SAÉ 1.03

Installation d'un Poste de Développement

Partie 1 - Présentation et premiers pas

Introduction à la SAÉ 1.03

Cette SAÉ s'intitule *Installation d'un Poste de Développement*. Sa finalité n'est pas aussi évidente qu'il y paraît. Ce n'est pas simplement l'installation d'un VSC et d'un GCC.

Le développement est un sujet assez large. Il peut s'agir de programmation Système, de développement d'applications sur Desktop, sur Mobile, de développement Web, de travaux en BDD, de programmation d'équipements électroniques, de beaucoup d'autres choses et parfois même de plusieurs d'entre elles combinées ensemble.

Stricto sensu, développer c'est programmer. Mais durant la phase de développement, un·e développeu·r·se fait souvent d'autres tâches :

- Documentation (Doxygen, Javadoc etc.)
- Gestion de code source (Git, Svn, etc.)
- Conversions de fichiers (CSV -> Json, images PNG -> JPG pour le Web, etc.)
- etc.

Tout ceci nécessite l'installation de logiciels, d'outils, de librairies, de services etc. sur le poste de développement.

En tant que futur·e·s professionnel·le·s du développement, vous devez savoir comment installer et configurer votre environnement de travail, votre poste de développement. Point important : vous devez apprendre aussi comment le faire proprement, d'une façon compatible avec le travail en équipe, et d'une façon reproductible facilement et rapidement.

Même dans les grosses structures où il y a des équipes d'installation et de maintenance matérielle et logicielle, le développeur a généralement un certain degré de liberté pour la configuration de son poste.

C'est le sujet de cette SAÉ.

Que vous travailliez en solo ou en équipe, installer un poste de développement n'est pas compliqué en soi, mais c'est souvent une phase un peu laborieuse et rarement très intéressante. On a toujours peur d'oublier quelque chose, on remet à plus tard tel ou tel

point en se disant qu'on installera ça quand on en aura besoin. Et, comme nous le dit très sagement Murphy[1], le besoin arrive toujours au mauvais moment.

En équipe, il peut aussi arriver qu'on n'ait pas tous exactement les mêmes versions de certains outils, de librairies installées sur nos machines. Parfois on rencontre aussi, sur sa propre machine, des incompatibilités entre les logiciels, les outils, les librairies qu'on installe. On ajoute un truc et ça casse des choses qui fonctionnaient bien avant. Et quand on décide de retirer le truc qui a tout cassé, c'est encore pire !

On a bien d'autres choses plus intéressantes à faire que perdre du temps à résoudre ces problèmes. Nous allons donc voir ensemble comment on prépare son poste de développement et des techniques pour se simplifier la vie.

Quelques rappels du cours

Virtualisation

Vous connaissez sans doute la *virtualisation* avec Virtualbox (gratuit) ou peut-être avec d'autres solutions telles que VMWare[2] ou encore Parallels (toutes deux payantes).

Même si la technologie n'est pas récente, elle était réservée aux gros systèmes professionnels très coûteux. Ces solutions de virtualisation se sont démocratisées dans les années 2000 avec l'arrivée de Linux dans un monde où Windows était omniprésent (95% des PC en 1998[3]).

Une *Machine Virtuelle*, qu'on appelle généralement une VM[4], est un ordinateur logique[5] émulé dans un ordinateur physique, qu'on appelle un hôte.

Certains d'entre vous ont peut-être déjà eu l'occasion d'installer une VM pour travailler sous Linux[6] chez eux dans certaines matières : Systèmes d'Exploitation, Algo/Prog, BDD, Web et d'autres à venir.

On pourrait se demander pourquoi émuler logiciellement quelque chose dont on dispose déjà physiquement. Il existe plusieurs raisons de le faire. Sans être exhaustif :

- On peut faire fonctionner plusieurs VM sur un même hôte et ça revient moins cher d'acheter un plus gros ordinateur que plusieurs petites machines. Généralement un gros ordinateur est aussi de meilleure qualité, de meilleure conception et a finalement moins de risques de tomber en panne qu'un petit. Et si on a plusieurs petites machines, on multiplie alors les risques de panne sur le long terme.

[1] La *Loi de Murphy* est aussi connue sous le nom de *l'emmerdement maximum*
[2] Sous macOS elle s'appelle Fusion.
[3] 75% en 2021.
[4] **V**irtual **M**achine.
[5] *Logique* signifie qu'il est virtuel, qu'il existe logiciellement mais pas matériellement.
[6] https://youtu.be/A6sVbOuRys8

- On peut parfois avoir des besoins ponctuels et si on dédie une machine à ce besoin ponctuel, elle est monopolisée pour ce besoin. Une VM ne monopolise pas un ordinateur. Quand on arrête une VM, elle ne consomme et ne coûte plus rien si ce n'est un peu d'espace sur un disque dur[7].
- On peut faire fonctionner des systèmes différents dans chaque VM : Windows, Linux et d'autres Unix, dans différentes versions, le tout disponible et accessible en même temps, en parallèle. Ceux d'entre vous qui ont installé un Linux dans une VM chez eux savent bien pourquoi ils ont fait ainsi et le confort qu'ils retirent d'une telle solution : on a un Windows et un Linux sur la même machine, fonctionnant en même temps. On travaille le cours de Systèmes (si, si) dans la VM Linux en même temps qu'on garde son Windows ouvert pour tout plein d'autres usages.
- On peut cloisonner des applications sur chaque VM et ainsi empêcher qu'elles ne communiquent entre elles ou qu'elles ne compromettent la sécurité des autres applications, ce qui est compliqué voire impossible si toutes ces applications tournent sur un même ordinateur physique. A l'ère du tout Internet, c'est une caractéristique très importante pour les entreprises et leur sécurité.
- Dans une VM, on peut avoir des droits différents (et notamment plus élevés) que ceux qu'on possède dans l'hôte, sans impacter la sécurité de l'hôte.

Conteneurisation

La technologie dite de *conteneurisation* ressemble beaucoup à la *virtualisation*, mais elle diffère d'elle sur certains aspects. C'est une technologie cousine.

On parle de VMs pour la virtualisation, on parle de conteneurs pour la conteneurisation.

Un air de famille

Comme avec les cousins, il y a toujours un air de famille, et voici ce qui les rapproche :

- Un conteneur est quelque chose de logiciel et non pas matériel.
- Un conteneur a besoin d'un hôte physique pour exister, pour vivre.
- On peut faire fonctionner simultanément plusieurs conteneurs sur un même hôte.
- Comme son nom l'indique, un conteneur permet de cloisonner et d'isoler son contenu.
- Un conteneur arrêté ne consomme rien et ne coûte rien. On peut donc en créer beaucoup, démarrer ceux dont on souhaite disposer à un moment donné, et arrêter ceux qui ne servent à rien.
- Un conteneur peut partager des fichiers et des dossiers avec son hôte ou avec d'autres conteneurs (très pratique pour les échanges).
- Un conteneur permet d'avoir un environnement de travail facilement reproductible d'un hôte à un autre.

[7] Un disque dur est une ressource très économique aujourd'hui.

Des différences

- Un conteneur n'a pas de Système d'Exploitation (de Noyau[8]), il utilise le Système de l'hôte. On ne peut donc pas mixer différents types de conteneurs (ex : Windows avec Linux) sur une même machine physique. Cela signifie aussi qu'il n'y a pas la phase de démarrage de l'OS qu'on aurait avec une VM. Démarrer un conteneur est extrêmement rapide.
- Un conteneur est simplement un processus, du point de vue du système hôte.
- La conteneurisation s'adresse uniquement à des outils en mode ligne de commande[9]. Dans ce cas, on parlera de mode **CLI** pour **C**ommand **L**ine **I**nterface.

On aura l'occasion de voir d'autres différences fondamentales entre conteneurs et VMs plus tard, l'année prochaine.

Résumons

Docker **n'est pas** une technologie de *virtualisation*.

Docker **est** une technologie de *conteneurisation*.

Docker est **très présent** dans le domaine du **développement logiciel**, en entreprise.

Docker fonctionne nativement et **exclusivement sous Linux**.

Docker fonctionne nativement sous Linux, mais...

... mais Docker peut aussi être utilisé sous Windows[10] et macOS.

Ça semble être en totale contradiction avec ce qui vient d'être dit précédemment : Linux est l'OS hôte exclusif de Docker !

L'explication est simple et sans doute surprenante : sous Windows et macOS, Docker est installé et s'exécute dans... une VM Linux !

Quel intérêt ? Si Docker trouve sa place comme solution alternative à une VM mais qu'il faut une VM pour faire tourner Docker, ça semble effectivement contre productif.

La réponse est simplement qu'il ne faut pas mettre Docker et les VMs en compétition. Ils ont chacun leurs avantages et leurs inconvénients et peuvent fonctionner ensemble. C'est un peu comme mettre le train et la voiture en concurrence, mais on met bien des voitures sur des trains dans certaines circonstances (ex : l'Eurostar entre la France et l'Angleterre transporte aussi des voitures). On transporte aussi des choses sur des bateaux de façon similaire, d'ailleurs on appelle cela des... conteneurs évidemment !

[8] Kernel
[9] Il existe quelques très rares applications graphiques conteneurisées, mais assez compliquées à mettre en œuvre. Ça sort du cadre de ce cours.
[10] macOS est un Unix mais Windows est à 1000 lieues d'un système Unix.

Faire fonctionner Docker dans une VM Linux est un passage incontournable pour les OS autres que Linux, comme pour passer le Tunnel sous la Manche en voiture.

Ça permet de pouvoir travailler en équipe quand les membres ne sont pas tous sur des ordinateurs Linux. Mais ça reste dédié à la phase de développement. En production, les conteneurs seront toujours installés/exécutés sur des hôtes sous Linux.

Conclusion

Nous avons vu quelques différences notables entre conteneurs et VMs. Ce n'est pas grave si vous ne comprenez pas encore toutes les subtilités et toute la puissance de la conteneurisation.

Par contre, nous allons voir ensemble comment utiliser Docker et vous devez bien assimiler les commandes que nous allons rencontrer, la **syntaxe**, les **options**, etc.

Il est donc important de continuer de **prendre des notes**, mais vous êtes aguerris à cet exercice maintenant.

Ces technologies de conteneurisation sont très prisées depuis quelques années. Le monde de l'hébergement Cloud fonctionne beaucoup de cette manière et les entreprises utilisent beaucoup le Cloud, CQFD.

Nous allons découvrir pas à pas comment vous pourrez en tirer avantage dans vos études et votre future vie professionnelle.

Je vérifie que j'ai compris...

Cochez ce qui est vrai pour Docker :

[__] c'est une machine virtuelle

[__] c'est un Système d'Exploitation

[__] on doit installer un Système d'Exploitation dans chaque conteneur

[__] nécessite un ordinateur physique pour fonctionner

[__] nécessite un ordinateur logique pour fonctionner

[__] nécessite un hôte sous Linux

[__] un conteneur est un processus

[__] un conteneur est un espace confiné où s'exécutent un ou des processus

Premiers pas

Images et Conteneurs

Pour pouvoir créer un conteneur il nous faut une image.

Une image est un modèle qui sert de base de départ pour la création d'un conteneur, un peu comme si on décompressait un fichier *ZIP* dans un dossier. Ce n'est pas exactement la même chose, mais l'idée en est assez proche.

L'image est une sorte de recette de cuisine pour créer des conteneurs "à son image".

Pour reprendre l'analogie du *ZIP* : on peut décompresser un *ZIP* autant de fois qu'on le souhaite, dans autant de dossiers que nécessaire. Chaque dossier s'apparentant un peu à un conteneur : chaque dossier provenant du *ZIP* peut contenir des fichiers et des dossiers qui eux-mêmes contiennent encore d'autres éléments et ainsi de suite. Chaque dossier est une mini-arborescence à lui tout seul, créée à l'identique depuis le modèle, depuis le *ZIP*. Dans cette arborescence il peut y avoir des programmes prêts à être exécutés.

Une image, elle aussi, sert à créer des conteneurs, qui peuvent être aussi nombreux qu'on le souhaite. Tous ces conteneurs ont donc le même contenu de départ que l'image qui a servi de modèle.

Une image contient, elle aussi, une arborescence de fichiers et de dossiers qui vont constituer l'intérieur du/des conteneurs qu'on va créer "à son image". Il peut y avoir (il y a même forcément), à l'intérieur, des programmes prêts à être exécutés depuis cette arborescence issue de l'image.

Même si on a commencé ce chapitre en faisant l'analogie suivante :

Image ⇨ Conteneur *est similaire à* ZIP ⇨ Dossier

avec un conteneur il y a une caractéristique supplémentaire qui fait toute la différence : un conteneur est en quelque sorte *vivant*, il n'existe que parce qu'il contient un (parfois plusieurs) processus actifs. En fait, le conteneur est tout simplement ce processus. Tous deux ne font qu'un.

Hello World!

On l'a évoqué lors d'un TP précédent (Coder en C avec VSC[11]), il est une tradition en informatique : on fait généralement ses premiers pas en programmation en codant un petit exercice qui affiche à l'écran : **`Hello World!`**

[11] C'est un TP d'Algo.

Docker n'est pas un langage de programmation, mais il existe quand même une petite image de test qui reprend cette tradition en affichant cette phrase célèbre[12] à l'écran. L'informaticien.ne aime les traditions.

Voici comment créer et lancer votre 1er conteneur sur la base d'une image **`hello-world`**.

Dans ce TP nous allons utiliser 2 Terminaux : **`T1`** et **`T2`**, c'est ainsi qu'ils seront appelés à partir de maintenant. Si vous le souhaitez, vous pouvez changer le titre de vos terminaux depuis le menu **`Terminal > Définir le titre`**.

Etape 1 - Récupération de l'image

Placez-vous dans **`T1`**.

Il vous faut d'abord récupérer l'image depuis un **Dépôt**[13].

Un dépôt est un espace de stockage des images mises à votre disposition.

Généralement il s'agit d'un dépôt public. Le dépôt officiel est celui de la société Docker :

`hub.docker.com`

Sur les ordinateurs de l'IUT, c'est un peu différent. Pour des raisons techniques et sécuritaires, le dépôt officiel de Docker n'est pas accessible. Vous devrez donc utiliser celui qu'on a mis à votre disposition. Pour information, il se trouve ici mais il est inutile d'aller visiter cette URL :

`docker.iutlan.etu.univ-rennes1.fr`

Par contre, si vous souhaitez reproduire les exercices des TP chez vous, vous devrez utiliser le dépôt public de Docker.

La récupération de l'image se fait à l'aide de la commande suivante :

```
docker image pull hello-world
```

A ce stade, l'image **`hello-world`** est arrivée sur votre ordinateur. Elle est placée aussi dans un dépôt, mais celui-ci est local. Un **`docker pull`** ne récupérera une image depuis le dépôt public que s'il ne la trouve pas déjà localement sur votre ordinateur.

Etape 2 - Création du conteneur

Toujours dans **`T1`**.

Il ne vous reste plus qu'une action : créer le conteneur depuis cette image récupérée à l'étape précédente, et lui donner vie :

[12] Elle est célèbre car c'est Brian Kernighan et Dennis Ritchie, créateurs du langage C, qui ont proposé ce premier code dans leur livre d'apprentissage du C.
[13] Repository en anglais.

```
docker container run hello-world
```

Vous devez observer l'affichage du message **`Hello from Docker!`** avec plein d'autres lignes d'explication ensuite. On retiendra juste de cet essai que… ça fonctionne !

Tic tac

Dans la série des choses inutiles, voici une autre image qui ne restera pas dans les annales, mais qui va permettre d'illustrer nos propos.

Voici les étapes à suivre :

- Placez-vous dans **`T1`**
- Récupérez (**`pull`**) l'image **`clock`**
- Créez un conteneur (**`run`**) à partir de cette image
- Observez quelques instants son exécution (nous vous endormez pas)

Q.1 Dans **`T2`**, affichez tous les processus, sous forme arborescente, tournant dans votre système (cf TP 3 en R1.04).

Voyez-vous le processus **`tictac`** ?

Quel processus a lancé ce processus ?

Quel utilisateur a lancé ce processus ?

Dans cet affichage, repérez le **`PID`** du processus **`tictac`**.

Vous noterez que le processus **`tictac`** affiche aussi son propre **`PID`** (dans le terminal **`T1`**) mais que, bizarrement, il ne semble pas du tout être le même que celui affiché dans **`T2`** !

Vous avez simplement sous les yeux un premier constat qu'un conteneur vit sa vie de façon isolée du monde extérieur, et que ce monde a ses propres règles, comme notamment qu'un processus, vu de l'intérieur du conteneur, possède une identité[14] différente de celle sous laquelle il apparaît vu du monde extérieur (l'hôte).

Q.2 Dans **`T2`**, arrivez-vous à lancer manuellement une commande **`tictac`** ? (juste une commande **`tictac`**, pas un **`docker run clock...`**)

Explications

On l'a déjà évoqué, un conteneur permet d'exécuter un processus de façon isolée du reste du monde, et en particulier de son hôte.

[14] Le **PID** est l'identité unique d'un processus dans son environnement.

L'expérience avec le conteneur **clock** montre que le conteneur dispose de son environnement d'exécution : la commande **tictac** qui s'exécute est pourtant impossible à lancer (introuvable[15]) sur l'hôte, dans un Terminal (testé à la Q.3).

Alors, où est-elle ?

Evidemment, c'est bien dans le conteneur que se trouve **tictac**, et c'est bien depuis ce conteneur que s'exécute **tictac**.

Comment vérifier tout cela ? On va le voir maintenant.

Exploration rapide d'un conteneur

On pourrait le vérifier en regardant ce que contient le conteneur.

Quand on crée un conteneur et si on ne spécifie rien d'autre, c'est le programme, la commande, l'exécutable qui a été configuré par défaut par le créateur de l'image qui se lance (ici c'est la commande **tictac**).

> On peut créer un conteneur en précisant le nom d'un autre programme que celui prévu par défaut, à exécuter juste après la création du conteneur.

Nous allons utiliser cette technique pour explorer un peu notre conteneur **clock**.

Pour le moment, laissons ce conteneur **clock** et sa commande **tictac** tourner dans **T1**. Nous allons lui rendre une petite visite prochainement…

Dans **T2**, voici la syntaxe à utiliser :

```
docker container run clock which tictac
```

Cette commande crée un autre conteneur **clock**, mais cette fois-ci, par l'ajout d'un **which tictac**, on indique au conteneur quelle est la commande à exécuter, à la place de la commande par défaut qui est **tictac**.

Si vous ne vous rappelez plus ce que fait la commande **which**, reprenez vos notes (ou cf TP 3) ou cherchez dans le **man**.

Q.3 Qu'a affiché le conteneur ?

Dans **T2**, lancez un affichage de tous les processus du système, voyez-vous une commande **which** tourner ?

[15] On pourrait lancer une recherche de la commande **tictac** sans arriver à la trouver sur le FS de l'hôte.

Est-ce que le conteneur dans **T2** est toujours en cours d'exécution comme c'est actuellement le cas pour le 1er conteneur lancé dans **T1** ?

Qu'en déduisez-vous concernant le conteneur que vous venez de lancer, existe-t-il toujours ?

> On a déjà évoqué (en CM et précédemment dans ce TP) le fait qu'un conteneur **est un processus**.
>
> En fait, un **conteneur** et le **processus** qui s'exécute à l'intérieur du conteneur sont la même chose.
>
> Ce qui le différencie des autres processus du système hôte est qu'il a une vision très restreinte du monde dans lequel il vit, dans lequel il s'exécute :
> • Il ne voit que les fichiers qui sont dans le conteneur.
> • Il ne voit que les autres processus qui ont été lancés (généralement par lui d'ailleurs) dans le conteneur.

Evaporation des conteneurs ?

Avant de revenir sur notre **tictac** qui continue inlassablement d'égrener le temps dans **T1**, posons-nous la question de ce que devient un conteneur qui arrête son exécution. Car c'est bien ce qui s'est passé avec le test de **which tictac** de la question précédente, le conteneur s'est arrêté après avoir exécuté le **which**.

Quand un processus se termine (cf R1.04 TP 3), il disparaît, il n'apparaît plus dans la liste d'un **ps** par exemple.

Pour un conteneur, on pourrait se dire que c'est la même chose, surtout depuis qu'on sait qu'un conteneur est un processus, celui qui est précisément lancé à la création du conteneur.

Mais un conteneur est aussi un environnement d'exécution pour le processus. En cela il est donc un peu plus qu'un simple processus.

Alors, évaporation totale du conteneur ou pas ? Vérifions ensemble dans **T2** :

```
docker container ps
```

A l'instar de la commande **ps** qui liste les processus actifs du système, cette commande liste les conteneurs actifs.

Q.4 Qu'a affiché cette commande ?

Y trouvez-vous des choses familières ?

Voici une petite explication des colonnes affichées, une ligne par conteneur :

- **`CONTAINER ID`** : un identifiant unique du conteneur (on y revient plus loin).
- **`IMAGE`** : l'image qui a servi de modèle pour créer le conteneur.
- **`COMMAND`** : la commande exécutée dans le conteneur, c'est-à-dire soit celle configurée par défaut dans l'image, soit celle passée en remplacement sur la ligne du **`docker container run`**.
- **`CREATED`** : âge du conteneur.
- **`STATUS`** : état actuel du conteneur (**`Up`** signifie qu'il est actif, en cours d'exécution). On y revient plus loin.
- **`PORTS`** : vu plus tard.
- **`NAMES`** : un petit nom unique pour chaque conteneur. On peut le spécifier à la création du conteneur, ou laisser le hasard faire les choses[16].

La commande précédente affichait uniquement les conteneurs actifs. Il existe une option permettant de lister tous les conteneurs (y compris ceux qui ne sont plus actifs) :

```
docker container ps -a
```

et avec cette option **`-a`**[17], on découvre que, contrairement aux processus qui disparaissent totalement une fois leur exécution terminée, pour les conteneurs ce n'est pas tout à fait la même chose.

Q.5 Qu'a affiché cette commande ?

Y trouvez-vous encore des choses familières ?

La colonne **`STATUS`** montre des informations du genre **`Exited ...XX... ago`**.

Notez que la valeur entre **`()`** est généralement la valeur retournée par la fonction **`main()`**, à savoir le **`EXIT_SUCCESS`** ou **`EXIT_FAILURE`**, ou parfois d'autres valeurs. Vous verrez cela en BUT2.

La colonne **`COMMAND`** est aussi intéressante en principe. Y trouvez-vous le **`which tictac`** ?

Arrêt d'un conteneur

Il est temps de revenir voir notre **`tictac`** dans **`T1`**.

Q.6 Essayez d'arrêter le processus au clavier (cf TP 3). Y arrivez-vous ?

[16] Le hasard n'est pas toujours très inspiré !
[17] **`a`** pour **`all`**

Certains processus peuvent être arrêtés au clavier, et d'autres (comme ça devrait être le cas ici de notre **tictac**) ne l'autorisent pas. On ne va pas entrer dans le détail du pourquoi ni du comment, ce n'est pas très important.

Il existe des commandes pour agir sur les conteneurs.

Toutes les actions nécessitent, comme pour les processus, de pouvoir indiquer la cible de l'action. Dans Docker, ce n'est pas le **PID** d'un processus mais l'**ID** unique du conteneur qu'on utilise. Cet **ID** est dans la colonne **CONTAINER ID**.

Notez que cet **ID** est une chaîne hexadécimale qui fait 64 caractères de long, et dont seulement les 12 premiers sont indiqués dans l'affichage d'un **docker container ps**.

En fait, 12 caractères sont bien suffisants car la probabilité d'avoir les 12 premiers caractères identiques entre deux conteneurs est extrêmement faible[18]. On peut même utiliser seulement les quelques chiffres hexadécimaux du début d'un **ID**, à partir du moment où il n'y a aucune ambigüité avec un autre **ID** ayant ces mêmes chiffres. Souvent 2 ou 3 chiffres sont largement suffisants.

A l'aide de la commande précédente, localisez l'**ID** du conteneur **clock** qui exécute actuellement **tictac** dans **T1**.

Dans T2, tapez une commande en utilisant la syntaxe (remplacez **<ID>** par la bonne valeur) suivante :

```
docker container stop <ID>
```

Q.7 Que s'est-il passé dans **T1** ?

Vérifiez que le conteneur est bien listé comme **Exited** maintenant, avec un :

```
docker container ps -a
```

Redémarrage d'un conteneur

L'action **stop** a arrêté le conteneur **clock**, qui est maintenant noté **Exited**.

Qui dit **stop**, laisse entendre peut-être aussi un **start** ?

Q.8 Dans **T2**, tentez l'expérience d'un **start** à la place du **stop**, sur le même conteneur. Que se passe-t-il ?

[18] 1 chance sur ~281 475 milliards.

Le conteneur est-il redémarré ?

Vérifiez avec la commande adaptée.

Q.9 Dans **T1**, l'affichage de **tictac** a-t-il repris ?.

Explications

Quand on **run** un conteneur il est attaché au Terminal : son **STDIN** et ses **STDOUT** et **STDERR** sont attachés au Terminal. S'il écrit des choses (**printf(...)**), ça s'affiche sur le Terminal.

Quand on **stop** un conteneur, le processus du conteneur s'arrête et disparaît de la liste des processus (cf un **ps -a**), mais on a vu que le conteneur demeure présent dans la liste des conteneurs, dans un état **Exited**. Plus de processus en exécution signifie, plus de rattachement à un Terminal.

Quand on **start** un conteneur qui a été **stop**, celui-ci reprend vie, un processus **tictac** est à nouveau créé et c'est reparti !

Le problème est que rien ne permet de lier **tictac** au Terminal **T1**, qui peut même avoir été fermé depuis ! Le nouveau processus **tictac** est donc créé… détaché de tout Terminal !

Alors, où se font les affichages ?

Logs d'un conteneur

Les affichages (on appelle cela les logs) ne sont pas perdus, ils sont conservés quelque part par Docker et une commande permet d'y accéder. Voici sa syntaxe (remplacez **<ID>** par la bonne valeur) :

```
docker container logs -f <ID>
```

cette commande **logs** est similaire au filtre **tail** qui affiche la fin d'une source de données, généralement un fichier. C'est effectivement la même chose et ici il s'agit simplement du **STDOUT** du conteneur. Le **-f**[19] permet un affichage sans fin, dès que de nouvelles lignes arrivent, elles sont affichées sur le Terminal.

Par contre, un **CTRL+C** mettra fin au **tail -f** mais n'aura aucune action sur le conteneur, il ne permettra pas d'interrompre le conteneur. Pour ce faire, il faudra donc utiliser encore un **docker container stop <ID>**.

[19] **f** comme **follow**, c'est-à-dire : suivre l'arrivée de nouvelles lignes.

Intrusion dans un conteneur

Maintenant qu'on sait créer, lister, arrêter et redémarrer un conteneur, qu'on a aussi vu qu'on peut créer un conteneur en lui faisant exécuter une autre commande que celle prévue par défaut, terminons par une intrusion dans la vie d'un conteneur.

Si ce n'est pas déjà fait, redémarrez le conteneur **`clock`** qui fait tourner **`tictac`** (voir plus haut).

On a vu qu'un conteneur est un processus s'exécutant dans un environnement cloisonné.

Dans **`T1`**, à l'aide de la syntaxe suivante, que vous saurez maintenant adapter, introduisez-vous dans le conteneur **`clock`** :

```
docker container exec -ti <ID> sh
```

si tout se passe bien, vous devez obtenir un prompt Shell différent de celui qu'on a habituellement. Sans doute quelque chose du genre :

```
/ #
```

si ce n'est pas le cas, demandez de l'aide à votre enseignant·e.

Ce **`#`** est une bonne nouvelle, vous êtes **`root`** ! 💪🙏😱

Pour rappel, **`root`** est le Dieu du Système, celui qui a le droit de vie et de mort sur tout ce qui se passe chez lui, dans le Système !

Super, vous êtes **`root`**… mais uniquement dans le conteneur, car là vous êtes entré (**`exec`**) dans le conteneur et avez été accueilli·e par un Shell (**`sh`**). Vous êtes donc dans un espace très cloisonné. C'est un peu comme avoir un trône dans sa chambre d'étudiant·e, et une couronne en papier doré sur la tête. Vous êtes le la reine/le roi… de votre chambre. Mais une fois sorti·e dans le couloir, c'est un dur retour à la réalité des choses, vous redevenez madame ou monsieur tout-le-monde.

Ici, c'est pareil, donc pas besoin de vous exciter !

Explications sur la syntaxe de la commande **`exec`** :

`exec` permet d'exécuter une commande dans un conteneur actif (uniquement)

`-ti` est la forme raccourcie de **`-t -i`**

-t[20] signifie que la commande doit être exécutée attachée à un Terminal

-i[21] signifie que la commande pourra lire des choses au clavier.

sh est simplement la commande à exécuter, ici il s'agit simplement d'un shell : **sh** est un Shell, comme **bash**, mais en plus rudimentaire et cette image **clock** ne contient que **sh**, pas de **bash**.

Traitez les questions ci-après depuis ce shell **sh** que vous venez de lancer dans le conteneur **clock** :

Q.10 Vérifiez la présence de la commande **tictac** dans ce conteneur. Pour ce faire, utilisez la commande qui localise une commande dont le nom est passé en paramètre. Rappel : on l'a déjà utilisée à la fin de la Q.2.

Q.11 Listez tous les processus en cours d'exécution dans ce conteneur.

Y voyez-vous un **tictac** en cours ?

Q.12 Essayez de lancer aussi un nouveau **tictac**. Depuis **T2** listez tous les processus.

Voyez-vous 2 **tictac** maintenant ?

Ont-ils un lien de parenté proche ?

Arrêtez au clavier le **tictac** que vous venez de lancer. Cette fois-ci vous devez y parvenir car nous avons obtenu un accès dans le conteneur en mode interactif (**-ti**).

Q.13 Maintenant que vous êtes en visite dans le conteneur, y trouvez-vous votre espace personnel (**/FILER/HOME/INFO/votre_login**) ?

Pourquoi ?

Quittez votre intrusion dans le conteneur (**exit** ou **CTRL+D**)

Suppression d'un conteneur

Il est temps de faire un peu de ménage.

La syntaxe pour supprimer un conteneur est la suivante :

```
docker container rm <ID>
```

Attention, ça supprime le conteneur et son… contenu !

[20] **t** comme **TTY** (cf R1.04 TP 3)
[21] **i** comme **interactive**

Q.14 Dans **T1**, essayez de supprimer le conteneur **clock**.

Y arrivez-vous ?

Que se passe-t-il ?

Corrigez la situation pour pouvoir ensuite supprimer le conteneur. Note : certaines actions peuvent parfois prendre quelques secondes pour rendre la main à l'utilisateur.

Profitez-en pour supprimer tous les conteneurs arrêtés qui ne vous servent plus.

Note : certaines commandes peuvent opérer sur plusieurs conteneurs en même temps. Par exemple **docker stop <ID1> <ID2> ...** ou encore **docker rm <ID1> <ID2> ...** . N'hésitez pas à globaliser/factoriser vos commandes ainsi.

Injection dans un conteneur

Nous avons vu qu'un conteneur est un environnement confiné. Ce qui est dedans n'est pas visible de dehors (de l'hôte) et réciproquement.

Ça risque quand même de poser des problèmes pour échanger des données, des fichiers.

Il y a des solutions, que nous allons étudier dans un prochain sujet de TP.

En attendant, voici déjà une façon d'injecter des fichiers dans un conteneur actif.

Q.15 Dans **T1**, récupérez l'image **imagick** (elle est nouvelle pour nous, alors remontez en début de sujet si vous avez un trou de mémoire concernant la façon de récupérer une image).

On a vu qu'un conteneur exécute (ou encore "est assimilé à") un processus et que le conteneur s'arrête quand le processus en question s'arrête.

L'image **imagick** ne contient pas de processus sans fin comme dans le cas de **clock**.

Le processus qui est configuré pour s'exécuter dans **imagick** est un simple **bash**. Or **bash** ne fonctionne qu'en mode Interactif (il lit des commandes au clavier et les exécute). Il est donc indispensable de créer un conteneur attaché au Terminal et qui fonctionne en mode Interactif.

C'est nouveau pour nous mais on a fait quelque chose de très similaire avec le **docker container exec** un peu plus tôt : on lui a passé des options (**-t -i**, raccourcies en **-ti**) pour s'attacher à un conteneur existant en mode Interactif. Les options pour, cette fois-ci, lancer un conteneur en mode Interactif sont exactement les mêmes :

```
docker container run -ti imagick
```

Quand tout est prêt dans **T1**, depuis **T2** utilisez cette syntaxe pour injecter un fichier dans le conteneur (attention, adaptez cette syntaxe) :

```
docker container cp fic_source_sur_hote <ID>:chemin_absolu_dest_container
```

Téléchargez une image couleur PNG libre de droit depuis Internet, renommez-là **orig.png**, puis injectez-là dans le dossier **/work** de votre conteneur à l'aide de la commande précédente.

Ensuite, dans **T1**, exécutez cette commande :

```
convert orig.png -colorspace gray out.png
```

qui convertit une image couleur en nuances de gris.

Enfin, récupérez votre résultat avec une commande similaire à celle de l'injection, mais dans l'autre sens (à adapter aussi) :

```
docker container cp <ID>:chemin_absolu_source_container fic_dest_sur_hote
```

Observez le résultat de votre image convertie. Est-ce bon ?

Qu'en tire-t-on comme avantage ?

Vous venez de voir quelques commandes de base de manipulation des conteneurs avec Docker.

Il est très probable que vous ayez déjà Image Magick sur votre machine à l'IUT, mais en supposant que cet outil ne soit pas installé, ou le soit dans une version trop ancienne par exemple, nous venons de voir ensemble comment utiliser un outil sans avoir besoin de l'installer sur son ordinateur.

En tout cas, il n'est pas venu polluer votre espace de travail. Si vous n'en avez qu'un usage très ponctuel et exceptionnel, rien ne sert d'installer des choses qui resteront ensuite très longtemps (parfois définitivement), au risque même de venir perturber votre système.

Nous allons avoir l'occasion de faire des expériences plus complexes et plus intéressantes dans les prochaines semaines de SAÉ.

Comment exploiter Docker chez vous ?

Il est aussi possible d'installer Docker chez vous.

Il existe des solutions sur Windows et sur macOS. Ça sort du cadre de cette SAÉ mais nous pouvons en discuter en aparté pour ceux qui sont intéressés. Sachez qu'il vous faudra une machine capable de faire tourner Virtualbox.