
Removing project_web_1 ... done
Removing project_app_1 ... done
Removing network project-default
```

Pour reconstruire les images, nous pouvons exécuter une compilation, puis une commande up pour mettre à jour l'état des conteneurs du projet:

```
$ docker-compose build
$ docker-compose up -d
```

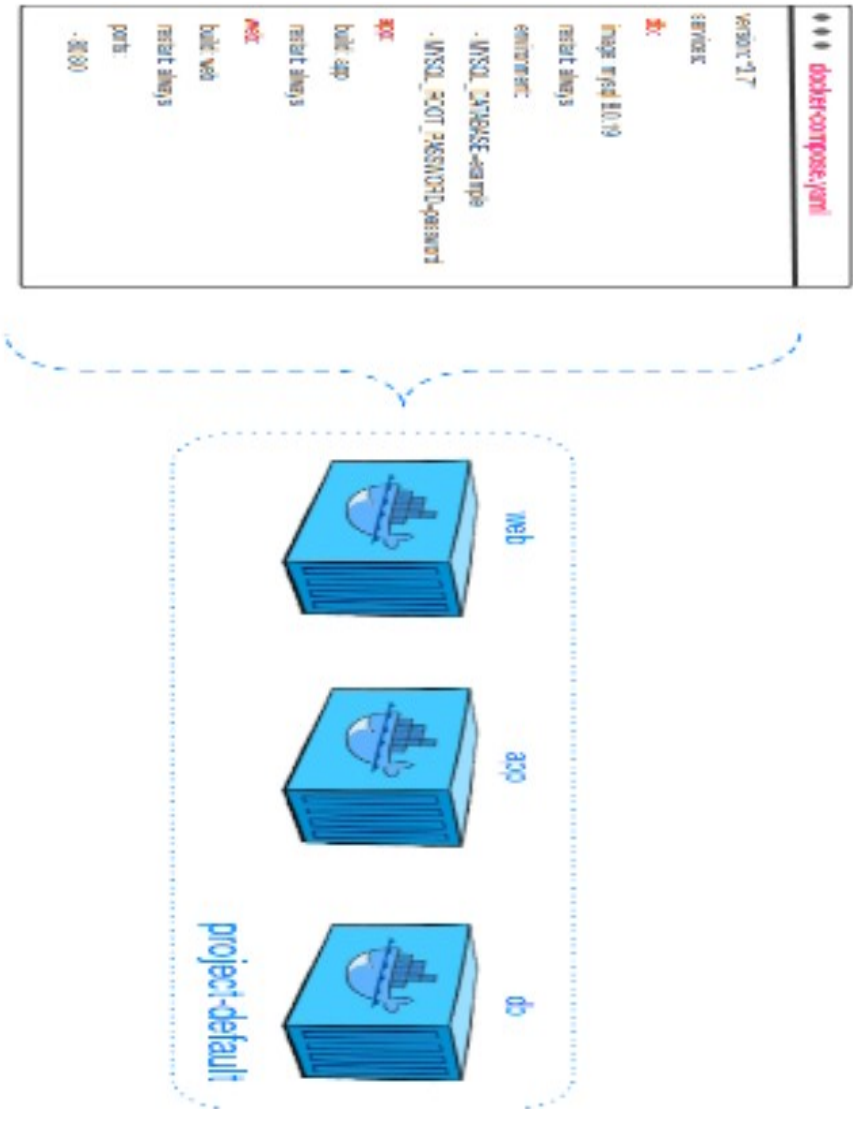
Comme on peut le voir, il est assez facile de gérer le cycle de vie des conteneurs de projet avec docker-compose.

Meilleures pratiques pour l'écriture de fichiers Compose

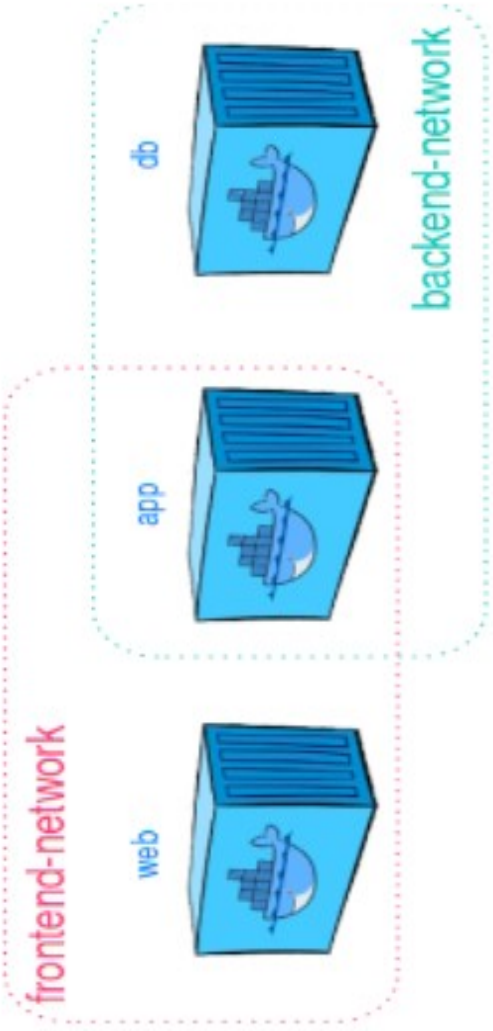
Analysons le fichier Compose et voyons comment nous pouvons l'optimiser en suivant les meilleures pratiques pour écrire des fichiers Compose.

Séparation du réseau

Lorsque nous avons plusieurs conteneurs, nous devons contrôler la manière de les câbler ensemble. Nous devons garder à l'esprit que, comme nous ne définissons aucun réseau dans le fichier de composition, tous nos conteneurs se termineront dans le même réseau par défaut.



Ce n'est peut-être pas une bonne chose si nous voulons que seul notre service Python puisse accéder à la base de données. Pour résoudre ce problème, dans le fichier de composition, nous pouvons en fait définir des réseaux séparés pour chaque paire de composants. Dans ce cas, le composant Web ne pourra pas accéder à la base de données.




```
docker-compose.yaml
version: '3.7'
services:
  db:
    image: mysql:8.0.19
    ...
    networks:
      - backend-network
  app:
    build: app
    ...
    networks:
      - backend-network
      - frontend-network
  web:
    build: web
    ...
    networks:
      - frontend-network

networks:
  backend-network:
  frontend-network:
```

Docker Volumes

Chaque fois que nous démontons nos conteneurs, nous les supprimons et perdons donc les données que nous avons stockées lors des sessions précédentes. Pour éviter cela et conserver les données de base de données entre différents conteneurs, nous pouvons exploiter des volumes nommés.

Pour cela, nous définissons simplement un volume nommé dans le fichier Compose et spécifions un point de montage pour celui-ci dans le service db comme indiqué ci-dessous:

```
version: "3.7"
services:
  db:
    image: mysql:8.0.19
    command: '--default-authentication-plugin=mysql_native_password'
    restart: always
    volumes:
      - db-data:/var/lib/mysql
    networks:
      - backend-network
    environment:
      - MYSQL_DATABASE=example
      - MYSQL_ROOT_PASSWORD=password

  app:
    build: app
    restart: always
    networks:
      - backend-network
      - frontend-network

  web:
    build: web
    restart: always
    ports:
      - 80:80
    networks:
      - frontend-network
volumes:
  db-data:
networks:
  backend-network:
  frontend-network:
```

Docker Secrets

Comme nous pouvons l'observer dans le fichier Compose, nous définissons le mot de passe db en texte brut. Pour éviter cela, nous pouvons exploiter les Docker secrets pour stocker le mot de passe et le partager en toute sécurité avec les services qui en ont besoin. Nous pouvons définir des secrets et les référencer dans les services comme ci-dessous. Le mot de passe est stocké localement dans le fichier project/db/password.txt et monté dans les conteneurs sous /run/secrets/<secret-name>.

```
version: "3.7"
services:
```

```
  db:
    image: mysql:8.0.19
    command: '--default-authentication-plugin=mysql_native_password'
    restart: always
    secrets:
      - db-password
    volumes:
      - db-data:/var/lib/mysql
    networks:
      - backend-network
    environment:
      - MYSQL_DATABASE=example
      - MYSQL_ROOT_PASSWORD_FILE=/run/secrets/db-password

  app:
    build: app
    restart: always
    secrets:
      - db-password
    networks:
      - backend-network
      - frontend-network

  web:
    build: web
    restart: always
    ports:
      - 80:80
    networks:
      - frontend-network
volumes:
  db-data:
secrets:
  db-password:
    file: db/password.txt
networks:
  backend-network:
  frontend-network:
```

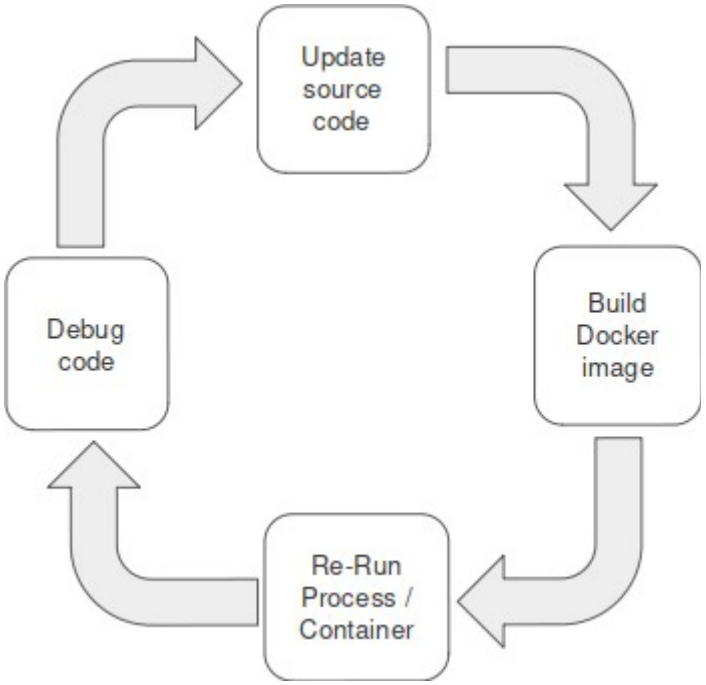
Nous avons maintenant un fichier Compose bien défini pour notre projet qui suit les meilleures pratiques.

Containerized Python Development – Part 3

Dans cette dernière partie, nous passons en revue le cycle de développement du projet et discutons plus en détail comment appliquer les mises à jour de code et les échecs de débogage des services Python conteneurisés. L'objectif est d'analyser comment accélérer ces phases récurrentes du processus de développement de manière à obtenir une expérience similaire à celle du développement local.

Application des mises à jour de code

En général, notre cycle de développement conteneurisé consiste à écrire / mettre à jour le code, à le construire, à l'exécuter et à le déboguer.



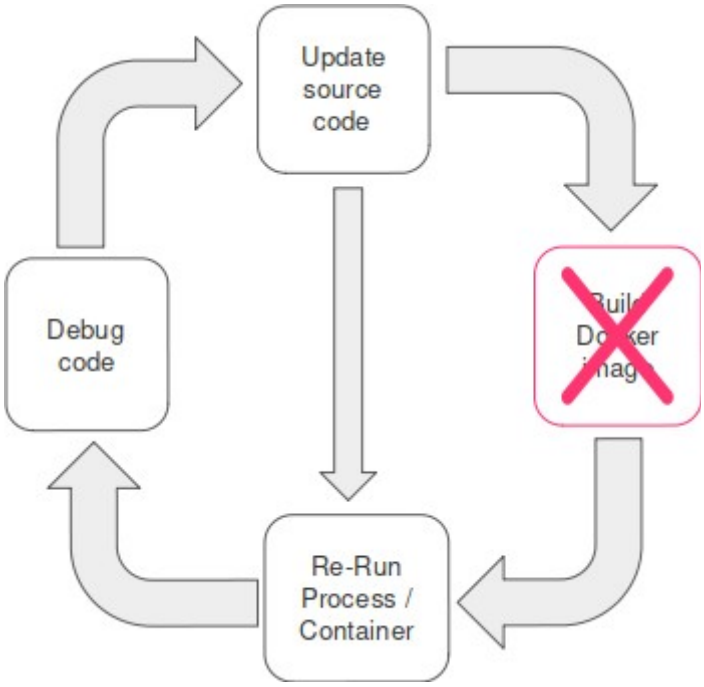
Pour la phase de construction et d'exécution, comme la plupart du temps nous devons attendre, nous voulons que ces phases se déroulent assez rapidement afin de nous concentrer sur le codage et le débogage.

Nous analysons maintenant comment optimiser la phase de construction pendant le développement. La phase de construction correspond au temps de construction de l'image lorsque nous modifions le code source Python. L'image doit être reconstruite afin d'obtenir les mises à jour du code Python dans le conteneur avant de le lancer.

Nous pouvons cependant appliquer des changements de code sans avoir à construire l'image. Nous pouvons le faire simplement en fixant le répertoire source local à son chemin dans le conteneur. Pour cela, nous mettons à jour le fichier de composition comme suit:

```
...
  app:
    build: app
    restart: always
    volumes:
```

```
      - ./app/src:/code
```
…
Avec cela, nous avons un accès direct au code mis à jour et nous pouvons donc ignorer la construction de l'image et redémarrer le conteneur pour recharger le processus Python.



De plus, nous pouvons éviter de redémarrer le conteneur si nous exécutons à l'intérieur un processus de rechargement qui surveille les modifications de fichiers et déclenche le redémarrage du processus Python une fois qu'un changement est détecté. Nous devons nous assurer que nous avons monté en liaison le code source dans le fichier Compose comme décrit précédemment.
Dans notre exemple, nous utilisons le framework Flask qui, en mode débogage, exécute un module très pratique appelé **the reloader**. the reloader surveille tous les fichiers de code source et redémarre automatiquement le serveur lorsqu'il détecte qu'un fichier a changé. Pour activer le mode de débogage, il suffit de définir le paramètre de débogage comme

```
#server.py
server.run(debug=True, host='0.0.0.0', port=5000)
```

Si nous vérifions les journaux du conteneur d'application, nous voyons que le serveur flask s'exécute en mode de débogage.

```
$docker-compose logs
Attaching to project_app_1
app_1 | * Serving Flask app "server" (lazy loading)
app_1 | * Environment: production
app_1 | WARNING: This is a development server. Do not use it in a production
deployment.
app_1 | Use a production WSGI server instead.
```

```
app_1 | * Debug mode: on
app_1 | * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
app_1 | * Restarting with stat
app_1 | * Debugger is active!
app_1 | * Debugger PIN: 315-974-099
```

Une fois que nous avons mis à jour le code source et enregistré, nous devrions voir la notification dans les journaux et recharger.

```
$docker-compose logs
Attaching to project_app_1
app_1 | * Serving Flask app "server" (lazy loading)
...
app_1 | * Debugger PIN: 315-974-099
app_1 | * Detected change in '/code/server.py', reloading
app_1 | * Restarting with stat
app_1 | * Debugger is active!
app_1 | * Debugger PIN: 315-974-099
```

Debugging Code

Nous pouvons déboguer le code de deux manières.

La première est la manière ancienne qui consiste à placer des instructions d'impression dans tout le code pour vérifier la valeur d'exécution des objets / variables. L'application de cela aux processus conteneurisés est assez simple et nous pouvons facilement vérifier la sortie avec une commande **docker-compose logs**.

La deuxième approche est la plus sérieuse. Elle consiste à utiliser un débogueur. Lorsque nous avons un processus conteneurisé, nous devons exécuter un débogueur à l'intérieur du conteneur, puis nous connecter à ce débogueur distant pour pouvoir inspecter les données de l'instance.

Nous reprenons comme exemple notre application Flask. Lorsqu'il est exécuté en mode débogage, outre le module **reloader**, il comprend également un débogueur interactif. Supposons que nous mettons à jour le code pour déclencher une exception, le service Flask renverra une réponse détaillée avec l'exception.

Un autre cas intéressant à exercer est le débogage interactif où nous plaçons des points d'arrêt dans le code et effectuons une inspection en direct. Pour cela, nous avons besoin d'un IDE avec Python et un support de débogage à distance. Si nous choisissons de nous fier à Visual Studio Code pour montrer comment déboguer du code Python s'exécutant dans des conteneurs, nous devons procéder comme suit pour nous connecter au débogueur distant directement à partir de VSCode.

Tout d'abord, nous devons mapper localement le port que nous utilisons pour nous connecter au débogueur. Nous pouvons facilement le faire en ajoutant le mappage de port au fichier Compose:

```
...
  app:
    build: app
    restart: always
    volumes:
      - ./app/src:/code
```

```
    ports:
      - 5678:5678
…
```

Ensuite, nous devons importer le module de débogage dans le code source et le faire écouter sur le port que nous avons défini dans le fichier Compose. Nous ne devons pas oublier de l'ajouter également au fichier de dépendances et de reconstruire l'image pour le service d'application pour installer le package de débogage. Pour cet exercice, nous choisissons d'utiliser le package de débogage ptvsd pris en charge par VS Code.

server.py

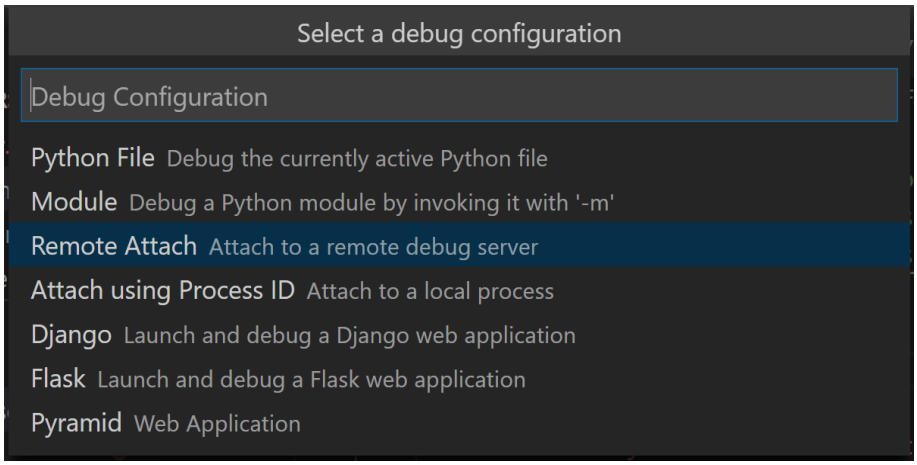
```
...
import ptvsd
ptvsd.enable_attach(address=('0.0.0.0', 5678))
...
```

requirements.txt

```
Flask==1.1.1
mysql-connector==2.2.9
ptvsd==4.3.2
```

Nous devons nous rappeler que pour les modifications que nous apportons au fichier de composition, nous devons exécuter une commande de composition pour supprimer la configuration actuelle des conteneurs, puis exécuter un docker-compose up pour redéployer les nouvelles configurations dans le fichier de composition.

Enfin, nous devons créer une configuration «Remote Attach» dans VS Code pour lancer le mode de débogage.



Le launch.json de notre projet devrait ressembler à ceci:

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Python: Remote Attach",
            "type": "python",
            "request": "attach",
            "port": 5678,
            "host": "localhost",
            "pathMappings": [
```

```
                {
                    "localRoot": "${workspaceFolder}/app/src",
                    "remoteRoot": "/code"
                }
            ]
        }
    ]
}
```

Nous devons nous assurer que nous mettons à jour la carte de chemin localement et dans le conteneur.
Une fois que nous faisons cela, nous pouvons facilement placer des points d'arrêt dans l'EDI, démarrer le mode de débogage en fonction de la configuration que nous avons créée et, enfin, déclencher le code pour atteindre le point d'arrêt.

Resources

- Containerized Python Development [auteur: Anca Iordache]
  https://www.docker.com/blog/author/anca-iordache/
- Best practices for writing Dockerfiles
  - https://docs.docker.com/develop/develop-images/dockerfile_best-practices/
  - https://www.docker.com/blog/speed-up-your-development-flow-with-these-dockerfile-best-practices/
- https://devopssec.fr/
  AJDAINI Hatim
- Docker Desktop
  - https://docs.docker.com/desktop/
- Docker Compose
  - https://docs.docker.com/compose/
- Project skeleton samples
  - https://github.com/docker/awesome-compose