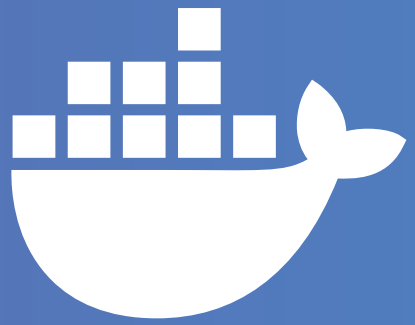
TOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE



Docker

Conteneurs d'applications

Les conteneurs sont une technologie clé dans le paysage du cloud natif. Ils sont utilisés pour emballer et **isoler** les applications avec leurs **dépendances** entières, y compris le système d'exploitation, les bibliothèques système, les scripts, etc.

Cela permet d'assurer que l'application fonctionne de manière cohérente et fiable **dans n'importe quel environnement**, que ce soit en développement, en test ou en production.

Conteneurs d'applications

Un conteneur est **plus léger** qu'une machine virtuelle traditionnelle car il partage le système d'exploitation de l'hôte et n'a pas besoin de son propre système d'exploitation.

Cela rend les conteneurs très efficaces en termes d'utilisation des ressources système et de temps de démarrage.

Conteneurs d'applications

Évolutivité : Les conteneurs peuvent être démarrés et arrêtés rapidement, ce qui facilite leur mise à l'échelle en fonction de la demande. Si la demande pour une application augmente, plus de conteneurs peuvent être lancés pour gérer cette demande.

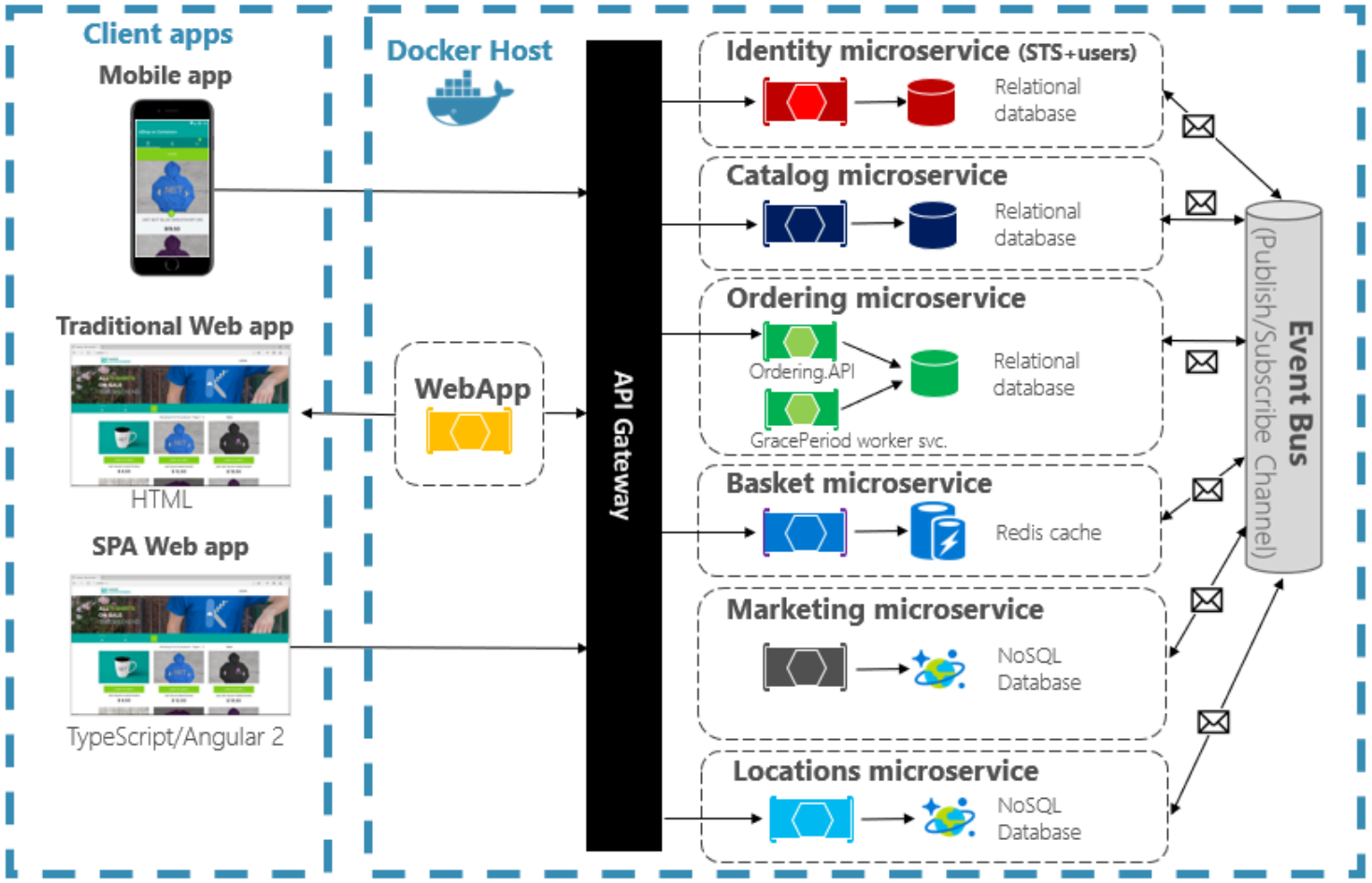
Déploiement continu et livraison continue (CI/CD) : Les conteneurs sont parfaits pour les pipelines CI/CD car ils permettent de déployer rapidement de nouvelles versions d'une application.

Conteneurs d'applications

Isolation : Chaque conteneur fonctionne de manière isolée. Cela signifie qu'un conteneur n'a pas d'effet sur les autres conteneurs et peut avoir ses propres configurations système et logicielle.

Portabilité : Les conteneurs garantissent que l'application fonctionne de manière identique dans tous les environnements. Cela permet de déplacer facilement les applications d'un système à un autre, ou d'un cloud à un autre.

Conteneurs d'applications



Docker

Installation de Docker Desktop

Lorsque l'on veut travailler avec Docker, il nous faut dans un premier temps l'avoir installé. Sur Windows, il nous faudra installer Docker Desktop, que l'on peut obtenir sur le site officiel

Les bases de Docker

Une fois ceci fait, il nous est possible d'utiliser des commandes de base pour interagir avec Docker Desktop / Docker Engine. Par exemple, pour lancer un conteneur basé sur une image, on a la syntaxe `docker run nom-image`:

```
# Pour lancer un conteneur basé sur Node.js
docker run node
```

Les bases de Docker

Pour observer nos conteneurs, il nous est possible d'utiliser la commande `docker ps`:

```
# Pour afficher les conteneurs en cours d'exécution
docker ps
# Pour afficher également ceux à l'arret
docker ps -a
```

Il est parfois nécessaire d'utiliser les options `-i` et `-t` lorsque l'on lance un conteneur, dans le but de pouvoir interagir avec en usant du système de base de connexion Linux :

```
docker run -it node
```

Le Dockerfile

Dans le cas où nous souhaitons dockeriser nos applications, il nous faudra la plupart du temps avoir recours à un type de fichier particulier: le **Dockerfile**. Ce fichier va servir à Docker de script pour effectuer la création d'une image pouvant être mise dans un conteneur par la suite.

Le Dockerfile

Il est important, lorsque l'on réalise un Dockerfile, de respecter une syntaxe particulière. Par exemple, dans le cadre d'un application Node.js, on pourrait avoir un Dockerfile de ce type:

```
FROM node
WORKDIR /app
COPY . .
RUN npm install
EXPOSE 80
CMD [ "node", "server.js" ]
```

Le Dockerfile

- **FROM**: Sert à définir l'image de base à partir de laquelle nous créons notre propre image
- **WORKDIR**: Sert à définir un dossier (qui sera créé) à partir duquel les commandes suivantes seront exécutées (il devient le dossier courant)
- **COPY**: Sert à copier des fichiers dans l'image

Le Dockerfile

- **RUN**: Sert à exécuter des commandes durant la création de l'image
- **EXPOSE**: Sert à avertir que des ports seront utilisés par l'image
- **CMD**: Sert à définir la commande exécutée par le conteneur à son lancement

Pour pouvoir créer un conteneur à partir de notre fichier Dockerfile, il nous faut utiliser la commande `docker build chemin-vers-fichier`:

```
# Si le fichier Dockerfile est à l'emplacement du terminal:
docker build .
```

Le Dockerfile

Il ne nous reste plus qu'à lancer notre conteneur basé sur l'image générée. Dans le cas où le processus est bloqué (car en écoute), il sera possible de le terminer via l'ouverture d'un autre terminal et de l'utilisation de la commande `docker stop id-ou-nom-conteneur`:

```
docker ps
# On regarde le nom du conteneur
docker stop nom-conteneur
# Stoppera le conteeur
```

Le Dockerfile

Dans notre exemple, il nous faudra ajouter la capacité de notre conteneur à **rediriger les ports** du conteneur vers un port de l'hôte, ce dans le but de pouvoir atteindre l'application interne au conteneur. Pour réussir à faire cela, il nous faut publier les ports, via l'option `-p port-hote:port-conteneur` dans le cas d'une commande de lancement de conteneur :

```
docker run -p 80:80 id-image
```

Les modifications du code

Dans le cas où l'on souhaite par la suite changer le code de notre application, il est crucial de ne pas oublier qu'une image est créée à un instant T et contient le code qui était présent à ce moment. Il n'est donc pas possible de voir les changements dans un conteneur du temps qu'une nouvelle image n'a pas été créée et qu'un conteneur usant de cette image n'est pas lancé sur notre machine.

Les modifications du code

Il est également intéressant de connaitre le fonctionnement de Docker lors de la création d'images. Ces images sont basées sur des **couches**, qui peuvent se servir d'un **cache** dans le but d'accelérer les créations ultérieures d'images.

En séparant les instructions de la création d'image, il est par exemple possible d'**optimiser** la création de notre image de Node.js de sorte à ne recréer que les sections concernées.

Les modifications du code

On va ainsi faire en sorte d'exclure la partie de génération des dépendances de la copie de nos fichiers de code :

```
FROM node
WORKDIR /app
COPY package.json .
RUN npm install
COPY . .
EXPOSE 80
CMD [ "node", "server.js" ]
```

Gérer nos conteneurs et nos images

Si l'on le veut, il est possible de **relancer** un conteneur actuellement à l'arrêt via la commande `docker start id-ou-nom-conteneur`. Via cette commande, le conteneur est lancé de façon détachée, en arrière-plan.

Gérer nos conteneurs et nos images

Pour obtenir un **mode détaché** dans le cas de `docker run`, il va falloir utiliser l'option `-d`. De même, pour obtenir un mode attaché lors de l'utilisation de `docker start`, on peut utiliser l'option `-a`. Enfin, il est possible de s'attacher à un conteneur lancé de façon détachée via l'utilisation de `docker attach nom-id-conteneur`.

Gérer nos conteneurs et nos images

Logs : si notre conteneur est du genre à envoyer des instructions à la sortie standard de notre console, il peut également être utile d'utiliser la commande `docker logs nom-id-conteneur` (à laquelle on peut ajouter `-f` pour suivre les évolutions futures) pour en afficher le contenu dans notre terminal.

Gérer nos conteneurs et nos images

Le **mode interactif** d'un conteneur, atteignable via l'utilisation de l'option `-i`, sert à pouvoir, si l'on en a le besoin, entrer des informations dans le conteneur. Par exemple, si l'on souhaite lancer des commandes bash, ou si le conteneur nécessite des entrées utilisateur dans son fonctionnement.

Gérer nos conteneurs et nos images

Prenons pour exemple un script **Python** dont l'objectif est le calcul d'une somme de deux nombres. Pour ce faire, le script doit demander en console deux entrées successives puis afficher en sortie la somme. Ceci n'est possible au sein d'un conteneur que si l'utilisateur est capable d'entrer les données. Dans le cas contraire, le conteneur est inutilisable et sera bloqué.

Gérer nos conteneurs et nos images

Pour obtenir le résultat escompté, il nous faudra utiliser `-i` pour le mode **interactif** (on permet la lecture de **STDIN**) et `-t` pour obtenir un pseudo-TTY (un **terminal** exposé par le terminal dans lequel on peut interagir).

```
docker run -it nom-image
```

Supprimer un conteneur ou une image

Si l'on se rend compte que l'on a beaucoup trop d'images ou de conteneurs dans notre système, il est possible de les supprimer. Pour supprimer manuellement les conteneurs, il est d'abord nécessaire de les stopper via la commande `docker stop`. Une fois cela fait, il est possible de supprimer un à un les conteneurs via la commande `docker rm nom-id-conteneur...` ou de supprimer d'un coup tous les conteneurs à l'arrêt via `docker container prune`.

Supprimer un conteneur ou une image

Dans le cas des images, il est possible de réaliser une suppression sélective via `docker rmi nom-id-image...`. Pour supprimer toutes les images ne portant pas de tag, il est possible de faire `docker image prune` (l'option `-a` supprimera également celles portant des tags mais n'étant pas utilisées par des conteneurs).

Si l'on le veut, lors du lancement d'un conteneur, il est possible d'utiliser l'option `--rm` pour **supprimer automatiquement** le conteneur une fois son exécution terminée.

Supprimer un conteneur ou une image

Si l'on souhaite en savoir plus sur nos images, il est possible d'utiliser la commande `docker image inspect nom-id-image` pour se voir offrir un JSON de détails que l'on peut ainsi lire.
Pour pouvoir éditer les fichiers d'un conteneur, il est possible d'utiliser la commande `docker cp`:

```
# Vers le conteneur
docker cp mon-fichier.txt mon-conteneur:/tmp/mon-fichier.txt
# Depuis le conteneur
docker cp mon-conteneur:/tmp/mon-fichier.txt mon-fichier.txt
```

Noms et tags

Plutôt que de devoir travailler avec des ID géants pour les noms d'images, ou des noms générés aléatoirement pour les conteneurs, il est possible de tagger nos images à la création via l'option `-t nom-image:tag`, et de nommer nos conteneurs via l'option `--name nom-conteneur`.

Noms et tags

Le tagging d'une image permet de créer des sous-versions de nos images, par exemple une version plus légère, ou plus récente. Par défaut, lors de l'utilisation d'une image, on utilisera le tag `latest`.

Ainsi, pour notre Dockerfile d'image Node.js, il est possible d'utiliser l'image `node:alpine` pour obtenir une image de notre application plus légère.

Dockerhub

Dans l'univers Docker, il est courant de vouloir soit partager, soit sauvegarder nos images dans le cloud. A la manière de Git, il est possible de le faire via l'utilisation de Dockerhub.
Ce service offre de façon gratuite la capacité de stocker en ligne nos images, qui seront publiques (il est possible d'avoir des images privées en cas d'utilisation de la formule d'abonnement). Les images dont nous nous servons comme base depuis le début sont localisées sur ce service en ligne.

Dockerhub

Pour nous servir de Dockerhub, il va nous falloir créer un compte, puis configurer Docker pour y lier notre compte. La commande à utiliser se nomme `docker login`. Une fois connecté, il est possible de nous servir de `docker pull` pour récupérer des images, ainsi que de `docker push` pour envoyer des images dans le registre de conteneur sélectionné par notre configuration de docker.

```
docker pull repository/image-name:tagname
docker push repository/image-name:tagname
```

Dockerhub

Pour pouvoir envoyer ou recevoir d'un registre privé, il nous faudra ajouter le nom de l'hôte avant le nom de l'image:

```
docker pull hote:repository/image-name:tagname
docker push hote:repository/image-name:tagname
```

Dockerhub

Pour obtenir une image portant le bon nom, il peut être nécessaire d'utiliser la commande `docker tag` pour créer un duplicat de notre image de base, dans le but de lui offrir le nom convenant à la syntaxe demandée par le push sur un registre de conteneur:

```
docker tag nom-image:tag nom-registre/nom-image:tag
```

Merci pour votre attention

Des questions ?