Ateliers Bash et Docker

IFT-2001: Systèmes d'exploitation
GLO-2001: Systèmes d'exploitation pour l'ingénierie

Été 2023

1 Installation

Cet atelier nécessite l'utilisation de la machine virtuelle du cours, que vous pouvez télécharger à partir du lien suivant : LIEN vers VM. Il est aussi possible de configurer une machine virtuelle en suivant les instructions de willGuimont/ateliers-iftglo-2001. Une fois que vous avez téléchargé la machine virtuelle, vous pouvez l'ouvrir à l'aide de l'application VirtualBox. Pour vous authentifier sur la machine virtuelle, utilisez le nom d'utilisateur `glo2001` et le mot de passe `glo2001`.

Une fois authentifié, ouvrez un terminal avec `CTRL-ALT-T` où vous pouvez exécuter des commandes. Pour vérifier que la machine virtuelle fonctionne correctement, exécutez la commande suivante dans le terminal :

```
echo 'Hello world'
```

Si tout est configuré correctement, vous devriez voir s'afficher dans la console le texte `Hello world`. Pour activer le partage du presse-papier (*clipboard*), modifiez l'option dans `Devices -> Shared Clipboard -> Bidirectional`.

2 Consignes

La machine virtuelle fournie est déjà configurée correctement pour les ateliers. Les commandes de corrections sont de la forme `correction_nn`, où `nn` est un nombre entier représentant le numéro de l'exercice, par exemple `correction_03` pour l'exercice 3. Les instructions pour valider les commandes seront fournies avec la première question ayant un résultat attendu.

Pour réaliser cet atelier, vous devrez effectuer chaque exercice directement dans le terminal. Vous devrez utiliser l'éditeur de texte `nano` pour modifier les scripts demandés. Pour ouvrir un fichier avec nano, exécutez la commande suivante dans le terminal.

```
nano test.sh
```

Vous pourrez alors éditer le fichier, inscrivez y

```
echo 'Hello world'
```

Pour quitter nano, appuyez sur `CTRL-X`, puis répondez `y` pour sauvegarder les modifications. Exécutez les commandes suivantes pour valider que le tout fonctionne :

```
chmod +x test.sh
./test.sh
```

Ces commandes devraient afficher le texte `Hello world`.

Pour les aventuriers seulement : Si vous êtes à l'aise avec la ligne de commande, nous vous encourageons à essayer de réaliser cet atelier en utilisant `vim` comme éditeur de texte. Une brève introduction aux commandes de `vim` est disponible sur le site de The Missing Semester par le MIT. `vim` offre de nombreux raccourcis pour éditer du texte et du code de manière très efficace. En tant que programmeur, vous passerez beaucoup de temps à écrire du code, il est donc judicieux d'investir dans l'apprentissage de `vim` pour le reste de votre carrière. Il existe des *plugins* émulant les commandes `vim` pour la majorité des IDEs : Vim pour vscode ou IdeaVim pour les produits JetBrains **Veuillez noter qu'aucune assistance concernant `vim` ne sera fournie pendant cet atelier.**

3 Solution

Les solutions proposées aux exercices de ces ateliers sont disponibles dans le dépôt GitHub suivant : willGuimont/ateliers-iftglo-2001.

4 Bash

C'est une journée comme les autres au bureau. Vous arrivez tôt le matin, votre café à la main et votre ordinateur portable sous le bras. Alors que vous entrez dans votre département, vous remarquez que votre patron, habituellement calme et détendu, marche nerveusement dans le couloir. Son expression livide et ses mains tremblantes attirent immédiatement votre attention.

« Inquiétant, que se passe-t-il ? » lui demandez-vous, soucieux.

« La production a planté », répond-il d'une voix tremblante. « Nous recevons des centaines d'appels de clients, plus rien ne fonctionne... »

Votre cœur s'accélère alors que vous réalisez l'ampleur du problème. Votre entreprise dépend d'un serveur qui héberge la majeure partie de ses services, et il semble qu'il y ait un dysfonctionnement. Votre patron vous fixe droit dans les yeux.

« Résoudre ce problème est votre priorité », vous dit-il avec fermeté.

Il vous tend une feuille de papier avec les identifiants de connexion de l'utilisateur `glo2001` (dont le mot de passe est aussi `glo2001`...) pour accéder au serveur. Vous vous sentez à la fois nerveux et excité à l'idée de résoudre ce problème critique. Votre patron vous conduit alors à la salle des serveurs, où un vieil écran cathodique et un clavier vous attendent. Il vous explique qu'il s'agit d'un système d'exploitation Linux en version serveur, donc sans interface graphique. Vous devrez donc résoudre les problèmes en utilisant uniquement le terminal.

4.1 La ligne de commande et le manuel

Ouvrez un terminal avec le raccourci `CTRL-ALT-T`, ou appuyez sur la touche `super` (la touche Windows sur le clavier) et cherchez pour l'application `Terminal`.

Comme première étape, vous décidez d'inspecter les fichiers présents sur le serveur afin de trouver les fichiers de log. Cependant, vous n'êtes pas certain de la commande à utiliser pour cela.

Heureusement, la ligne de commande vous offre la possibilité de connaître le fonctionnement de chaque commande. La commande `man` permet de lire la page du manuel correspondant à une commande spécifique. Par exemple, pour obtenir la documentation sur la commande `ls`, vous pouvez utiliser la commande suivante :

```
man ls
```

Une fois que vous avez ouvert la page du manuel, vous pouvez naviguer à l'aide des flèches du clavier pour lire le contenu. Pour quitter la page du manuel, appuyez simplement sur la touche `q`.

Aussi, certaines commandes acceptent l'argument `--help` qui affiche un message d'aide décrivant les arguments que l'on peut passer à la commande. Par exemple :

```
ls --help
```

En exécutant cette commande, vous obtiendrez un message d'aide détaillant les différentes options et arguments que vous pouvez utiliser avec la commande `ls`.

N'hésitez pas à utiliser l'argument `--help` avec les commandes que vous souhaitez explorer afin d'obtenir des informations supplémentaires sur leur utilisation. Cela peut vous aider à mieux comprendre les fonctionnalités disponibles et à utiliser correctement les commandes dans votre exploration du serveur.

Pour certaines commandes comme `cd` (qui est une commande incluse directement dans le *Shell Bash*), il faut plutôt utiliser

```
help cd
```

NOTEZ BIEN: Raccourcis dans le terminal

Auto complétion : Vous pouvez utiliser la touche `TAB` pour obtenir de l'auto complétion dans le terminal.

Quitter une commande : Dans un terminal, le raccourcit `CTRL-C`, plutôt que de copier, termine l'exécution d'une commande. Par exemple, la commande `yes` répète indéfiniment la lettre `y` dans le terminal. Pour quitter, appuyer sur `CTRL-C` :

```
yes
# CTRL-C pour quitter l'execution de yes
```

Pour copier et coller, le terminal utilise plutôt `CTRL-SHIFT-C` et `CTRL-SHIFT-V`
Pour fermer un terminal, vous pouvez utiliser `CTRL-D`.

Exercice 1: Commandes de base

Cet exercice vise à vous familiariser avec l'utilisation de quelques commandes de base qui vous seront utiles tout au long de l'atelier. Pour ce faire, utilisez la commande `man` et l'argument `--help` pour obtenir des informations détaillées sur le fonctionnement de chaque commande. Dans le fichier `exercice_01.txt`, décrivez brièvement l'utilité de chacune des commandes suivantes. Cet aide mémoire vous sera utile tout au long de l'atelier. Vous pourrez le consulter avec la commande `cat exercice_01.txt`.

Ouvrez `exercice_01.txt` avec `nano` avec :

```
nano exercice_01.txt
```

Dans un autre terminal (`CTRL-ALT-T` pour ouvrir un autre terminal ou `CTRL-SHIFT-T` pour ouvrir un autre terminal dans un onglet), utilisez le terminal pour déterminer le comportement des commandes suivantes :

Commandes
`nano`
`cd`
`ls`
`cat`
`mkdir`
`rm`
`rmdir`
`mv`
`pwd`
`cp`
`chmod`
`touch`

Solution: 1

Commande	Définition
`nano`	Éditeur de texte simple
`cd`	Permet de changer de dossier courant
`ls`	Permet de lister les fichiers dans un dossier
`cat`	Affiche le contenu d'un fichier
`mkdir`	Créé un dossier
`rm`	Supprime un fichier ou un dossier
`rmdir`	Supprime un dossier vide
`mv`	Déplacer un fichier
`pwd`	Affiche le chemin du répertoire courant
`cp`	Copie un fichier
`chmod`	Change les permissions d'un fichier
`touch`	Met à jour le timestamp d'un fichier, ou le crée s'il n'existe pas

Maintenant que vous êtes familiarisé avec quelques commandes de base, il est temps d'inspecter les fichiers du serveur et trouver les fichiers de log du serveur.

Exercice 2: Navigation du système de fichiers

Utilisez les commandes listées plus haut afin d'explorer l'arborescence de dossier et trouver le fichier avec l'extension `.log`.

Notez bien: Voici quelques chemins spéciaux
- `.` représente le dossier courant ;
- `..` représente le dossier parent ;
- ∼ représente le dossier `home` de l'utilisateur (`/home/glo2001/`) ;
- `-` représente le chemin du dernier dossier visité.

Vous pouvez utiliser ces chemins spéciaux avec plusieurs commandes, notamment `cd`.

```
# Navigue dans le dossier abc/def
cd abc/def

# Navigue dans le dossier parent (abc)
cd ..

# Retourne au dossier home
cd
# ou
cd ~

# Retourne au dossier precedant (abc)
cd -
```

1. Listez les fichiers dans le répertoire courant
2. Déplacez-vous dans les différents dossiers et tentez de trouver le fichier portant l'extension `.log`
3. Copiez le chemin **absolu** (depuis la racine du système de fichier `/`) du dossier dans lequel se trouvent les fichiers de log dans le fichier ∼`/out_02.txt`. Ne pas ajouter de nouvelle ligne à la fin.

Exécutez la commande de correction `correction_02.sh` pour valider votre réponse.

Solution: 2 Voici les commandes à exécuter pour réaliser l'exercice

1. `ls`
2. `cd TODO`
3. `ls`
4. `pwd`

Le chemin attendu est `/home/glo2001/ApplicationData/output/logs`

Maintenant que vous avez trouvé le fichier de log, il est temps d'en faire une copie dans votre dossier `home`.

Notez bien: Le dossier home est l'endroit où sont stockés les fichiers personnels d'un utilisateur. Chaque utilisateur a son propre dossier dans `/home/`. Dans votre cas, votre utilisateur a comme nom d'utilisateur `glo2001`, alors sont dossier home est situé à `/home/glo2001`.
Il existe aussi un raccourcit pour référer au dossier home ∼. Ainsi, chacune des commandes suivantes vous permet de retourner au dossier home

```
# Utilisation de chemin absolu
cd /home/admin
# Utilisation de tilde
cd ~
# Sans argument, cd retourne au dossier home
cd
```

Exercice 3: Dossier `home` et copie

Copiez le fichier `.log` dans le dossier `~/log_backup`.

1. Retournez dans votre dossier `home` ;
2. Créez un nouveau dossier qui s'appelle `log_backup` dans votre dossier `home` ;
3. Copiez le fichier .log dans le dossier backup, en lui donnant le nom `build_backup.log`.

Exécutez la commande de correction `correction_03.sh` pour valider votre réponse.

Solution: 3 Liste des commandes pour réaliser l'exercice

1. `cd`
2. `mdkdir log_backup`
3. `cp ~/ApplicationData/output/logs/build.log ~/log_backup/build_backup.log`

Maintenant que vous avez effectué une sauvegarde des logs de votre application, vous décidez d'exécuter le script de diagnostic du projet. Veuillez retourner dans le répertoire du fichier log et tenter d'exécuter le script `./diagnostic.sh`.

Cependant, vous rencontrez une erreur de permissions. Pour résoudre ce problème, inspectez les permissions du script en utilisant la commande `ls -l`. Ensuite, modifiez les permissions pour rendre le script exécutable.

Exercice 4: Permissions

Modifiez les permissions du script `diagnostic.sh`, puis exécutez le script.

- Inspecter les permissions du fichier ;
- `chmod` avec l'argument `+x` pour modifier les permissions ;
- Exécuter `diagnostic.sh`.

Exécutez la commande de correction `correction_04.sh` pour valider votre réponse.

Solution: 4

```
# On regarde les permissions
ls -l
# On change les permissions
chmod +x diagnostic.sh
# On execute le script
./diagnostic.sh
```

Le script a généré une dizaine de fichiers `.out` contenant les résultats de l'analyse du système. Vous devez déplacer ces fichiers dans un nouveau dossier nommé `output`. Au lieu de déplacer chaque fichier manuellement avec la commande `mv out_01.out output/`, ce qui serait fastidieux, vous pouvez utiliser le caractère générique *wildcard* (`*`), qui permet de sélectionner plusieurs fichiers à la fois.

Avant de déplacer les fichiers, vous pouvez essayer la commande `cat *`. Cette commande affiche le contenu de tous les fichiers présents dans le répertoire courant.

Cependant, dans votre cas, vous ne souhaitez pas sélectionner tous les fichiers. Vous pouvez spécifier un format spécifique en ajoutant un préfixe ou un suffixe aux noms des fichiers. Par exemple, pour sélectionner tous les fichiers `.sh`, vous pouvez utiliser la commande `ls *.sh`. Cela affichera la liste des fichiers portant l'extension `.sh`.

Exercice 5: Wildcards

- Créez un dossier nommé `output_backup` dans le dossier `~/ApplicationData/output/logs` ;
- Déplacez tous les fichiers terminant par `.out` dans le dossier en une seule commande.

Exécutez la commande de correction `correction_05.sh` pour valider votre réponse.

Solution: 5

```
# Deplacer dans le bon dossier
cd ~/ApplicationData/output/logs
# Creation du dossier output
mkdir output_backup
# Deplacer les fichiers dans le dossier
mv *.out output_backup/
```

En plus de générer des fichiers .out, le script a également généré des fichiers temporaires `.tmp` et un dossier nommé `temp`. Ces fichiers et ce dossier peuvent être supprimés.

Exercice 6: Suppression de fichiers et de dossiers

- Supprimez les fichiers ayant l'extension `.tmp` ;
- Supprimez le dossier `temp`.

Exécutez la commande de correction `correction_06.sh` pour valider votre réponse.

Solution: 6

```
# Suppression des fichiers .tmp
rm *.tmp
# Suppression du dossier
rm -r temp
```

4.2 Scripts

Pour simplifier la tâche de déplacement et de suppression des fichiers générés par le script `diagnostic.sh`, vous pouvez créer un script qui automatisera ces actions pour vous.

Notez bien: Voici un exemple de script Bash

```
#!/usr/bin/env bash
echo 'Debut du script'
ls *.sh
echo 'Fin du script'
```

Dans ce script, la ligne `#!/usr/bin/env bash` est appelée un *shebang* (she = `#`, bang = `!`). Elle indique à l'interpréteur quel programme doit être utilisé pour exécuter le script, dans ce cas, il s'agit de Bash. Il serait aussi possible de spécifier un autre programme comme interpréteur. Par exemple, pour interpréter le script comme du Python3, on utiliserait le shebang suivant : `#!/usr/bin/env python3`.

Utiliser `#!/usr/bin/env bash` est généralement préférable à `#!/bin/bash`, car cela permet de rechercher l'emplacement de l'exécutable Bash dans l'environnement de l'utilisateur, ce qui le rend plus portable d'un système à l'autre.

Vous pouvez créer un fichier texte avec l'extension `.sh` (par exemple, `exemple.sh`), y copier le script ci-dessus, puis rendre le fichier exécutable à l'aide de la commande `chmod +x exemple.sh`. Ensuite, vous pourrez exécuter le script en utilisant `./exemple.sh` pour exécuter le script.

Exercice 7: Scripting de base

1. Créer un script nommé `cleanup.sh` dans le dossier `~/ApplicationData/output/logs/` ;
2. Donnez les permissions d'exécution au script ;
3. Utilisez les commandes que vous avez tapées dans les exercices précédents pour créer votre script ;
   - Créez un dossier nommé `output_backup` ;
   - Déplacez tous les fichiers terminant par `.out` dans le dossier en une seule commande ;
   - Supprimez les fichiers ayant l'extension `.tmp` ;
   - Supprimez le dossier `temp`.
4. Assurez-vous de supprimer le dossier `output` que vous avez créé plus tôt ;
5. Exécutez `diagnostic.sh` un autre fois, et tester votre script.

Exécutez la commande de correction `correction_07.sh` pour valider votre réponse.

Solution: 7

- `touch cleanup.sh`
- `chmod +x cleanup.sh`
- `nano cleanup.sh` et y ajouter le texte suivant

```
#!/usr/bin/env bash

# Creation du dossier output
mkdir output
# Deplacer les fichiers dans le dossier
mv *.out output/

# Suppression des fichiers .tmp
rm *.tmp
# Suppression du dossier
rm -r temp
```

4.3 Composition de programmes et commandes avancées

Maintenant que vous avez effectué quelques opérations de nettoyage et que vous vous êtes familiarisé avec la ligne de commande, il est temps d'analyser les logs et les sorties du programme de diagnostic. Pour cela, vous devrez combiner plusieurs commandes en utilisant le `piping` (ou pipeline) et des commandes plus avancées. Avant d'aborder le sujet, vous explorerez quelques commandes plus avancées qui vous seront utiles pour la suite.

4.3.1 Commandes avancées

Exercice 8: Commandes avancées

Comme à l'exercice 1, utilisez la commande `man` et l'argument `--help` pour obtenir des informations détaillées sur les commandes du tableau suivant. Ces commandes sont un peu différentes de celles vues à l'exercice 1, celles-ci peuvent prendre en entrée un fichier, ou on peut y `pipe` la sortie d'une autre commande, ce qui sera le sujet de la section suivante. Vous pouvez créer un fichier de test (`test_file.txt`) afin de tester les commandes.
Dans le fichier `exercice_08.txt`, décrivez brièvement l'utilité de chacune des commandes suivantes. Cet aide mémoire vous sera utile tout au long de l'atelier.

Commandes
`tac`
`less`
`find`
`grep`
`sort`
`uniq`
`wc`
`head`
`tail`
`du`
`curl`
`sed`
`awk`
`kill`
`sleep`

Solution: 8

Commande	Définition
`tac`	Inverse l'ordre des lignes d'un fichier
`less`	Pagination de texte
`find`	Recherche de fichiers
`grep`	Filtre des lignes selon un pattern
`sort`	Trie les lignes
`uniq`	Enlève les lignes doublons
`wc`	Compte le nombre de lignes, de mots et de caractères
`head`	Affiche le début d'un fichier
`tail`	Affiche la fin d'un fichier
`du`	Affiche la taille de fichiers
`curl`	Requête réseau (HTTP, FTP, etc.)
`sed`	Éditeur de ligne
`awk`	Interpréteur pour le langage `awk`
`kill`	Arrête un processus
`sleep`	Attend un nombre de secondes

Avec ces nouvelles connaissances sur les commandes Bash, il est maintenant temps de retourner à votre problème de serveur.

Exercice 9: Commandes avancées 1

Le fichier `messages.txt` dans le dossier de log contient les messages d'erreurs qui causent la panne du système. Malheureusement, ce fichier contient aussi beaucoup de log qui ne vous sont pas utiles. Plutôt que de manuellement lire l'entièreté du fichier, vous décidez d'utiliser les commandes Bash que vous venez de découvrir.

Utilisez une commande afin d'afficher toutes les lignes de `messages.txt` qui contiennent la chaîne de caractère `Error` et copiez le résultat dans le fichier `~/errors.txt`.

Exécutez la commande de correction `correction_09.sh` pour valider votre réponse.

Solution: 9

```
grep Error messages.txt
```

Exercice 10: Commandes avancées 2

Après avoir inspecté les erreurs, que vous avez copiées dans `/errors.txt`, vous réalisez qu'il y a beaucoup de doublons. Utilisez une commande Bash afin d'enlever les lignes dupliquées et copiez le résultat dans `~/errors_2.txt`.

Exécutez la commande de correction `correction_10.sh` pour valider votre réponse.

Solution: 10

```
uniq ~\errors.txt
```

Exercice 11: Commandes avancées 3

Après avoir inspecté les erreurs filtrées du fichier `~/errors_2.txt`, remplacer les erreurs possédant un code 400 (400, 403 et 404) par des avertissement. Avec `sed`, remplacer le texte en utilisant l'expression régulière
`'Error \(4[0-9]\+\)`
par le texte suivant :
`Warning \1'`
où `\1` va copier le nombre capturé dans l'expression régulière.
Indice : la commande aura la forme `sed 's/regex1/regex2/g`. `s` indique qu'il s'agit d'une **s**ubstitution que l'on applique **g**lobalement.
Utilisez une commande Bash afin modifier les messages et copiez le résultat dans `~/errors_3.txt`.

Exécutez la commande de correction `correction_11.sh` pour valider votre réponse.

Solution: 11

```
sed 's/Error \(4[0-9]\+\)/Warning \1/g' ~/errors_2.txt
```

4.3.2 Composition de programme

La composition de programme est au cœur de la philosophie Unix. Voici un extrait de *The Art of Unix Programming* [5] décrivant l'importance de la composition de programme (chapitre complet disponible ici).

> It's hard to avoid programming overcomplicated monoliths if none of your programs can talk to each other.
>
> Unix tradition strongly encourages writing programs that read and write simple, textual, stream-oriented, device-independent formats. Under classic Unix, as many programs as possible are written as simple filters, which take a simple text stream on input and process it into another simple text stream on output.
>
> Despite popular mythology, this practice is favored not because Unix programmers hate graphical user interfaces. It's because if you don't write programs that accept and emit simple text streams, it's much more difficult to hook the programs together.
>
> Text streams are to Unix tools as messages are to objects in an object-oriented setting. The simplicity of the text-stream interface enforces the encapsulation of the tools. More elaborate forms of inter-process communication, such as remote procedure calls, show a tendency to involve programs with each others' internals too much.
>
> To make programs composable, make them independent. A program on one end of a text stream should care as little as possible about the program on the other end. It should be made easy to replace one end with a completely different implementation without disturbing the other.
>
> — Chapter 1. Philosophy, Rule of Composition

Cette composition de programme peut se faire à l'aide de différents opérateurs, qui seront l'objet des prochaines sections.

Composition L'opérateur ; permet d'exécuter une commande après l'autre sur une même ligne.
(`commande1 ; commande2`)
Par exemple, pour l'exercice 7, on pourrait réécrire le programme comme suit :

```
mkdir output; mv *.out output/; rm *.tmp; rm -r temp
```

L'opérateur && permet de ne réaliser la deuxième commande que si la première a réussit (retourne un code de sortie égal à zéro).
(`commande1 && commande2`)

```
# Affiche "Hello World" car les deux commandes reussisent
echo 'Hello' && echo 'World'

# On affiche hello car le dossier existe, donc cd reussit
cd dossier_qui_existe && echo 'Hello

# On n'affiche pas Hello car le dossier n'existe pas, donc cd echoue
cd dossier_non_existant && echo 'Hello'
```

L'opérateur || permet de ne réaliser la deuxième commande que si la première a échoué (retourne un code de sortie différent de zéro).
(`commande1 || commande2`)

```
# Affiche seulement Hello, car echo 'Hello' reussit
echo 'Hello' || echo 'World'

# On n'affiche pas Hello, car cd reussit
cd fichier_qui_existe || echo 'Hello'

# On affiche hello, car cd echoue
cd fichier_non_existant || echo 'Hello'
```

Tuyaux L'opérateur | permet de passer la sortie d'une commande en entrée à une autre. Cet opérateur est aussi appelé *pipe*.
(`commande1 | commande2`)

```
# Liste les fichiers, et ne conserve que les .txt
ls | grep ".txt"

# Trie les lignes d'un fichier, ne conserve que les 5 premieres lignes
cat file.txt | sort | head -n 5
```

Redirection L'opérateur > redirige la sortie d'une commande vers un fichier, écrasant le fichier
(`commande > fichier`)

```
ls > file.txt
```

Par exemple, il est possible de télécharger un fichier texte avec `curl` :

```
curl https://www.ulaval.ca/ > ulaval.txt
```

Pour ignorer la sortie d'une commande :

```
ls > /dev/null 2>&1
```

L'opérateur < redirige l'entrée d'une commande depuis un fichier.
(`commande < fichier`)

```
wc < file.txt
```

L'opérateur >> comme > permet de rediriger la sortie d'une commande vers un fichier, mais ajoute à la fin du fichier (*append*) plutôt que de l'écraser :
(`commande >> fichier`)

```
ls >> file.txt
```

L'opérateur << comme <, mais permet de passer plusieurs lignes.
(`commande >>delim [plusieurs lignes] delim`)

```
cat <<EOF
    abc
    def
    hij
    EOF
# On peut remplacer EOF (end of file) par tout autre chaine de charactere
cat <<ABC
    abc
    def
    hij
    ABC
```

Exemples Voici quelques exemples utilisant les tuyaux et les redirections.

```
# Trouve les 5 premiers .txt en ordre alphabetique
ls -l | grep ".txt" | sort | head -n 5
# En ordre, voici ce que la commande fait
# 1. Liste les fichiers
# 2. Ne garde que les fichiers contenant .txt dans leur nom
# 3. Trie selon l'ordre alphabetique
# 4. Ne conserve que les 5 premiers resultats

# Ne conserve que les lignes contenant le texte "warning"
# remplace "warning" par "error", puis ecrit le resultat dans output.txt
cat data.txt | grep "warning" | sed 's/warning/error/g' > output.txt
# En ordre, voici ce que la commande fait
# 1. Affiche le contenu de data.txt
# 2. Filtre les lignes afin de ne conserver que les lignes contenant "warning"
# 3. Applique un regex pour remplacer "warning" par "error"
# 4. > permet d'ecrire le resultat dans le fichier output.txt

# Genere un fichier de test
echo -e "1\t2\n2\t3\n3\t4\n" > foo.txt
# En ordre, voici ce que la commande fait
# 1. Affiche une chaine de caractere formatee
# (l'argument -e fait en sorte que \t sera interprete comme une tabulation, \n comme un
    ↪ retour a la ligne)
# 2. Ecrit le resultat dans foo.txt

# Calcule la somme de chacune des colones
cat foo.txt | awk '{sum += $1; sum2 += $2} END {print sum; print sum2}' \
    | xargs echo "Sum of both columns"
# En ordre, voici ce que la commande fait
# 1. Affiche le contenu de foo.txt
# 2. Awk
# a. Cree une variable sum, a laquelle on ajoute la valeur
# de la premiere colonne ($1) pour chaque ligne
# b. Cree une variable sum2 a laquelle on ajoute la valeur
# de la deuxieme colonne ($2) pour chaque ligne.
# c. Affiche les valeurs de sum et sum2
# d. \ permet de continuer la commande sur la ligne suivante
# 3. Passe le resultat de awk a la commande echo qui va afficher la somme
```

On vous recommande d'essayer ces commandes une à la fois afin de bien comprendre chaque étape du pipeline, par exemple :

```
ls -l
ls -l | grep ".txt"
ls -l | grep ".txt" | sort |
ls -l | grep ".txt" | sort | head -n 5
```

Après cet interlude sur les commandes Bash et le piping, il est temps de régler le problème du serveur.

Exercice 12: Piping et redirection

Utilisez le piping et la redirection pour réécrire les exercices 9, 10 et 11 en une seule commande.

Écrivez le résultat de votre script dans le fichier `~/out_12.txt` (avec une redirection).

Exécutez la commande de correction `correction_12.sh` pour valider votre réponse.

Solution: 12

```
grep Error messages.txt | uniq | sed 's/Error \(4[0-9]\+\)/Warning \1/g' > ~/out_12.txt
```

Les messages d'erreur semblent un problème avec un fichier trop gros, utilisez vos connaissances en Bash pour trouver les plus gros fichiers dans le répertoire `files`.

Exercice 13: Taille de fichiers

Afin de trouver les cinq plus gros fichiers du dossier `Documents`, utilisez cette commande qui retourne la liste des fichiers ainsi que leur taille.

```
find Documents -type f | xargs du -sb
```

Chaque ligne contient la taille en octets et le nom du fichier séparé par un caractère `TAB`.

Utilisez la sortie de cette commande afin de trouver les cinq fichiers les plus volumineux. Votre script doit retourner cinq lignes dans le même format que `du`, c'est-à-dire que chaque ligne doit avoir le format suivant :

```
taille-en-octets nom-du-fichier
```

Écrivez le résultat de votre script dans le fichier `~/out_13.txt` (avec une redirection).

Exécutez la commande de correction `correction_13.sh` pour valider votre réponse.

Solution: 13

```
find ~/Documents -type f | xargs du -sb | sort -rh | head -n 5 > ~/out_13.txt
# ou
find ~/Documents -type f | xargs du -sb | sort -h | tac | head -n 5 > ~/out_13.txt
# ou
find ~/Documents -type f | xargs du -sb | sort -h | tail -n 5 | tac > ~/out_13.txt
# ou
find ~/Documents -type f -exec du -sb {} + | sort -rh | head -n 5 > ~/out_13.txt
```

Exercice 14: Analyse de données

Le fichier ~/ApplicationData/db.tsv contient des données sous la forme de TSV (*TAB separated value*). Voici le format du fichier :

id	date	name	type	size
123	2023-12-25	foo	error_log	23

Utilisez ce fichier afin de trouver les 10 lignes avec la plus petite taille (colonne size), ne conserver que la colonne nom. Vous devez conserver l'entête de la colonne conservée, c'est-à-dire la première ligne de ~/out_14.txt devrait être name.

Notez bien: Afin de réaliser cette tâche, vous pouvez utiliser awk, un langage de programmation spécialisé pour la manipulation de texte. Par exemple, pour extraire la 1e et la 3e colonne, séparé par un TAB, vous pouvez utiliser la commande.

```
awk '{print $1 "\t" $3}' file.txt
```

Adaptez cette ligne afin de résoudre le problème.

Écrivez le résultat de votre script dans le fichier ~/out_14.txt (avec une redirection).

Exécutez la commande de correction correction_14.sh pour valider votre réponse.

Solution: 14

```
awk '{print $5 " " $3}' db.tsv | sort -h | head -n 10 | awk '{print $2}' > ~/out_14.txt
# ou
awk '{print $5 " " $3}' db.tsv | sort -h | head -n 10 | sed 's/^[^ ]* //' > ~/out_14.txt
```

4.4 Scripting

Maintenant que vous êtes plus à l'aise avec Bash, il est temps de créer des scripts plus avancés.

NOTEZ BIEN:

```bash
#!/usr/bin/env bash
# Les arguments du script sont disponibles avec $n (ou n est un nombre naturel)
echo "The script name is: $0"
echo "The first argument is: $1"
echo "The second argument is: $2"
echo "All arguments:$@"
# {@:2:3} veut dire prendre une slice commencant a 2 (1-based) de longueur 3
echo "Arguments from 2 to 4: ${@:2:3}"

# Variables, noter qu'il n'y a pas d'espace autour du signe =
name="John"
age=25
# $age permet de remplacer la valeur de la variable dans la chaine de caractere entre "
echo "My name is $name and I am $age years old."
# Variables d'environnement
echo "The value of HOME is: $HOME"

# Substitution
current_directory=$(pwd)
# Les substitutions fonctionnent seulement pour des double quotes "
echo "The current directory is: $current_directory using double quotes"
# Pas pour des single quotes '
echo 'The current directory is: $current_directory using single quotes'
# Il est aussi possible d'appeler une commande directement dans une chaine de caractere
path="$(pwd)"/my/path
echo "Path: $path"

# Conditionnels
if [ "$age" -ge 18 ]; then
  echo "You are an adult."
else
  echo "You are a minor."
fi

# Boucles
for i in 1 2 3 4 5
do
    echo $i
done

fruits=("apple" "banana" "orange")
for fruit in "${fruits[@]}"
do
    echo "I like $fruit"
done

counter=1
while [ $counter -le 5 ]
do
    echo $counter
    ((counter++))
done

# Function
greet() {
  echo "Hello, $1!"
}

greet "Alice"
```

Finalement, après tout ce temps à déboguer, vous décidez qu'il serait plus sage de revenir à une version précédente. Pour ce faire, vous décidez d'écrire un script qui va essayer différentes version du projet, en commençant

par la plus récente, et en reculant jusqu'à ce que le script de test passe. Ce script sera réalisé en plusieurs étapes.

Exercice 15: Script de test

Complétez le script suivant afin de tester que le serveur Python est fonctionnel. Sauvegardez ce script dans le fichier ~/`test.sh`, et donnez-lui les permissions d'exécution.

```
#!/usr/bin/env bash

cd ~/server

# Lance le serveur en arriere plan
uvicorn main:app &
# Sauvegarde le process ID (pid) du serveur avec $!
server_pid=$!

# La fonction ‘stop_server‘ arrete le processus du serveur
stop_server() {
  kill $server_pid
}

# Enregistre la fonction ‘stop_server‘ qui sera appelee quand ce script terminera
trap stop_server EXIT

# Attendre que le server soit pret
sleep 1

# TODO faire une requete HTTP a l'adresse localhost:8000

# TODO Si le resultat est une erreur, quitte avec un code 1 avec la commande ‘exit 1‘
# $? contient le code de retour de la derniere commande
```

Exécutez la commande de correction `correction_15.sh` pour valider votre réponse.

Solution: 15

```
#!/usr/bin/env bash

cd ~/server

# Lance le serveur en arriere plan
uvicorn main:app &
# Sauvegarde le process ID (pid) du serveur avec $!
server_pid=$!

# La fonction ‘stop_server‘ arrete le processus du serveur
stop_server() {
  kill $server_pid
}

# Enregistre la fonction ‘stop_server‘ qui sera appelee quand ce script terminera
trap stop_server EXIT

# Attendre que le server soit pret
sleep 1

# TODO faire une requete HTTP a l’adresse localhost:8080
curl localhost:8000 > /dev/null 2> /dev/null

# TODO Si le resultat est une erreur, quitte avec un code 1 avec la commande ‘exit 1‘
# $? contient le code de retour de la derniere commande
if [ $? -eq 0 ]; then
  echo "Server is up"
else
  echo "Server is down"
  exit 1
fi
```

Exercice 16: Script de retour de version

Complétez le script suivant afin de retourner à une version précédente du serveur jusqu'à ce que le script de test passe. Sauvegardez ce script dans le fichier ∼/`revert.sh`, et donnez lui les permissions d'exécution.

```
#!/usr/bin/env bash

cd ~/server

revert_last_commit() {
  git reset HEAD~1
}

exit_status=1
# TODO faire une boucle while tant que $exit_status n’est pas egal a 0
# A chaque iteration, executez le script test.sh
# Si le script termine avec un code 0, arreter le script exit 0
# Sinon, appelez la fonction ‘revert_last_commit‘
```

Exécutez votre script.
Exécutez la commande de correction `correction_16.sh` pour valider votre réponse.

Solution: 16

```
#!/usr/bin/env bash

cd ~/server

revert_last_commit() {
  git reset HEAD~1
}

exit_status=1
while [ $exit_status -ne 0 ]; do
  # Execute le script de test
  ~/test.sh > /dev/null 2>&1
  # Sauvegarde le status du script de test
  exit_status=$?

  if [ $exit_status -ne 0 ]; then
    echo "The script returned a non-zero exit status ($exit_status). Reverting commit and
        ↪ retrying..."
    revert_last_commit
  else
    echo "The script returned 0 (success)."
    exit 0
  fi

  # Add a delay between retries (optional)
  sleep 1
done
```

4.5 Extras

4.5.1 .bashrc

`.bashrc` est un fichier exécuté lors de l'ouverture d'un terminal. Voici quelques utilisations courantes :

- Configuration par utilisateur de la Shell ;
- Définir des alias (le sujet de la prochaine section) ;
- Configurer des variables d'environnement (par exemple modifier la variable `PATH`, sujet d'une section extra) ;
- Configuration du prompt (ce qui est affiché dans la Shell `glo2001@server:~$`) grâce à la variable `PS1`, par exemple pour afficher le nom de la branche Git ;
- Définir des fonctions personnalisées.

4.5.2 Alias

Il est possible de définir des alias pour des commandes communes. Ces commandes sont souvent ajoutées au `.bashrc`. Voici quelques exemples d'alias :

```
# Affiche les permissions
alias ll='ls -alF'
# Affiche les fichier caches (commencant par un point)
alias la='ls -A'

# Raccourcits pour apt
alias update='sudo apt update'
alias upgrade='sudo apt upgrade'
alias install='sudo apt install'

# Git aliases
alias ga='git add'
alias gs='git status'
alias gc='git commit'
alias gp='git push'
```

```
# Raccourcits vers des dossiers commins
alias docs='cd ~/Documents'
alias dl='cd ~/Downloads'
alias desk='cd ~/Desktop'

# Une petite blague
alias emacs='vim'
```

4.5.3 PATH

La variable `PATH` est une variable d'environnement utilisée par le Shell pour déterminer les répertoires dans lesquels il recherche les exécutables lorsque vous tapez une commande dans le terminal. Lorsque vous entrez une commande, le Shell va parcourir chaque répertoire spécifié dans la variable `PATH` de gauche à droite, cherchant un fichier exécutable portant le nom de la commande. Le premier fichier exécutable trouvé est alors exécuté.

La valeur de la variable `PATH` est une liste de répertoire, séparés par des deux-points (`:`). Ajouter un chemin à cette variable permet d'accéder rapidement à des commandes sans devoir spécifier le chemine complet de chaque exécutable. Par exemple, pour ajouter un répertoire à la variable `PATH` :

```
export PATH=$PATH:~/.bin
```

4.5.4 Alias Git

Les alias Git sont des raccourcis personnalisés que vous pouvez créer pour simplifier l'utilisation des commandes Git fréquemment utilisées. Les alias vous permettent de définir des commandes abrégées qui exécutent une séquence de commandes Git plus longue ou complexe. Vous pouvez ajouter des commandes dans le fichier `.gitconfig` qui se trouve dans votre dossier `home`. Voici un exemple d'alias Git, extrait de willGuimont/.gitconfig.

```
[alias]
	lg1 = log --graph --abbrev-commit --decorate --format=format:'%C(bold blue)%h%C(
	    ↪ reset) - %C(bold green)(%ar)%C(reset) %C(white)%s%C(reset) %C(dim white)- %an%
	    ↪ C(reset)%C(bold yellow)%d%C(reset)' --all
	lg2 = log --graph --abbrev-commit --decorate --format=format:'%C(bold blue)%h%C(
	    ↪ reset) - %C(bold cyan)%aD%C(reset) %C(bold green)(%ar)%C(reset)%C(bold yellow)
	    ↪ %d%C(reset)%n'' %C(white)%s%C(reset) %C(dim white)- %an%C(reset)' --all
	lg = !"git lg1"
	st = status
	co = commit
	com = commit -m
	coam = commit --amend
	a = add
	aa = add .
	p = push
	pu = push -u origin HEAD
	cb = checkout -b
	c = checkout
	m = merge
	f = fetch
	g = pull
	wip = commit -a -m "wip"
	wap = commit -a -m
	todo = !"git commit -a -m \"todo\" && git p"
	refs = !"git pull && git commit -a -m \"updated references\" && git p"
	figs = !"git pull && git commit -a -m \"updated figures\" && git p"
	wdiff = diff --word-diff-regex=.
```

5 Docker

Après votre succès avec Bash, votre patron vous a confié une nouvelle mission : trouver une solution au problème de déploiement des applications. Votre entreprise rencontre fréquemment des pannes dues à des incompatibilités entre les différentes versions des dépendances. En effet, chaque développeur peut choisir le système d'exploitation sur lequel il souhaite travailler. Par conséquent, certains développeurs utilisent différentes distributions Linux (Ubuntu, Arch, Gentoo), tandis que d'autres préfèrent Windows.

Cette situation entraîne souvent des problèmes de version, car les dépendances varient d'un système d'exploitation à l'autre. Et le pire, c'est que les versions utilisées diffèrent de celles déployées sur le serveur de production. L'excuse récurrente «Ça fonctionnait sur ma machine» lorsqu'un développeur provoque une panne sur les serveurs de production a finalement exaspéré votre patron. Il vous demande donc de trouver une solution à ce problème.

C'est ainsi que vous découvrez Docker, une plateforme qui vous permet de créer, déployer et exécuter des applications dans des conteneurs légers et isolés. Grâce à Docker, vous pouvez créer un conteneur spécifique pour chaque microservice, en incluant toutes les bibliothèques et dépendances nécessaires. Cette approche garantit que chaque application fonctionnera de manière cohérente, indépendamment des versions des bibliothèques utilisées par les développeurs individuels. De plus, Docker permet de reproduire l'environnement de développement sur le serveur de production, éliminant ainsi les problèmes liés aux différences d'environnement.

Avec Docker, vous allez pouvoir relever le défi consistant à exécuter les trois principaux microservices de votre entreprise. Voici les microservices que vous devrez faire fonctionner :

1. un application de surveillance de la santé des serveurs web, écrite en Haskell ;
2. une API de gestion de documents, écrite en Rust, nécessitant une base de données PostgreSQL ; et
3. une application en Python permettant de visualiser la santé des serveurs web.

5.1 Pourquoi la conteneurisation ?

Isolation L'un des principaux avantages de la conteneurisation est l'isolation. Il arrive souvent que différentes applications nécessitent des versions spécifiques de dépendances. Avec Docker, il est possible d'installer différentes versions de dépendances dans des conteneurs distincts, sans risquer de conflits ou de compromettre le fonctionnement des autres applications. De plus, si vous devez exécuter des applications sur différents systèmes d'exploitation, tels qu'Ubuntu et Arch Linux, vous pouvez simplement les encapsuler dans des conteneurs Docker, évitant ainsi la nécessité de réécrire les applications pour chaque système.

Portabilité La portabilité est un autre avantage clé de Docker. Une fois que vous avez configuré un environnement de développement avec toutes les dépendances nécessaires, il devient fastidieux de reproduire cette configuration sur d'autres machines ou pour de nouveaux membres de l'équipe. Grâce à Docker, vous pouvez créer une image contenant toutes les dépendances et configurations requises, qui peut être facilement distribuée et exécutée sur différentes plateformes, qu'il s'agisse de systèmes Linux, Windows ou autres. Cela permet à un nouveau membre de l'équipe de démarrer rapidement sans passer des semaines à configurer son environnement. Et c'est aussi un avantage lors du déploiement des applications sur un serveur.

Reproductibilité La reproductibilité est également simplifiée grâce à Docker. Les images Docker sont versionnées, ce qui signifie qu'il est facile de reproduire les builds à l'identique, en garantissant que les environnements de développement, de test et de production sont cohérents. Il suffit de spécifier la version de l'image Docker utilisée pour garantir la cohérence et éviter les problèmes liés aux variations entre les environnements.

Efficacité En termes d'efficacité, Docker présente un avantage significatif par rapport aux machines virtuelles (VM). Contrairement aux VM, Docker n'a pas besoin d'exécuter un système d'exploitation complet pour chaque conteneur, ce qui réduit considérablement la consommation de ressources. Les conteneurs Docker partagent le noyau (*kernel*) de l'hôte, ce qui les rend légers et rapides à démarrer.

Communauté La communauté autour de Docker est très active et propose une multitude d'images déjà configurées pour divers outils et technologies populaires tels que Node.js, Python, et bien d'autres. Ces images préconfigurées facilitent le déploiement et l'utilisation de ces technologies, permettant aux développeurs de gagner du temps en évitant de configurer manuellement chaque environnement.

Industrie Enfin, il est important de souligner que Docker est devenu un standard de facto dans l'industrie du développement logiciel. De nombreuses entreprises utilisent Docker pour le développement, les tests et le déploiement de leurs applications. Docker est dorénavent un outil essentiel pour les développeurs logiciels.

5.2 Qu'est-ce que la conteneurisation ?

Docker est une plateforme logicielle open source qui permet de créer, déployer et exécuter des applications dans des conteneurs légers et isolés. Un conteneur Docker est une unité d'exécution qui encapsule une application ainsi que tous ses éléments nécessaires, tels que les bibliothèques, les dépendances et les fichiers de configuration. Ces conteneurs sont autonomes et portables, ce qui signifie qu'ils peuvent être exécutés de manière cohérente sur différents systèmes, qu'il s'agisse d'un environnement de développement, de test ou de production.

Docker est similaire à une machine virtuelle qui isole une application du système hôte. Toutefois, Docker est beaucoup plus efficace qu'une machine virtuelle, qui demande un système d'opération complet pour chaque instance. Docker virtualise partage plutôt le noyau (`kernel`) du système d'opération hôte, ce qui permet d'économiser des ressources et de rendre les conteneurs plus rapides à démarrer et à exécuter.

Voici un peu de terminologie sur Docker. Vous n'êtes pas obligé de lire les sections avancées pour réaliser cet atelier. En supplément, vous pouvez visionner la vidéo «Never install locally»[2].

Image Docker Une image Docker est un modèle ou un plan de construction qui contient tous les éléments nécessaires pour exécuter une application. Elle comprend le système d'exploitation, les bibliothèques, les dépendances, le code source de l'application et les fichiers de configuration. Les images Docker sont créées à partir de fichiers appelés `Dockerfile` qui spécifient les étapes pour construire l'image.

Dockerfile Un Dockerfile est un fichier texte qui contient les instructions pour construire une image Docker. Il spécifie les couches de l'image, les dépendances à installer, les fichiers à inclure et les commandes à exécuter lors de la construction de l'image.

Conteneur Docker Un conteneur Docker est une instance en cours d'exécution d'une image Docker. Il s'agit d'un environnement isolé qui exécute l'application avec ses dépendances. Les conteneurs sont légers, portables et autonomes, ce qui permet de les déployer facilement sur différentes machines.

Registre Docker Un registre Docker est un référentiel centralisé qui stocke et gère les images Docker. Le registre public par défaut est Docker Hub, où vous pouvez trouver de nombreuses images prêtes à l'emploi. Vous pouvez également créer et utiliser votre propre registre privé pour stocker vos propres images.

***Union filesystem* (avancé)** L'*Union Filesystem*, également connu sous le nom de *UnionFS* ou *OverlayFS*, est une technologie utilisée par Docker pour gérer les images et les couches de conteneurs de manière efficace. L'*Union Filesystem* permet de superposer plusieurs systèmes de fichiers en une seule vue logique, sans les fusionner physiquement. Cela signifie que les images Docker et les conteneurs peuvent partager et réutiliser des couches de fichiers communs, ce qui permet d'économiser de l'espace de stockage. Cela permet aussi d'accélérer la construction des images en utilisant une cache des couches déjà construites. Lorsqu'un conteneur est démarré, une nouvelle couche en lecture-écriture est ajoutée au-dessus des couches d'image, permettant ainsi les modifications spécifiques à ce conteneur sans affecter les autres.

***cgroups* (avancé)** Les *cgroups*, ou *control groups*, sont une fonctionnalité du noyau Linux utilisée par Docker pour limiter et gérer les ressources système utilisées par les conteneurs. Les *cgroups* permettent de contrôler les ressources telles que le processeur, la mémoire, la bande-passante du disque et le réseau, afin de garantir une utilisation équilibrée et équitable des ressources système entre les conteneurs. Docker utilise les *cgroups* pour définir des limites et des quotas sur les ressources allouées à chaque conteneur, assurant ainsi une isolation et une performance prévisibles.

***Namespaces* (avancé)** Les namespaces sont une fonctionnalité du noyau Linux qui permet d'isoler les ressources système entre les processus. Docker utilise plusieurs types de namespaces pour fournir une isolation entre les conteneurs, notamment le namespace PID (isolation des processus), le namespace réseau (isolation du réseau), le namespace utilisateur (isolation des utilisateurs) et le namespace de montage (isolation des points de montage). Ces namespaces garantissent que chaque conteneur a sa propre vue isolée du système, ce qui empêche les processus d'un conteneur d'interférer avec d'autres conteneurs ou le système hôte.

***Chroot jail* (avancé)** Chroot, ou change root, est une fonctionnalité Unix/Linux qui permet de changer le répertoire racine d'un processus et de limiter son accès au système de fichiers. Docker utilise la fonction chroot pour créer un environnement isolé à l'intérieur du conteneur, où le répertoire racine du conteneur devient le nouveau point de départ pour tous les chemins de fichiers. Cela limite l'accès du conteneur aux fichiers et répertoires en dehors de son environnement isolé, renforçant ainsi la sécurité et l'isolation.

5.3 Conteneurs manuellement

Afin de bien illustrer ce que fait Docker, nous allons construire un conteneur simplifié manuellement sans l'aider de Docker. Cette section est inspirée par [4]. Dans un terminal, entrez les commandes suivantes :

```
# On valide que neofetch n'est pas installe
# La commande suivante devrait lancer une erreur, ne pas installer le package
neofetch

# Dossier pour les conteneurs
mkdir ~/containers; cd ~/containers

# Construit le systeme de fichier du conteneur
sudo apt update; sudo apt install -y debootstrap
sudo debootstrap jammy ./ubuntu-container http://archive.ubuntu.com/ubuntu/

# Cree un environnement isole (namespace)
sudo unshare --uts --pid --mount --ipc --fork

# Monter les dossiers de processus, systeme, les devices et les ppa
mount -t proc none ./ubuntu-container/proc/
mount -t sysfs none ./ubuntu-container/sys
mount -o bind /dev ./ubuntu-container/dev
mount -o bind /tmp ./ubuntu-container/tmp/
cp /etc/apt/sources.list ./ubuntu-container/etc/apt/sources.list

# On utilise chroot pour lancer une Shell dans le conteneur
chroot ./ubuntu-container/ /bin/bash
# et voila, on est dans un conteneur isole

# On installe neofetch
apt update
apt install -y neofetch

# On teste neofetch, la commande fonctionne!
neofetch

# On quitte le conteneur
exit

# On quitte le unshare
exit

# Si on essait neofetch a nouveau, la commande n'est pas trouvee.
# On a donc bien isole le conteneur!
neofetch
```

Ainsi, comme vous pouvez le voir, il n'y a pas de magie dans les conteneurs. On utilise des outils de base de Linux afin d'isoler un processus dans son propre système de fichiers. Voici une autre ressource intéressante sur les dessous de Docker : p8952/bocker: Docker implemented in around 100 lines of bash.

5.4 Les bases de Docker

Exercice 17: Exécution d'une image déjà faite

Exécutez l'image Docker `hello-world`.

Notez bien: Pour exécutez une image Docker, utilisez la commande `run`.

```
docker run nom-image
```

Si la commande fonctionne, vous verrez un message s'afficher.

Solution: 17

```
docker run hello-world
```

Notez bien: Pour exécuter une commande dans le conteneur, il suffit de spécifier la commande à la fin de la commande `run`. Dans les exemples suivants, on passe la commande `bash -c 'echo "Hello world"'` qui interprète la chaîne de caractère dans l'interpréteur Bash.

```
# Affiche 'Hello world' dans le container
docker run ubuntu bash -c 'echo "Hello world"'
# Affiche les informations du systeme hote
cat /etc/os-release
# Affiche les informations du conteneur
docker run archlinux bash -c 'cat /etc/os-release'
```

Notez bien: Pour lancer une commande intéractive (comme une Shell), utilisez l'argument `-it`. L'argument `--rm` permet de supprimer le conteneur une fois qu'il termine.

```
docker run -it --rm ubuntu bash
```

Exercice 18: Gestion de conteneurs

Notez bien: Pour voir les conteneurs en cours d'exécution `docker ps`.
Pour arrêter un conteneur `docker stop container_id`.

- Lancer un terminal interactif avec Docker ;
- Ouvrir un autre terminal, et inspecter les conteneurs en cours d'exécution ;
- Arrêter le conteneur Docker depuis le second terminal.

Solution: 18

```
# Dans le premier terminal
docker run -it ubuntu bash

# Dans le second terminal
docker ps
docker stop id
# ou
docker rm -f id
```

Notez bien: Afin de configurer des variables d'environnement dans un conteneur, utilisez l'argument `-e`.

```
docker run -it --rm -e HELLO=hello ubuntu sh -c 'echo $HELLO'
```

Notez bien: Afin d'ouvrir un port réseau, utilisez l'argument `-p docker:host`, où `docker` est le numéro du port dans le conteneur et `host` est le numéro de port de l'hôte.

```
docker run -p 127.0.0.1:8080:80 nginx
# On peut acceder au port a partir de l'url http://localhost:8080/
curl http://localhost:8080/
```

Exercice 19: Lancez une base de données PostgreSQL avec Docker.

- Le nom de l'image est `postgres` ;
- Configurez une variable d'environnement `POSTGRES_PASSWORD=postgres` ; et
- Redirigez le port 5432 du Docker vers le port 5432 de la machine hôte ;
- Laisser ce conteneur en cours d'exécution.

Solution: 19

```
docker run -e POSTGRES_PASSWORD=postgres -p 5432:5432 --rm postgres
# ou
docker run --name postgres -e POSTGRES_PASSWORD=postgres -p 5432:5432 --rm -d postgres
```

5.5 Dockerfile

Les Dockerfiles sont des fichiers de configuration utilisés pour créer des images Docker personnalisées. Ils permettent de définir de manière reproductible et automatisée l'environnement d'exécution d'une application à l'intérieur d'un conteneur Docker.

Un Dockerfile contient une série d'instructions qui spécifient les étapes nécessaires à la construction d'une image Docker. Ces instructions incluent des actions telles que la sélection de l'image de base, l'installation de dépendances, la configuration de variables d'environnement, la copie de fichiers, l'exécution de commandes et bien plus encore.

Voici un exemple annoté de Dockerfile pour une application Python.

```
# Utilise l'image de base Python 3.9 slim
FROM python:3.9-slim

# Definit le repertoire de travail a l'interieur du conteneur
# Specifie le repertoire pour les commandes RUN, CMD, ENTRYPOINT, COPY et ADD
WORKDIR /app

# Copie le fichier requirements.txt depuis la machine hote vers le conteneur Docker
COPY requirements.txt .

# Execute une commande Bash afin d'installer les dependances Python
RUN pip install -r requirements.txt

# Copie le code source dans le conteneur Docker
COPY . .

# Definit la commande par defaut lorsque le conteneur demarre
CMD ["python", "app.py"]
```

Pour construire l'image, on utilise la commande `docker build -t <tag> .`. L'argument `-t <tag>` permet de donner le nom `<tag>` à l'image créee. Par exemple, pour l'application précédente, on pourrait utiliser "`docker build -t python-app .`". Il est à noter que l'ordre des instructions est important.

Il est crucial de bien définir l'ordre des opérations dans un Dockerfile en raison de la façon dont Docker effectue le processus de construction de l'image. Chaque instruction dans le Dockerfile crée une nouvelle couche

dans l'image Docker, et l'ordre des opérations peut avoir un impact significatif sur l'efficacité et les performances de la construction de l'image.

Docker utilise un système de cache pour accélérer le processus de construction des images. Lorsque vous exécutez une instruction, Docker vérifie si cette instruction a déjà été exécutée dans une couche précédente. Si c'est le cas et que les paramètres sont identiques, Docker réutilise la couche en cache au lieu de la reconstruire. Cela permet d'économiser du temps de construction. Cependant, si vous modifiez une instruction plus haut dans le Dockerfile, toutes les instructions suivantes seront invalidées dans le cache et devront être reconstruites. Par exemple, si vous effectuez des opérations lourdes en termes de ressources, telles que la compilation de code qui changera à chaque fois que vous modifiez le code, il peut être préférable de les placer vers la fin du Dockerfile, afin de profiter du cache autant que possible.

Exercice 20: Utilisez le `README.md` du dossier `~/Applications/status-checker` afin d'écrire un Dockerfile permettant d'exécuter l'application Haskell.

Voici ce que votre Dockerfile devra faire :

- Utilisez l'image de base `haskell:9.0-buster` ;
- Exécutez `stack setup --install-ghc` ;
- Choissiez le dosser `/app` comme *workdir* ;
- Copiez le code source dans le dossier `/app` du conteneur ;
- Exécutez `stack build` ;
- Lancez le serveur avec `stack run`.

Ensuite, lancer le Docker en exposant le port 8080. Si tout est fonctionnel, vous devriez être capable d'exécuter la requête suivante :

```
curl http://localhost:8080/endpoints
# Reponse attendue
{}%
# Ou
{}
```

Laisser ce conteneur en cours d'exécution.

Solution: 20 Dockerfile :

```
FROM haskell:9.0-buster
RUN stack setup --install-ghc
WORKDIR /app
COPY . .
RUN stack build
CMD ["stack", "run"]
```

Exécution :

```
docker build -t status-checker .
docker run --rm -p 8080:8080 status-checker
```

Exercice 21: Utilisez le `README.md` du dossier `~/Applications/python_app` afin d'écrire un Dockerfile permettant d'exécuter l'application Python.
Ensuite, lancez l'application en mode interactif et avec le bon mode réseau.

> **Notez bien:** Cette application aura besoin d'accéder au réseau local de votre ordinateur pour faire des requêtes au conteneur de l'exercice précédent, pour ce faire passer l'argument `--network="host"` lorsque vous aller exécuter le conteneur.

Si tout est fonctionnel, vous devriez être capable d'utiliser l'application afin d'ajouter des URL à monitorer et de voir les résultats.

Solution: 21 Dockerfile :

```
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
CMD ["python", "main.py"]
```

Exécution :

```
docker build -t python_app .
docker run --rm -it --network="host" python_app
```

5.6 Docker avancé

5.6.1 *Construction d'image multi étape*

Les builds multi étapes (multi-stage builds) sont une fonctionnalité avancée de Docker qui permet de créer des images Docker de manière optimisée en utilisant plusieurs étapes distinctes dans le processus de construction. Cela permet de séparer les étapes de construction et de production, ce qui peut conduire à des images finales plus légères et plus sécurisées en minimisant ce qui reste dans l'image finale.

Par exemple, pour une application React en JavaScript, on peut séparer la construction en deux étapes.

Création de l'application :

```
npx create-react-app my-app
```

Dockerfile :

```
# Etape 1: Construire l'application

# Ici on utilise 'as build' pour nommer cette etape de construction
FROM node:14-alpine as build
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

# Etape 2: Preparer le serveur
# Ici, on commence une nouvelle image de zero
FROM nginx:alpine
# On copie l'application compilee depuis l'etape precedente
COPY --from=build /app/build /usr/share/nginx/html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

Pour exécuter :

```
docker build -t web-app .
docker run --rm -p 80:80 web-app
```

Exercice 22: Utilisez le `README.md` du dossier `~/Applications/rust_api` afin d'écrire un Dockerfile permettant d'exécuter l'application Python.

- Lancer la base de données PostgreSQL d'un exercice précédent ;
- Faites une première étape pour compiler le code Rust ;
- Faites une deuxième étape pour exécuter le code ;
  - Démarrer cette image à partir de `debian:bulleye-slim` ;
  - Installer le package `libpq-dev` avec la commande `apt update; apt install -y libpq-dev` ;
  - Assigner la variable d'environnement `DATABASE_URL=postgres://postgres:postgres@localhost`
  - Copiez `/app/target/release/rust_api` depuis l'étape précédente ;
- Lancer le conteneur en spécifiant `--network="host"` et exposer le bon port ;
- Tester que le tout fonctionne avec `curl http://localhost:8081/documents`.

Solution: 22

```
FROM rust:latest AS build
WORKDIR /app
COPY . .
RUN cargo build --release

FROM debian:bullseye-slim
RUN apt update; apt install -y libpq-dev
WORKDIR /app
COPY --from=build /app/target/release/rust_api ./rust_api
ENV DATABASE_URL=postgres://postgres:postgres@db
CMD ./rust_api
```

Exécution

```
docker run -e POSTGRES_PASSWORD=postgres -p 5432:5432 postgres
docker build -t rust_api .
docker run --rm --network="host" rust_api

# Valider que le serveur fonctionne
curl http://localhost:8081/documents
```

5.6.2 Volumes

En Docker, les volumes sont utilisés pour permettre aux conteneurs d'accéder, de partager et de persister des données entre le système hôte et le conteneur lui-même. Les volumes Docker permettent de stocker des données en dehors du cycle de vie des conteneurs. Cela signifie que même si vous détruisez ou recréez un conteneur, les données stockées dans le volume restent intactes. Cela permet de séparer la persistance des données de l'environnement du conteneur, offrant ainsi une meilleure gestion des données. Les volumes sont également utiles pour donner accès au conteneur à des données trop volumineuses pour être copiées à l'intérieur de l'image, comme un jeu de données d'entraînement d'un système d'apprentissage machine.

Par exemple, pour stocker les données de la base de données PostgreSQL dans un dossier de la machine hôte, on spécifie la variable `PGDATA` et on monte un volume :

```
docker run -e POSTGRES\_PASSWORD=postgres -p 5432:5432 --rm \
    -e PGDATA=/var/lib/postgresql/data/pgdata \
    -v custom/mount/path:/var/lib/postgresql/data \
    postgres
```

Afin éviter de devoir reconstruire l'image Docker après chaque modification du code source, et ainsi accélérer le développement, il est possible de rendre accessible le code source du projet à l'aide d'un volume. Ainsi, un DockerfileDev pour l'application Python pourrait ressembler à ceci :

```
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
# On a enleve la ligne COPY . .
CMD ["bash"]
```

Exécuter avec :

```
docker build -t python_app -f DockerfileDev .
docker run --rm -v .:/app my-app
# Dans le conteneur
python main.py
# On modifie le code source
python main.py
# Les changements se repercutent dans le conteneur
```

Ainsi, on peut réutiliser la même image même si le code source a changé.

5.7 Extras

5.7.1 Nettoyage

Lorsque vous utilisez Docker, il est important de prendre en compte l'espace de stockage utilisé par les images, les conteneurs, les volumes et autres artefacts Docker. Une mauvaise gestion de l'espace de stockage peut entraîner une utilisation excessive de l'espace disque et rendre difficiles la maintenance et la gestion des ressources Docker. Docker propose plusieurs commandes de `prune` afin de supprimer les conteneurs et les images non utilisées. Pour faire le ménage des différents artefacts Docker, la commande `docker system prune -a -f` supprime toutes les ressources inutilisées.

5.7.2 Docker Compose

Docker Compose est un outil qui permet de définir et de gérer facilement des applications multi conteneurs. Il simplifie le déploiement et l'orchestration des conteneurs Docker en utilisant un fichier de configuration simple et lisible.

Avec Docker Compose, vous pouvez spécifier les services, les réseaux, les volumes et autres configurations nécessaires pour exécuter une application composée de plusieurs conteneurs. Vous pouvez également définir les dépendances entre les conteneurs, les variables d'environnement, les ports exposés, etc.

Docker Compose utilise un fichier de configuration YAML pour décrire l'infrastructure de l'application. Ce fichier contient des sections telles que *services*, *networks*, *volumes*, etc., où vous pouvez définir les différentes parties de votre application et leurs configurations.

Voici un exemple de `docker-compose.yml` pour l'application Rust qui lance à la fois le server et la base de données PostgreSQL :

```
# Liste des conteneurs que l'on va lancer
services:
  # Conteur de la base de donnees
  # On retrouve les memes parametres que dans la commande ‘docker run‘
  db:
    # On specifie l'image que l'on veut lancer
    image: postgres:latest
    restart: always
    environment:
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=postgres
      - PGDATA=/var/lib/postgresql/data/pgdata
    expose:
      - 5432
    volumes:
      - db:/var/lib/postgresql/data
  # Serveur Rust
  api:
    # Va construire le Dockerfile du repertoire ou se trouve le docker-compose.yml
    build: .
    # Construit le service db avant api
    depends_on:
      - db
    ports:
      - 8081:8081
    # L'URL de la base de donnee change, plutot que @localhost
    # docker-compose rend db accessible sous le nom @db
    environment:
      - DATABASE_URL=postgres://postgres:postgres@db
    # On attend que la base de donnes soit prete avant de lancer le serveur
    entrypoint: bash -c "sleep 5 && ./rust_api"
# Necessaire pour le volume de la base de donnes
volumes:
  db:
    driver: local
```

Et on peut l'exécuter avec :

```
docker-compose up --build
```

5.7.3 Podman

Podman est une alternative open source à Docker. Docker et Podman sont deux outils populaires de conteneurisation qui partagent des fonctionnalités similaires, mais ils diffèrent dans leur architecture et leur approche de la sécurité. Docker utilise une architecture client-serveur, où le démon Docker s'exécute en tant que processus distinct et les commandes Docker sont exécutées via l'interface en ligne de commande (CLI). En revanche, Podman utilise une architecture sans démon (daemonless), ce qui signifie qu'il s'exécute directement en tant qu'utilisateur régulier et ne nécessite pas de processus démon distinct. Ainsi, Podman peut s'exécuter sans privilèges spéciaux, et ainsi limite les risques potentiels associés à l'exécution en tant que superutilisateur. [3]

Podman offre aussi des outils pour gérer des *pods*, un sujet hors de la portée de cet atelier.

5.7.4 Kubernetes

Kubernetes est un système open source d'orchestration de conteneurs qui facilite le déploiement, la gestion et la mise à l'échelle d'applications conteneurisées. L'utilisation de Kubernetes offre de nombreux avantages, tels que la tolérance aux pannes, la facilitation des déploiements, la gestion et la mise à l'échelle des applications conteneurisées.

Références

[1] Ken Cochrane, Jeeva S. Chelladhurai, and Neependra K. Khare. *Docker Cookbook : Over 100 Practical and Insightful Recipes to Build Distributed Applications with Docker, 2nd Edition*. Packt Publishing, 2nd edition, 2018. ISBN 1788626869.

[2] Coderized. Never install locally. URL https://www.youtube.com/watch?v=J0NuOlA2xDc.

[3] Red Hat. What is podman ? URL https://www.redhat.com/en/topics/containers/what-is-podman.

[4] Akash Rajpurohit. Build your own docker with linux namespaces, cgroups, and chroot : Hands-on guide. URL https://akashrajpurohit.com/blog/build-your-own-docker-with-linux-namespaces-cgroups-and-chroot-handson-guide/.

[5] Eric Steven Raymond. *The Art of Unix Programming*. Addison-Wesley, Boston, 2004. ISBN 0131429019 9780131429017. URL http://www.faqs.org/docs/artu/.