



Formation sur docker et les conteneurs OCI

 www.comwork.io



idriss.neumann@comwork.io



Comwork.io SASU

128 rue de la Boétie 75008 Paris SIRET : 83875798700014

Comwork



Au programme

❖ Les conteneurs

- différences entre VM et conteneur
- OCI open container initiative

❖ Docker

- Les notions de bases (images, layers, networks, volumes)
- Le Dockerfiles
- Builder une image avec docker
- Démarrer un conteneur avec docker
- Créer une application qui repose sur plusieurs conteneurs avec docker-compose
- Optimiser le temps de build en activant buildkit

❖ Les registries publiques et privées

- docker hub
- harbor

❖ Analyser des images

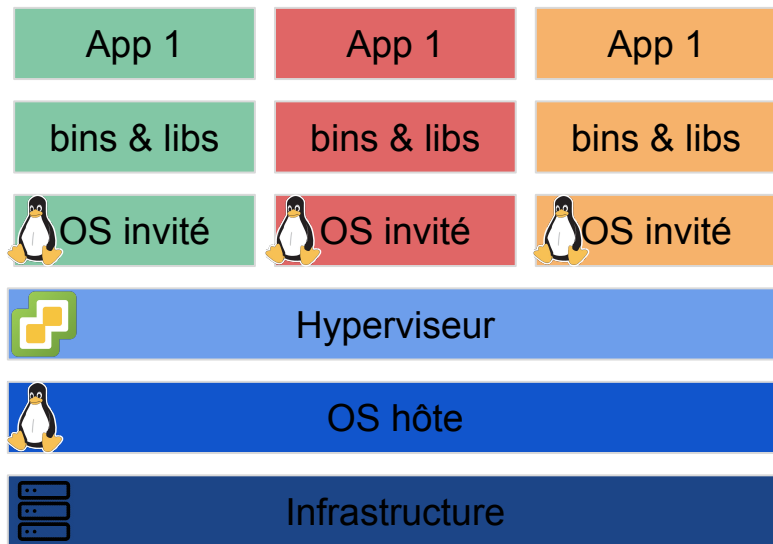
- **diver** pour analyser le contenu des layers
- **trivy** pour analyser les vulnérabilités

❖ Mise en pratique

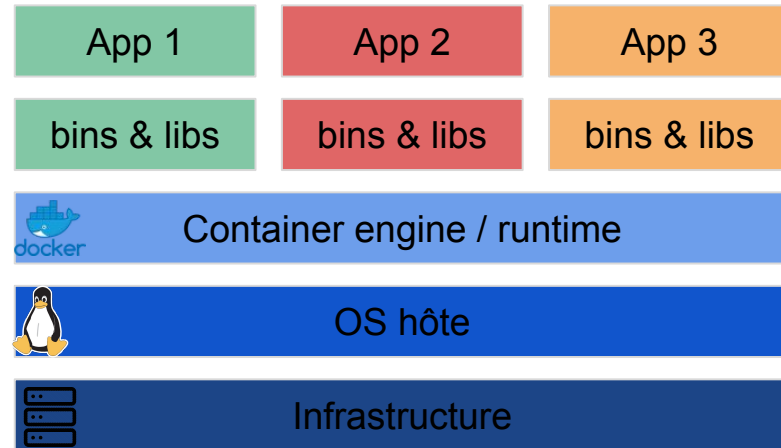
- conteneuriser une API en Python / flask
- conteneuriser une API en Java / Springboot
- conteneuriser une API en PHP / Lumen
- conteneuriser une application front Angular
- automatisation du build avec gitlab-ci

Les conteneurs OCI

Différences entre machines virtuelles et conteneurs



Machines virtuelles



Conteneurs

Les conteneurs OCI

Open container initiative (OCI)

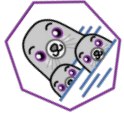
Lancé en 2015 par Docker et d'autres leaders de l'industrie des conteneurs, il s'agit du standard permettant l'interopérabilité des images et conteneurs dits "OCI" : <https://opencontainers.org>

Aujourd'hui, il existe en effet aujourd'hui de nombreux runtimes de conteneurs (ou "containers engine"):

- containerd
- docker
- podman
- cri-o
- etc

Et différentes façon de construire des images:

- docker
- buildkit
- buildah
- kaniko
- etc



podman



cri-o



buildah



docker

OCI est le garant de l'interopérabilité des Dockerfiles et des images buildées d'une plateforme à l'autre pour être runnées d'une plateforme à l'autre.

Docker

Les notions de bases

- ❖ Les images
 - identifiées par un nom et un tag
 - constituent un ensemble cohérent de “layers” (couches) pour démarrer un service atomique
 - peuvent hériter d'autres images ou ré-importer des layers d'autres images
 - les images de bases sont généralement des images publiques disponibles sur docker hub (ou autre registry publique)

```
lineumann on master * ~/docker $ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
harbor.comwork.io/todoapi/unittest	latest	54d886278da6	13 days ago	917MB
harbor.comwork.io/todoapi/api	latest	9f451edeb0d7	13 days ago	916MB
harbor.comwork.io/clockify/clockify	main-3f7c0792	74a24034adc5	2 weeks ago	51.6MB
harbor.comwork.io/clockify/clockify	main-9af2e6fd	448bdd0a4e97	2 weeks ago	10.5MB
harbor.comwork.io/clockify/clockify	latest	53a61c087072	3 weeks ago	10.9MB
harbor.comwork.io/mutual/mutual_ui	latest	8668758e129a	8 weeks ago	63.7MB
harbor.comwork.io/mutual/mutual_ws	latest	d1bc87d46858	8 weeks ago	305MB
docker.elastic.co/kibana/kibana	7.13.2	0d7fac58828d	8 weeks ago	1.51GB
harbor.comwork.io/hub/esf_test	latest	63d5447f3e60	2 months ago	880MB
tomcat	9	d9e19311a36b	2 months ago	658MB
openjdk	8	6b24bd100507	2 months ago	509MB
mutual_db	latest	ef5df7c9d93f	2 months ago	300MB
comworkio/cmd-api	latest	232195981de6	2 months ago	120MB
docker.elastic.co/elasticsearch/elasticsearch	7.6.1	41072cdeebc5	17 months ago	790MB
postgres	9.4	897a27671908	17 months ago	241MB

dockerhub Explore Pricing Sign In [Sign Up](#)

Explore library/centos 7

centos:7
DIGEST: sha256:e4ca2ed020e76be184e75fb26d14bf974193579039d5573fb2348664deef76e

OS/ARCH: linux/amd64 COMPRESSED SIZE: 72.57 MB LAST PUSHED: 6 months ago by dojanky

IMAGE LAYERS ⓘ

1	ADD file ... in /	72.57 MB	Command
2	LABEL org.label-schema.schema-version=1.0 org.label...	0 B	ADD file:7f21ae7d20a8e347d8b678bcf26be83abb1ee27d3b567c9cddd993e45ce8ac34 in /
3	CMD ["/bin/bash"]	0 B	

Docker

Les notions de bases



Les layers

- artifact OCI qui sont des sortes d'archives tar qui composent les images docker et correspondent au produit d'une instruction dans le Dockerfile
- les layers sont identifiés par un sha et peuvent être mutualisés pour différentes images (via l'héritage ou le multistage build avec **copy --from**)
- les layers permettent d'éviter d'avoir à être rebuildé ou retéléchargés sur l'hôte s'ils ne changent pas

```
ineumann on master * ~/docker $ docker inspect harbor.comwork.io/todoapi/unittest | jq .[0].RootFS
{
  "Type": "layers",
  "Layers": [
    "sha256:0c4db5d7ee48e8d916e6d1f6f6f77c8dbca383eb80ab74fa85b6911767523219",
    "sha256:b48bc43bef8b688ca2c18f93a382b4f3c362de5b7061d4c6048f02018b75c59",
    "sha256:665bd204ab72b3539803767ed3b49b63ea337a425c496fe1bd5cdeb782b9f7b8d",
    "sha256:4859da74ce517234d0198363f169e43c7d8f005d4f2729b4300b4823d1d8c6d9",
    "sha256:410ec4a217374a1cea90fa0d91756571e0d6c380186b974b4c22bda54d0716f",
    "sha256:f876c0b805f9da088e2613342822b90fafc891c5b27e5f62432fa4f0035ac5da",
    "sha256:556a8c5d4e82079e79b80d5f1abdbdb44c093021415c799d9bbc60a4f3ab7f7",
    "sha256:9a9341f9cdf4e2f99768c0a1517eaa3416ae128acea5875d1642bcc9efafc7cb",
    "sha256:bba7cdea55f941b1bc7ad681d2c4b577807e70ddb733a82b414f385f20c7c976",
    "sha256:679568fa6490a2c09068a9410ad252549ecc7ed112df0b39d9b65ae689c394ca",
    "sha256:5f70bf18a08607016e948b04aed3b82103a36bea41755b6cddfaf10ace3c6ef",
    "sha256:f70ffdd13c9eead068cba7d9b9fcdff551d202eedc8a2f99a493fd7bc65d8541",
    "sha256:5f70bf18a08607016e948b04aed3b82103a36bea41755b6cddfaf10ace3c6ef",
    "sha256:ab1a79ebfa310d77ae0f8176791b945c4e886e993d600702fb46f6eb9da8e26ee"
  ]
}
```

```
ineumann on master * ~/docker $ docker pull docker.elastic.co/elasticsearch/elasticsearch:7.11.0
7.11.0: Pulling from elasticsearch/elasticsearch
0122c235edee: Already exists
9bc50b2741f6: Downloading [=====] 15.29MB/24.5MB
4697480b6de2: Download complete
add2fd0c5df: Downloading [=] 10.77MB/346.5MB
36f20916e73d: Download complete
2fd6f9204a99: Download complete
cb1cc36d3a3f: Download complete
```

Docker

Les notions de bases



Les networks

- permettent en fonction du driver de faire communiquer les conteneurs entre eux sur des réseaux virtuels (bridge) ou partager l'interface réseau de l'hôte (host) ou bien d'être complètement étanche (none)

```
ineumann on master * ~/docker $ docker network ls
```

NETWORK ID	NAME	DRIVER	SCOPE
def4764dba20	bridge	bridge	local
517952c7841c	host	host	local
11d5c2942d3e	none	null	local
edb0c2b6b5a9	pipeline_api_default	bridge	local
9644f1818d40	talend-docker-v2_default	bridge	local
135c0393ba81	todoapi_default	bridge	local
1a045c7435dc	todoapi_todo_api	bridge	local

```
ineumann ~ $ docker network inspect pipeline_api_default
[
  {
    "Name": "pipeline_api_default",
    "Id": "edb0c2b6b5a96401338384619b6745732e609828db1a9d3cb311c95a3da5fa35",
    "Created": "2021-07-30T11:15:57.853188176Z",
    "Scope": "local",
    "Driver": "bridge",
    "EnableIPv6": false,
    "IPAM": {
      "Driver": "default",
      "Options": null,
      "Config": [
        {
          "Subnet": "172.20.0.0/16",
          "Gateway": "172.20.0.1"
        }
      ]
    },
    "Internal": false,
    "Attachable": true,
    "Ingress": false,
    "ConfigFrom": {
      "Network": ""
    },
    "ConfigOnly": false,
    "Containers": {},
    "Options": {},
    "Labels": {
      "com.docker.compose.network": "default",
      "com.docker.compose.project": "pipeline_api",
      "com.docker.compose.version": "1.29.2"
    }
  }
]
ineumann ~ $ docker network inspect host
[
  {
    "Name": "host",
    "Id": "517952c7841c6a31ddf27577de9107167149de64f80537ed85968bf6b6ac3566",
    "Created": "2021-06-04T09:28:37.435149835Z",
    "Scope": "local",
    "Driver": "host",
    "EnableIPv6": false,
    "IPAM": {
      "Driver": "default",
      "Options": null,
      "Config": []
    },
    "Internal": false,
    "Attachable": false,
    "Ingress": false,
    "ConfigFrom": {
      "Network": ""
    },
    "ConfigOnly": false,
    "Containers": {},
    "Options": {},
    "Labels": {}
  }
]
```

Docker

Les notions de bases



Les volumes

- permettent d'assurer la persistance des données d'un conteneur même dans le cas où il est amené à être détruit et reconstruit
- permet de monter un répertoire ou fichier de l'hôte à l'intérieur d'un conteneur (ainsi il est par exemple possible d'utiliser des conteneurs génériques comme un openjdk par exemple pour démarrer n'importe quel fichier jar qu'on aurait construit sur l'hôte local)

```
ineumann on master * ~/docker $ docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED        STATUS        PORTS                               NAMES
ad3aa3fb2368   postgres:9.4  "docker-entrypoint.s..."  4 days ago    Up 4 days    0.0.0.0:5436->5432/tcp, :::5436->5432/tcp  todo_db

ineumann on master * ~/docker $ docker inspect todo_db|jq .[0].Mounts
[
  {
    "Type": "bind",
    "Source": "/Users/ineumann/docker/todoapi/install.sql",
    "Destination": "/install.sql",
    "Mode": "ro",
    "RW": false,
    "Propagation": "rprivate"
  },
  {
    "Type": "bind",
    "Source": "/Users/ineumann/docker/todoapi/data_volume",
    "Destination": "/var/lib/postgresql/data",
    "Mode": "z",
    "RW": true,
    "Propagation": "rprivate"
  }
]
```


Docker

Le Dockerfile

```
ARG MAVEN_VERSION=3.3-jdk-8
ARG TOMCAT_VERSION=9.0-jre8-slim
ARG OPENJDK_VERSION=8-jre-slim
```

Arguments de build (que l'on peut surcharger mais qui ont des valeurs par défauts qu'on pourra ré-utiliser comme variable au moment du build)

```
#####
# Stage spring_service_build: build maven for spring base app #
#####
```

```
FROM maven:${MAVEN_VERSION} AS spring_service_build
```

Build multistage (images temporaires dont on va se servir pour exporter certains layers résultant du build de ces stages)

```
ARG IMAGE_SERVICE
ENV BASE_DIR="/uprodit"
ARG MVN_ARTIFACT_VERSION=0.0.1-SNAPSHOT
```

Image de base (accessible depuis dockerhub) sur laquelle on va hériter cette image de stage

```
WORKDIR ${BASE_DIR}
```

```
COPY . ${BASE_DIR}
```

```
COPY ./prodit-batch/src/main/resources/env/docker/tomcat/log4j2.xml /
```

```
RUN mv /log4j2.xml /uprodit/prodit-${IMAGE_SERVICE}/src/main/resources && \
    mvn clean install -Dmaven.test.skip -P ${IMAGE_SERVICE} && \
    mv /uprodit/prodit-${IMAGE_SERVICE}/target/prodit-${IMAGE_SERVICE}-${MVN_ARTIFACT_VERSION}.war /ROOT.war && \
    mv manifest.json /manifest.json
```

Docker

Le Dockerfile

```
#####
# Stage vertx_service_build: build maven for vert.x base app #
#####
```

```
FROM maven:${MAVEN_VERSION} AS vertx_service_build
```

```
ARG IMAGE_SERVICE
```

```
ENV BASE_DIR="/uprodit"
```

```
ARG MVN_ARTIFACT_VERSION=0.0.1-SNAPSHOT
```

```
WORKDIR ${BASE_DIR}
```

```
COPY . ${BASE_DIR}
```

```
COPY ./prodit-batch/src/main/resources/env/docker/vertx/log4j2.xml /
```

```
RUN mv /log4j2.xml /uprodit/prodit-${IMAGE_SERVICE}/src/main/resources && \
    mvn clean install -Dmaven.test.skip -P ${IMAGE_SERVICE} && \
    mv /uprodit/prodit-${IMAGE_SERVICE}/target/prodit-${IMAGE_SERVICE}-${MVN_ARTIFACT_VERSION}-fat.jar /vertx-fat.jar && \
    mv /uprodit/prodit-${IMAGE_SERVICE}/src/main/resources/config.json /config.json && \
    mv manifest.json /manifest.json
```

Commandes qui sont effectuées au moment du build de l'image (et non pas du runtime du conteneur). Il faut éviter de répéter les instructions RUN et les condenser en une seule pour optimiser le nombre de layers immutables

Fixer le répertoire dans lequel on va effectuer les commandes de build (RUN)

copie de fichier ou répertoires comme layers à l'intérieur de l'image au moment du build (à pas confondre avec un volume non immutable monté au runtime)

Docker

Le Dockerfile

```
#####  
# Stage spring_service: exposing a spring based app #  
#####
```

```
FROM tomcat:${TOMCAT_VERSION} AS spring_service
```

```
COPY ./prodit-batch/src/main/resources/env/docker/tomcat/server.xml /usr/local/tomcat/conf/server.xml:z  
COPY ./prodit-batch/src/main/resources/env/docker/tomcat/catalina.sh /usr/local/tomcat/bin/catalina.sh:z  
COPY ./prodit-batch/src/main/resources/env/docker/tomcat/logging.properties  
/usr/local/tomcat/conf/logging.properties:z
```

```
RUN mkdir -p /prodit/prodit_cache /BACK_PRODIT && \  
rm -rf /usr/local/tomcat/webapps/ROOT && \  
rm -rf /usr/local/tomcat/webapps/manager && \  
rm -rf /usr/local/tomcat/webapps/examples && \  
rm -rf /usr/local/tomcat/webapps/host-manager && \  
rm -rf /usr/local/tomcat/webapps/docs
```

```
COPY --from=spring_service_build /ROOT.war /usr/local/tomcat/webapps/ROOT.war  
COPY --from=spring_service_build /manifest.json /manifest.json
```

```
EXPOSE 8080
```

```
ENTRYPOINT ["catalina.sh", "run"]
```

copie de fichiers en provenance de layers construit par les stages précédentes
(from={nom de la stage})

Docker

Le Dockerfile

```
#####  
# Stage vertx_service: exposing a vertx based app #  
#####  
  
FROM openjdk:${OPENJDK_VERSION} AS vertx_service  
  
ENV VERTX_PORT=80  
  
COPY --from=vertx_service_build /vertx-fat.jar /config.json /manifest.json /  
  
EXPOSE 80  
  
CMD ["java", "-jar", "/vertx-fat.jar", "-conf", "/config.json"]
```

→ Port qui sera exposé à l'instanciation du conteneur sur le réseau docker

→ Commande qui sera exécutée au runtime du conteneur

Mise en pratique

Bonnes pratiques dans l'écriture de Dockerfile pour différentes technologies

La suite se passe sur ce repo git (à cloner et suivre le README.md): https://gitlab.comwork.io/comwork_training/docker

