Développement et exploitation

XMS1IE080: TPs

Erwan Bousse

Version 2026-2027, 2026-10-07

Table des matières

1. ▯▯ Notice : infrastructure et outils . . . 1
   1.1. Infrastructure Nantes Université . . . 1
   1.2. Outils essentiels . . . 2
   1.3. Pouvoir se connecter à l'environnement de déploiement . . . 2
   1.4. Générer une clé SSH personnelle . . . 4
2. ▯▯ Notice : environnement de déploiement . . . 7
   2.1. Se connecter au serveur de déploiement . . . 7
   2.2. Utiliser le serveur de déploiement . . . 10
3. ▯▯ Notice : Git et Gitlab . . . 15
   3.1. Connexion et page d'accueil . . . 15
   3.2. Cloner le code source d'un projet Gitlab . . . 16
   3.3. Stocker ses images de conteneur dans le registre . . . 17
4. 💻 TP : Découverte de l'infrastructure et des outils . . . 21
   4.1. Présentation et objectifs . . . 21
   4.2. Usages principaux du serveur de déploiement . . . 21
   4.3. Comprendre un peu l'environnement de déploiement . . . 23
   4.4. Quelques petits déploiements (sans conteneurs) . . . 24
5. 💻 TP : Créer et manipuler des conteneurs . . . 27
   5.1. Présentation et objectifs . . . 27
   5.2. Connexion au serveur de déploiement . . . 27
   5.3. Tutoriel : un premier conteneur Ubuntu . . . 27
   5.4. Exercice : Asciidoctor depuis un conteneur Alpine . . . 31
   5.5. Tutoriel : utiliser un conteneur pour déployer `replace_string` . . . 34
   5.6. Exercice : utiliser des conteneurs pour déployer différents programmes . . . 35
   5.7. Faire le ménage sur le serveur de déploiement . . . 37
6. 💻 TP : Montage de répertoires/fichiers . . . 39
   6.1. Présentation et objectifs . . . 39
   6.2. Connexion au serveur de déploiement . . . 39
   6.3. Tutoriel : premiers montages dans un conteneur Ubuntu . . . 39
   6.4. Ménage . . . 42
   6.5. Exercice : faciliter l'usage du conteneur Asciidoc avec un montage . . . 42
   6.6. Exercice : faciliter le déploiement de `economic_dispatch` . . . 43
   6.7. Exercice : déployer le logiciel *docker-ffmpeg-converter* . . . 44
7. 💻 TP : Publication de ports réseaux . . . 47
   7.1. Tutoriel : déployer un serveur web *nginx* avec publication de port . . . 47
   7.2. Exercice : déployer un service Gokapi avec publication de port . . . 51
   7.3. Exercice : déployer un service Kanboard avec publication de port . . . 53
   7.4. Exercices additionnels . . . 55
8. 💻 TP : Conteneuriser (manuellement) un logiciel . . . 57
   8.1. Préparatifs . . . 57
   8.2. Tutoriel : une première image Ubuntu avec `cowsay` . . . 57
   8.3. Exercice : image Alpine avec Asciidoctor . . . 61
   8.4. Tutoriel : une image pour déployer un site web avec *nginx* . . . 62

8.5. Exercice : une image de conteneur pour `economic_dispatch` . . . 65
8.6. Exercice : créer une image pour *Web Calculator* (Python) . . . 65
9. 💻 TP : Scripter la conteneurisation d'un logiciel . . . 67
9.1. Présentation et objectifs . . . 67
9.2. Préparatifs . . . 67
9.3. Tutoriel : image `cowsay` via un `Containerfile` . . . 67
9.4. Exercice : image Asciidoctor via `Containerfile` . . . 69
9.5. Tutoriel : image site web *nginx* via `Containerfile` . . . 69
9.6. Exercice : créer une image pour *Web Calculator* (Python) . . . 70
9.7. Exercice : créer une image *webdav-embedded-server* (Java) . . . 72
9.8. Exercice : créer une image *WordcloudGenerator* (langage R) . . . 75
10. 💻 TP : Faire un déploiement multi-conteneurs . . . 77
10.1. Tutoriel : premiers réseaux virtuels . . . 77
10.2. Tutoriel : déployer un service Mealie à l'aide d'un réseau virtuel . . . 82
10.3. Exercice : déployer un service Kanboard+MariaDB avec un réseau virtuel . . . 86
10.4. Exercice : déployer *lldap*, *Kanboard* et *Mealie* à l'aide de réseaux virtuels . . . 88
10.5. Prise de recul sur le travail réalisé dans les séances de TP jusqu'ici . . . 91
10.6. Exercices additionnels . . . 92

Chapitre 1. Notice : infrastructure et outils

Les travaux pratiques nécessitent une certaine **diversité d'outils** ainsi qu'une petite **infrastructure** disponible depuis les salles de TP informatique.

Les différentes notices à votre disposition dans cette partie du site visent à présenter l'essentiel de ce qui sera utilisé tout au long des TPs.

Tous les TPs sont à réaliser depuis un poste informatique **Linux**.

1.1. Infrastructure Nantes Université

On peut schématiser l'infrastructure à disposition de la manière suivante :

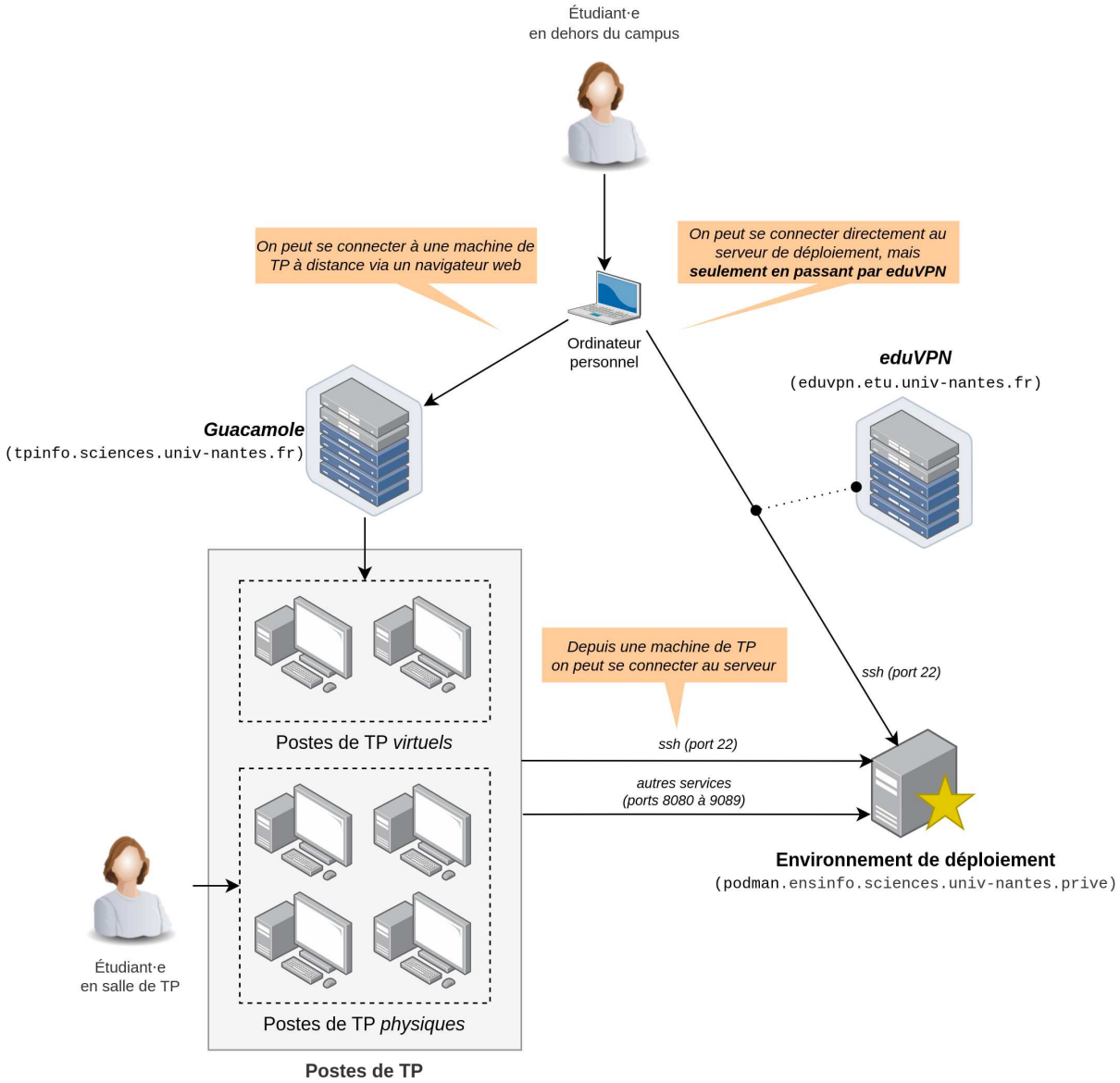


Figure 1. Infrastructure de Nantes Université pour les TPs

On y trouve :

- Tout en bas à gauche, les **postes de TP physiques** à votre disposition en salle informatique. Démarrés sous Linux, ces postes possèdent un certain nombre d'outils utilisés pour le cours

(terminal, éditeur de code, navigateur web, client ssh, etc.).

- En bas à droite, l'**environnement de déploiement**, dédié à l'enseignement de la conteneurisation. C'est un environnement d'accueil Linux puissant à disposition de tous les étudiants pour apprendre à déployer des logiciels. Voir Chapitre 2. Il possède comme nom d'hôte réseau `podman.ensinfo.sciences.univ-nantes.prive` (qui peut être simplifié en `podman` depuis une machine de TP).
- En haut à droite, le **serveur eduVPN de l'université**. Ce service vous permet de connecter votre ordinateur au réseau interne de l'université, et ainsi de pouvoir vous connecter directement à l'environnement de déploiement depuis votre ordinateur.
- En haut à gauche, le **serveur Guacamole de l'université** (accessible ici [https://tpinfo.sciences.univ-nantes.fr]). Ce service vous permet de vous connecter, y compris depuis l'extérieur du campus, à un poste de TP *virtuel* en tous points identique aux postes de TP physiques. Depuis un poste virtuel, on peut notamment se connecter au serveur de déploiement. Voir Section 1.3.3.

1.2. Outils essentiels

Les TPs seront tous réalisés depuis une machine de salle de TP informatique, et vous utiliserez au moins les outils essentiels suivants :

- Le **terminal** est l'outil privilégié utilisé tout au long du semestre. Il permet notamment d'utiliser l'utilitaire `ssh` (voir item suivant), et d'exécuter de nombreuses tâches liées à la construction des logiciels.

> **Si vous n'êtes pas à l'aise avec l'utilisation d'un terminal sous Linux**, il est fortement recommandé de vous familiariser un minimum pour les travaux pratiques. Vous pouvez par exemple regarder le tutoriel de la documentation Ubuntu [https://doc.ubuntu-fr.org/tutoriel/console_ligne_de_commande] (qui est la distribution utilisée en salle de TP) ou bien le tutoriel de la documentation Fedora [https://doc.fedora-fr.org/wiki/Debuter_avec_le_terminal] (une autre distribution Linux).

- L'utilitaire `ssh` peut être utilisé depuis un terminal, et permet de se connecter à un environnement distant à l'aide du protocole SSH. Nous utiliserons cet outil pour nous connecter sur le serveur de déploiement.
- L'utilitaire `nano` est un éditeur de texte qui peut être utilisé directement depuis un terminal. Il nous sera particulièrement utilise lorsque vous serez connecté sur le serveur de déploiement, car nous ne pourrons interagir avec ce serveur que via le terminal et `ssh`, et nous ne pourrons donc pas ouvrir d'éditeur de texte graphique dans cet environnement.
- **Visual Studio Code** (aussi installé sous le nom *VSCodium* sur les postes de salle informatique) sera l'éditeur privilégié pour éditer des fichiers.

1.3. Pouvoir se connecter à l'environnement de déploiement

Pour des raisons de de sécurité, l'environnement de déploiement n'est accessible que depuis le réseau interne de Nantes Université. Avec cette contrainte, voici les différentes manières possible de s'y connecter.

1.3.1. Option 1 : physiquement depuis une salle de TP

Il est directement possible de se connecter à distance à l'environnement de déploiement depuis une machine de TP.

1.3.2. Option 2 : depuis son ordinateur personnel, via eduVPN

Vos ordinateurs personnels ne sont jamais directement sur le réseau interne de l'université, même lorsqu'ils sont connectés au wifi *eduroam*. Il est cependant possible de les connecter au réseau interne à l'aide du VPN de l'université, appelé **eduVPN**.

Procédure à suivre :

1. Contacter l'enseignant responsable du cours (par email ou Mattermost), et lui demander d'activer votre accès eduVPN (*cet accès n'est pas actif par défaut pour tous les comptes étudiants*).
2. Installer eduVPN à partir de cette page [https://www.eduvpn.org/client-apps/].
3. Lancer eduVPN, et démarrer une connexion au serveur `eduvpn.etu.univ-nantes.fr`.
4. Une page s'ouvre dans votre navigateur web, avec laquelle vous devez vous identifier avec votre compte étudiant.

L'outil devrait alors affiché *Connected*, ce qui signifie que vous êtes désormais sur le réseau interne de l'université. Il est alors possible de se connecter au serveur **à l'aide de son nom d'hôte complet : `podman.ensinfo.sciences.univ-nantes.prive`** (et non à l'aide du nom d'hôte simplifié `podman`).

Pour travailler, il vous faudra à minima installer le client OpenSSH [https://www.openssh.com/] sur votre ordinateur (ou autre client SSH, à défaut) :

- Sous Linux, installez simplement le paquet correspondant, par exemple :
  - sous Ubuntu : `apt install openssh-client`
  - sous Fedora : `dnf install openssh-clients`
- Sous MacOS, vous aurez besoin de HomeBrew [https://brew.sh/], avec lequel vous pourrez installer le paquet nécessaire avec la commande : `brew install openssh`
- Sous Windows, il est possible de l'installer via le menu système *Optional Features*, trouvable sous le nom *OpenSSH Client*.

1.3.3. Option 3 : depuis son ordinateur personnel, via Guacamole

Le serveur Guacamole de l'université est un déploiement du logiciel Guacamole [https://guacamole.apache.org/], qui permet de se connecter à distance sur un poste de TP virtuel via un simple navigateur web. Dans le déploiement de l'université, ce logiciel permet de se connecter à une *poste de TP virtuel* qui permet de travailler sans être physiquement dans une salle de TP.

Le service est disponible à cette adresse [https://tpinfo.sciences.univ-nantes.fr]. On peut alors choisir un poste virtuel, puis se connecter à l'aide de son mot de passe université.

Une notice d'utilisation est également disponible sur le site du Service Commun Informatique (SCI) de l'université [https://intrasci.sciences.univ-nantes.fr/] : depuis le menu à gauche, se rendre dans *Pôle Enseignement*, puis *Accès à distance*, et de là se rendre à la fin de la section *TpInfo*.

Ce service est ouvert le lundi de 12h15 à 20h et du mardi au vendredi de 8H à 20H.

Même si l'affichage est un petit peu ralenti et pas toujours réactif, c'est simplement un problème d'affichage. La machine sous-jacente est elle bien rapide et pourra compiler et ou exécuter vos projets sans difficulté.

1.4. Générer une clé SSH personnelle

Ce cours fait usage du protocole SSH [https://fr.wikipedia.org/wiki/Secure_Shell] afin de se connecter à distance au serveur de déploiement (voir Chapitre 2).

Pour sécuriser une connexion, le protocole SSH peut reposer sur ce qu'on appelle une **clé SSH**, qui prend la forme d'une paire de deux fichiers : une *clé publique* et une *clé privée*. Ces deux fichiers sont *uniques à chaque utilisateur*, et s'utilisent de la manière suivante :

- La clé publique est installée sur un système distant (par exemple un serveur Linux) sur laquelle on souhaite se connecter. *Cette installation de clé n'est à faire qu'une seule fois* par système distant.
- La clé privée est utilisée pour prouver au système distant que nous sommes bien le propriétaire de cette clé publique.

Pour mieux comprendre tout ce fonctionnement, ne pas hésiter à se renseigner sur la cryptographie asymétrique [https://fr.wikipedia.org/wiki/Cryptographie_asym%C3%A9trique], qui est la technique à l'œuvre ici.

Pour générer sa clé SSH personnelle unique, on exécute la commande suivante depuis un système Linux :

[sur votre poste de travail local]

```
$ ssh-keygen -t ecdsa
```

Suite à cette commande, un message s'affiche qui débute par :

```
Enter file in which to save the key
```

Il n'est pas nécessaire de saisir un fichier/chemin, vous pouvez simplement appuyer sur *Entrée* pour qu'il utilise le chemin par défaut (qui est le chemin souhaité de toute manière).

L'option `-t ecdsa` s'assure qu'on génère une clé à l'aide d'un standard cryptographique puissant et moderne, à savoir ECDSA [https://fr.wikipedia.org/wiki/Elliptic_curve_digital_signature_algorithm].

Une **phrase de passe** (en anglais, *passphrase*) de protection de clé, optionnelle, est alors demandée. Une phrase de passe permet de *chiffrer votre clé privée* SSH, afin d'éviter à quelqu'un qui s'empare malicieusement de votre clé privée de pouvoir s'en servir. Dans ce cas, il faudra saisir la phrase de passe de la clé à chaque connexion.

Si on souhaite ne plus avoir à taper de mot de passe lors d'une connexion SSH, on peut choisir une *phrase de passe vide* – ce choix est tout à fait valide dans le cadre de ce cours, les enjeux de sécurité étant faibles.

Deux fichiers sont alors générés aux emplacements suivants de votre répertoire utilisateur :

- La clé publique : `.ssh/id_ecdsa.pub`
- La clé privée : `.ssh/id_ecdsa`

Vous êtes désormais prêt pour utiliser votre clé SSH, que ce soit pour une connexion à distance ou pour un usage avec Git !

Chapitre 2. ⚠️ Notice : environnement de déploiement

Un serveur vous est mis à disposition comme **environnement de déploiement** pour apprendre à y déployer des logiciels, et pour expérimenter l'usage du moteur de conteneurisation Podman. Sur le réseau interne de la faculté des sciences et techniques, il est accessible sous le nom d'hôte "podman" .

Quelques caractéristiques importantes de cet environnement d'accueil :

- **On ne peut accéder à ce serveur que depuis le réseau interne de l'université**. Vous ne pouvez notamment pas vous y connecter depuis l'extérieur de l'université (sauf via eduVPN).
- **On ne peut pas y ouvrir de session graphique**, et la seule manière de s'y connecter est au travers du *protocole* SSH, via un terminal et à l'aide d'un clavier. Voir Section 2.1.
- **Le moteur de conteneurisation *Podman* y est installé**. C'est dans cet environnement que vous apprendrez à manipuler des images et à déployer des conteneurs.
- **Votre répertoire étudiant habituel n'est pas accessible** depuis ce serveur. Vous possédez à la place un répertoire utilisateur uniquement pour ce serveur, non partagé avec votre répertoire étudiant habituel. Cela permet d'éviter que vos expériences de déploiement aillent altérer ou détruire vos données utilisateur habituelles.
- **On ne peut accéder à ce serveur que par les ports réseau 8080 à 9089** (sans compter le port 22 utilisé par SSH). Ainsi, vos déploiements devront toujours écouter sur des ports de cette intervalle, et plus précisément sur les ports d'un intervalle qui vous sera attribué individuellement, voir Section 2.2.1.

Quelques détails supplémentaires

Même si cela n'est pas utile pour les TPs, vous pouvez noter que :

- Ce serveur est une **machine virtuelle** exécutée sur une machine physique du Service Commun Informatique (SCI) de l'université [https://intrasci.sciences.univ-nantes.fr/].
- Le **système d'exploitation** installé est la distribution Linux appelée Debian [https://www.debian.org/index.fr.html], en version 13. C'est un système d'exploitation utilisé de manière très courante pour un immense nombre de serveurs à travers le monde.

2.1. Se connecter au serveur de déploiement

Le **protocole SSH** est un protocole réseau standardisé permettant de se connecter de manière sécurisée à un environnement Linux distant. Même si différents outils existent pour créer une connexion SSH, le plus fréquent est d'utiliser l'utilitaire `ssh` du même nom. Cet utilitaire s'utilise uniquement via une interface textuelle depuis un terminal.

Le serveur de déploiement auquel on souhaite se connecter est accessible par le nom d'hôte `podman`.

2.1.1. Se connecter avec mot de passe

Pour se connecter par SSH au serveur de déploiement, il faut appeler l'outil `ssh` en lui indiquant qu'on souhaite se connecter sur l'environnement `podman` à l'aide du même identifiant que son compte université habituel :

[sur votre poste de travail local]

```
$ ssh EXXXXXX@podman
```

en remplaçant `EXXXXXX` par son identifiant étudiant habituel.

Il est nécessaire d'écrire votre identifiant étudiant en MAJUSCULES.

Si c'est la première fois que vous vous connectez, vous aurez un message similaire à celui-ci :

```
The authenticity of host 'podman.ensinfo.sciences.univ-nantes.prive
(172.16.129.149)' can't be established.
ED25519 key fingerprint is
SHA256:WF+uCve0uqbVBlnsIB9vMGj/4p1Pihlxtfz79KjpYEs.
This key is not known by any other names.
Are you sure you want to continue connecting (yes/no/[fingerprint])?
```

Ce message est parfaitement normal : il permet (si besoin) de vérifier l'identité du serveur SSH auquel vous vous connectez, et de confirmer explicitement en saisissant `yes` que c'est bien le bon serveur.

Une fois ce message vu une première fois, il ne s'affichera plus par la suite, car le serveur SSH sera mémorisé comme "sûr".

Il faut alors saisir votre mot de passe dans une boîte de texte invisible, puis valider.

Mais on peut observer que vous êtes normalement déjà connecté·e avec un compte utilisateur `eXXXXXX` sur la machine de TP que vous utilisez. On peut donc simplifier la commande précédente en omettant l'identifiant de connexion à utiliser, car `EXXXXXX` sera utilisé par défaut :

[sur votre poste de travail local]

```
$ ssh podman
```

```
bousse-e@tpinfo3:~$ ssh podman
Linux podman 6.1.0-9-amd64 #1 SMP PREEMPT_DYNAMIC Debian 6.1.27-1 (2023-05-08) x86_64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Wed Sep 20 10:40:44 2023 from 172.16.129.114
bousse-e@podman:~$ echo "Hello again"
Hello again
bousse-e@podman:~$ ls
deployments  git  tmp
bousse-e@podman:~$
```

Figure 2. Exemple de connexion SSH au serveur de déploiement

Une fois la connexion établie, vous verrez que votre invite de commande (aussi appelé *prompt*) aura changé de couleur, et ressemble désormais à :

[sur l'environnement de déploiement distant]

```
EXXXXXX@podman:~$
```

Vous êtes désormais en connexion à distance sur le serveur de déploiement ! Toutes les commandes que vous exécutez ici se produiront sur ce serveur distant, et non sur le poste devant lequel vous vous trouvez.

Par exemple si vous exécutez la commande :

[sur l'environnement de déploiement distant]

```
$ ls
```

vous pourrez observer que les fichiers ne sont pas identiques à ceux de votre répertoire étudiant habituel.

De même, si vous exécutez la commande :

[sur l'environnement de déploiement distant]

```
$ firefox
```

vous aurez en retour le message `commande introuvable`, car Firefox n'est pas installé sur le serveur de déploiement.

Par contre, si vous exécutez la commande :

[sur l'environnement de déploiement distant]

```
$ podman --version
```

vous aurez alors un retour vous indiquant la version installée de Podman.

Enfin, pour quitter la session SSH, faire simplement la commande :

[sur l'environnement de déploiement distant]

```
$ exit
```

2.1.2. Se connecter avec une clé SSH

Avec la méthode présentée plus haut, il est nécessaire de saisir un simple mot de passe à chaque connexion SSH. **Pour rendre plus sûres et plus rapides vos connexions SSH**, il est possible d'activer la connexion par **clé SSH**, qui remplace alors la connexion par mot de passe.

Pour cela vous devez avoir au préalable généré votre clé SSH personnelle comme indiqué dans la section dédiée de la notice infrastructure.

Si cela est déjà fait, **la commande suivante permet de mettre en place la connexion par clé SSH** au serveur de déploiement (*à ne faire qu'une seule fois*) :

[sur votre poste de travail local]

```
$ ssh-copy-id podman
```

Il faut alors saisir votre mot de passe utilisateur, nécessaire pour ouvrir une brève connexion SSH pour copier la clé.

Et normalement tout est désormais bon ! Pour tester, faites :

[sur votre poste de travail local]

```
$ ssh podman
```

et la connexion se fera à l'aide de votre clé SSH et sans nécessiter votre mot de passe de connexion.

2.2. Utiliser le serveur de déploiement

Une fois connecté sur le serveur, on peut utiliser Podman comme présenté dans le cours dans les chapitres 📎 Conteneuriser (manuellement) un logiciel en image de conteneur [https://gl.univ-nantes.io/developpement_exploitation/workbook/site/2026-2027/cms/creer-images/] et 📎 Créer et manipuler des conteneurs [https://gl.univ-nantes.io/developpement_exploitation/workbook/site/2026-2027/cms/creer-conteneurs/].

D'autres points techniques vous sont expliqués ci-après.

2.2.1. Usage discipliné des ports disponibles

Comment évoqué précédemment, les seuls ports réseaux que nous avons le droit d'utiliser sur le serveur de déploiement sont les ports 8080 à 9089. Étant donné que ce serveur sera partagé par tous les étudiants, **il est nécessaire de faire preuve d'une certaine discipline pour partager ces ports entre tous les usagers du serveur**.

Une liste intitulée "*Attribution par étudiant·e des ports réseaux du serveur de déploiement*" est disponible sur Madoc [https://madoc.univ-nantes.fr/mod/resource/view.php?id=2396489], et attribue à chaque étudiant·e **un intervalle de ports**. Il vous est demandé de *ne pas faire de publications de ports avec Podman en dehors de votre intervalle.*

Il ne sera pas possible de facilement vérifier que chaque étudiant·e respecte cette règle, c'est pourquoi nous comptons sur votre rigueur et votre courtoisie pour ne pas aller utiliser des ports réservés à vos camarades !

2.2.2. Créer et éditer un fichier

Lors de votre travail avec Podman sur le serveur de déploiement, il peut arriver que vous ayez besoin de créer ou modifier un fichier dans cet environnement, par exemple un fichier `Containerfile` ou `compose.yml`. Or, comme évoqué précédemment, **cet environnement ne possède pas d'interface graphique, et on ne peut y agir que par le terminal**.

Il est possible à la place d'utiliser un éditeur de texte sans interface graphique, en utilisant un utilitaire

console comme `nano`.

Pour créer ou ouvrir un fichier avec `nano`, depuis un terminal :

[sur l'environnement de déploiement distant]

```
$ nano monfichier
```

On arrive alors dans un éditeur de texte basique où tout doit se faire au clavier. On promène un indicateur de position dans le texte à l'aide des flèches directionnelles, et on écrit là on l'indicateur se situe.

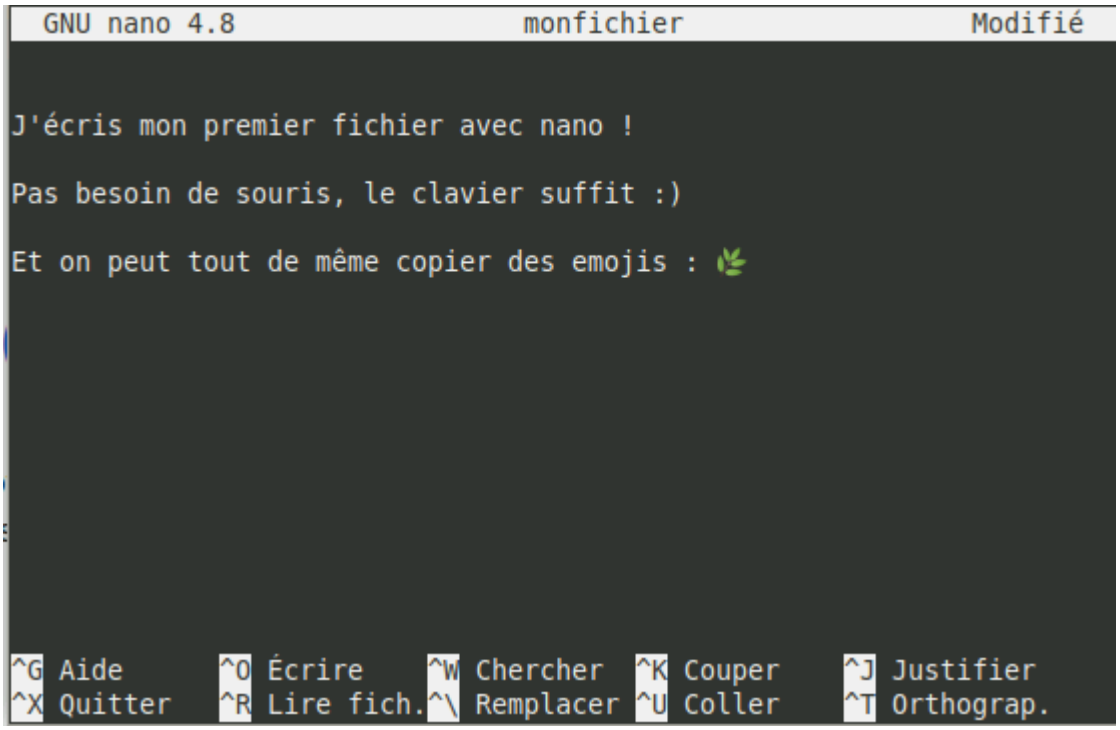


Figure 3. Exemple d'écriture de fichier avec `nano`

On peut voir en bas de l'écran un ensemble de raccourcis écrits sous la forme `^W` ou `^X`. Le symbole `^` signifie qu'il faut utiliser la touche clavier CTRL pour réaliser le raccourci. Par exemple, `^X` signifie qu'il faut faire CTRL + `x` pour exécuter l'action correspondante.

Les raccourcis les plus importants sont les suivants :

- CTRL + `o` : enregistrer le fichier (faire *Entrée* pour valider le nom du fichier)
- CTRL + `w` : rechercher dans le fichier
- CTRL + `k` : supprime la ligne actuelle
- CTRL + `x` : quitte l'éditeur

Pour copier et coller du texte sélectionné dans un terminal, la bonne manipulation est CTRL+SHIFT+`c` (pour copier), et CTRL+SHIFT+`v` (pour coller).

2.2.3. Transférer des fichiers

Comme évoqué précédemment, **le serveur de déploiement n'a pas directement accès à votre**

répertoire étudiant habituel. Il peut cependant être utile de copier des fichiers depuis votre répertoire étudiant vers le serveur, ou inversement depuis le serveur vers votre répertoire étudiant.

Voyons deux méthodes très différentes pour faire de tels transferts de fichiers.

Via scp

L'utilitaire `scp` est capable de copier des fichiers d'une machine à une autre à l'aide du protocole SSH.

Pour des raisons de sécurité, il est conseillé de ne pas utiliser une version de `scp` antérieure à la version 9.0 de OpenSSH. Si ce n'est pas possible, il faut privilégier l'utilitaire `rsync` en lieu et place.

Pour copier un fichier vers le serveur de déploiement depuis un poste Linux :

[sur votre poste de travail local]

```
$ scp monfichier podman:~
```

Pour rappel, le symbole `~` est un raccourci qui signifie *répertoire utilisateur*, et que dans cette commande cela fait référence à votre répertoire utilisateur *sur le serveur podman*.

Si on souhaite copier le fichier dans un sous-répertoire appelé `mon_repertoire_distant`, on peut faire :

[sur votre poste de travail local]

```
$ scp monfichier podman:~/mon_repertoire_distant
```

Pour copier un répertoire vers le serveur de déploiement depuis un poste Linux :

[sur votre poste de travail local]

```
$ scp -r mon_repertoire podman:~
```

On notera l'usage de l'option `-r` qui signifie "copie récursive" et qui est nécessaire pour copier un répertoire.

Pour télécharger un fichier situé sur le serveur de déploiement, vers le répertoire actuel d'un poste Linux :

[sur votre poste de travail local]

```
$ scp podman:~/monfichier .
```

Pour rappel, le point (`.`) est un raccourci qui signifie "répertoire courant".

2.2.4. Gérer ses données et son espace disque sur le serveur

Comme expliqué précédemment, chaque utilisateur du serveur de déploiement possède son propre espace de stockage sur ce dernier. Cet espace peut être utilisé pour effectuer des déploiements de logiciels, mais est également utilisé par Podman pour stocker toutes les données dont il a besoin.

En particulier, Podman utilise votre espace sur le serveur afin de stocker :

- toutes les **images de conteneurs** téléchargées ou créées,
- tous les **conteneurs** créés.

L'espace utilisé peut donc monter très très vite ! Pour ne pas dépasser votre quota, il vous est recommandé, à la fin de chaque séance de TP ou de chaque période de travail, de faire les commandes suivantes :

Première chose simple faire, vous pouvez stopper tous vos conteneurs :

[sur l'environnement de déploiement distant]

```
$ podman container stop --all
```

Puis vous pouvez supprimer tous vos conteneurs stoppés :

[sur l'environnement de déploiement distant]

```
$ podman container prune
```

Ainsi que vos réseaux virtuels désormais non utilisés :

[sur l'environnement de déploiement distant]

```
$ podman network prune
```

Enfin, vous pouvez supprimer toutes les images de conteneurs que vous avez téléchargées lors de vos créations de conteneurs :

[sur l'environnement de déploiement distant]

```
$ podman image prune --all
```

Il est également possible de remettre *entièrement* les données de Podman (images, conteneurs, réseaux, etc.) à zéro à l'aide de la commande suivante :

[sur l'environnement de déploiement distant]

```
$ podman system reset
```

Vous avez créé un certain nombre de répertoires et de fichiers pour vos conteneurs, et ces derniers ont également créé de nouveaux fichiers au sein de des répertoires que vous leur avez fourni par montage.

Cependant, si vous souhaitez supprimer toutes les données créées par vos conteneurs, par exemple à l'aide de la commande `rm -rf <mon répertoire>`, **vous ferez parfois face à des erreurs de permission !**

Pour bien comprendre ce qu'il se passe, et pour résoudre ce problème, se référer à la section correspondante du chapitre sur l'exécution de conteneurs [https://gl.univ-nantes.io/developpement_exploitation/workbook/site/2026-2027/cms/conteneurs-donnees/#sec-unshare].

Chapitre 3. ⚠️ Notice : Git et Gitlab

L'instance Gitlab de l'université [https://gitlab.univ-nantes.fr/] est un déploiement du logiciel Gitlab [https://gitlab.com/gitlab-org/gitlab], une forge logicielle *open-source* archi complète et similaire en terme de fonctionnalités au service en ligne commercial Github [https://github.com/].

Dans le cadre du cours, nous allons juste apprendre à **cloner le code source d'un projet Gitlab**, et à utiliser la fonctionnalité **registre de conteneurs** de Gitlab, afin d'y publier nos images de conteneur.

3.1. Connexion et page d'accueil

Pour se connecter sur le Gitlab de l'université, se rendre sur la page d'accueil [https://gitlab.univ-nantes.fr], choisir le mode de connexion *LDAP*, et saisir ses informations de connexion universitaires habituelles.

On arrive alors sur une page d'accueil similaire à ceci :

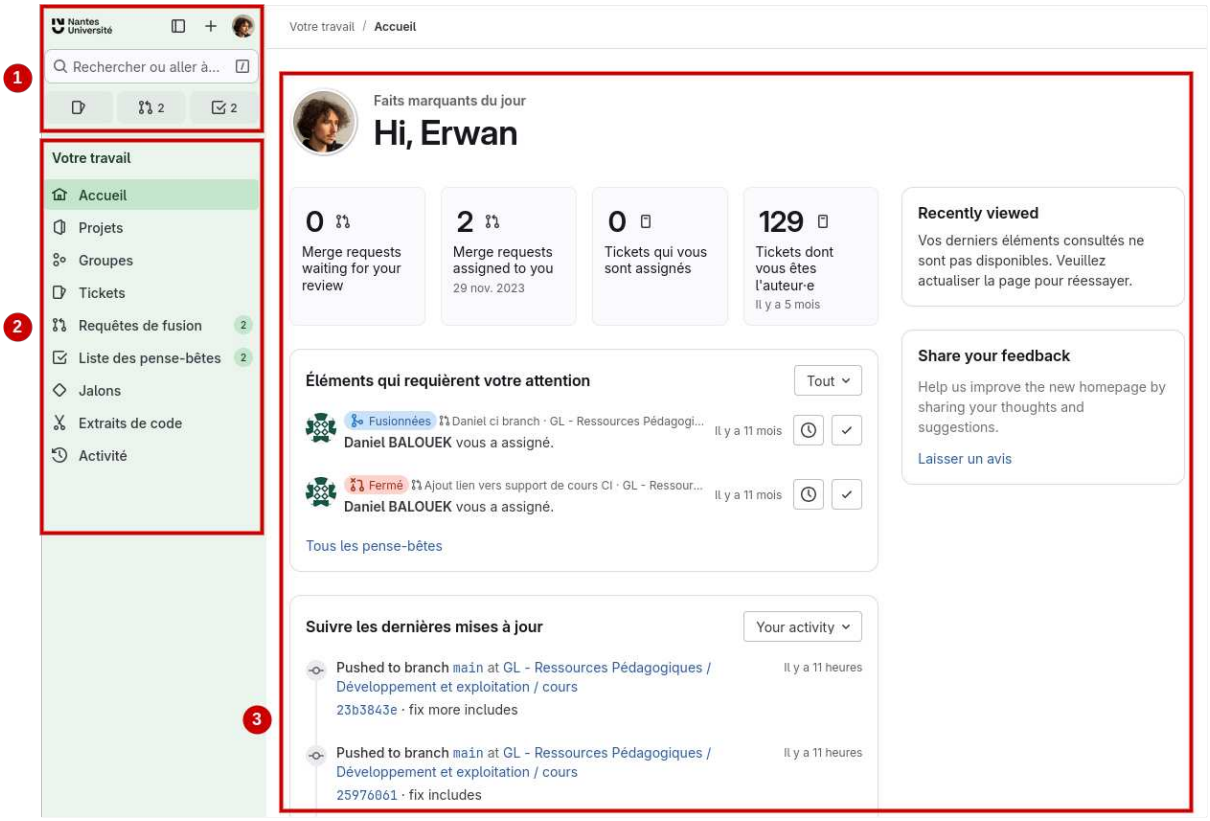


Figure 4. Écran d'accueil après connexion à Gitlab

1. Tout en haut à gauche, **un ensemble de boutons généraux** vous accompagnent sur toutes les pages où vous naviguerez.

On peut cliquer sur son avatar pour accéder au menu permettant de modifier son profil et ses préférences. On peut ainsi changer l'apparence de Gitlab ou son avatar.

2. À gauche, **un panneau permet d'accéder toutes les sections accessibles du contexte actuel**. Par défaut une fois connecté, c'est le contexte *Votre travail* qui est actif, et la sections *Accueil* qui est ouverte. On peut voir que dans le contexte *Votre travail*, les sections proposées incluent notamment tout ce qu'on peut créer avec un compte Gitlab (projets, groupes, tickets, etc).

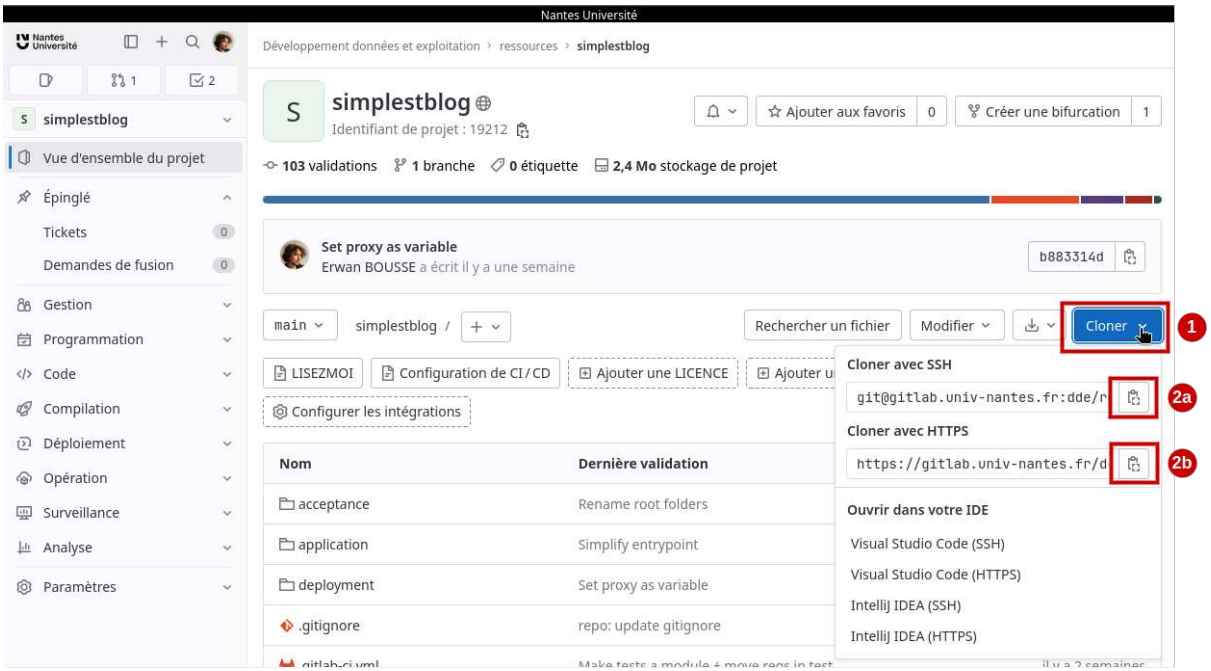
Le contenu du panneau changera complètement en fonction du contexte ! Par exemple si on navigue dans un projet, alors le panneau proposera toutes les sections en lien avec un projet.

3. Au milieu, sur la largeur de l'écran, on peut lire un résum des **dernières activités liées à votre compte**.

3.2. Cloner le code source d'un projet Gitlab

Pour récupérer le code source d'un projet Gitlab, on effectue un **clone** projet.

On se rend pour cela sur la page d'accueil d'un projet Gitlab :



1. Cliquer sur le bouton *Cloner* ouvre un menu avec différentes informations.
2. Deux adresses sont proposées, chacune avec un bouton à droite pour copier cette adresse dans votre presse-papier :
   a. *Cloner avec SSH* est l'adresse SSH pour cloner,
   b. *Cloner avec HTTPS* est l'adresse HTTPS pour cloner.
3. On peut ignorer les actions proposées dans *Ouvrir dans votre IDE*.

Le plus simple est de faire un clone par HTTPS, car le clone par SSH nécessite d'enregistrer une clé SSH dans Gitlab au préalable.

Ensuite on peut ouvrir un terminal sur un poste de travail Linux, et effectuer la commande suivante :

```
$ git clone adresse_ssh_ou_https
```

en remplaçant `adresse_ssh_ou_https` par le contenu de votre presse-papier.

Suite au `clone`, vous avez désormais un nouveau répertoire sur votre poste, qui porte normalement le même nom que votre projet Gitlab. Ce répertoire contiendra toujours à la fois le code source du

projet, et également un répertoire caché appelé `.git` qui contient tout l'historique Git du projet.

3.3. Stocker ses images de conteneur dans le registre

3.3.1. Créer un projet

Pour créer un projet vide, deux possibilités :

- soit depuis la section *Projets*, on utilise le bouton *Nouveau projet* situé tout en haut à droite de l'écran,
- soit depuis les boutons généraux en haut à gauche, on clique sur le ▫ et on choisit *Nouveau projet/dépôt*.

On choisit ensuite *Créer un projet vide*, et on arrive sur l'écran suivant :

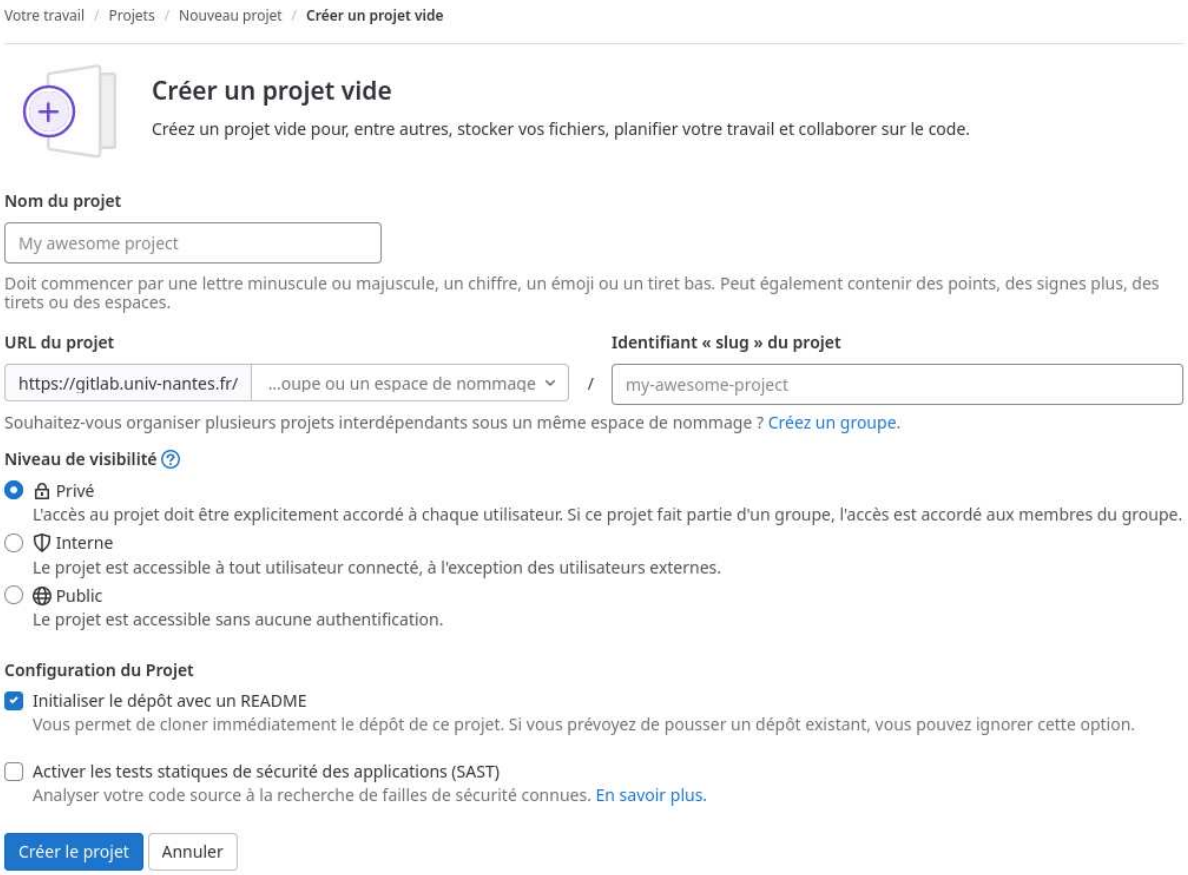


Figure 5. Écran de création de projet Gitlab

Concernant les informations à saisir :

- ***Nom du projet*** : Une description simple claire de votre projet, en plusieurs mots si nécessaire
- ***URL du projet*** : Un menu déroulant permet de choisir un *emplacement* pour le projet. En effet, un projet peut soit appartenir à un *compte utilisateur*, soir à un *groupe*. Dans le cadre du cours, vous pouvez ici toujours sélectionner votre propre compte utilisateur.
- ***Identifiant « slug » du projet***: Ne rien changer ici, sera automatiquement déduit du nom du projet.
- ***Niveau de visibilité*** : Dans le cadre du cours, on créera toujours des projets privés, afin de permettre à chaque groupe de travailler isolément.
- ***Configuration du Projet*** : Il est conseillé de *décocher* les deux cases, pour partir d'une

configuration vierge.

On peut alors valider avec le bouton *Créer le projet*, et on arrive sur l'écran suivant :

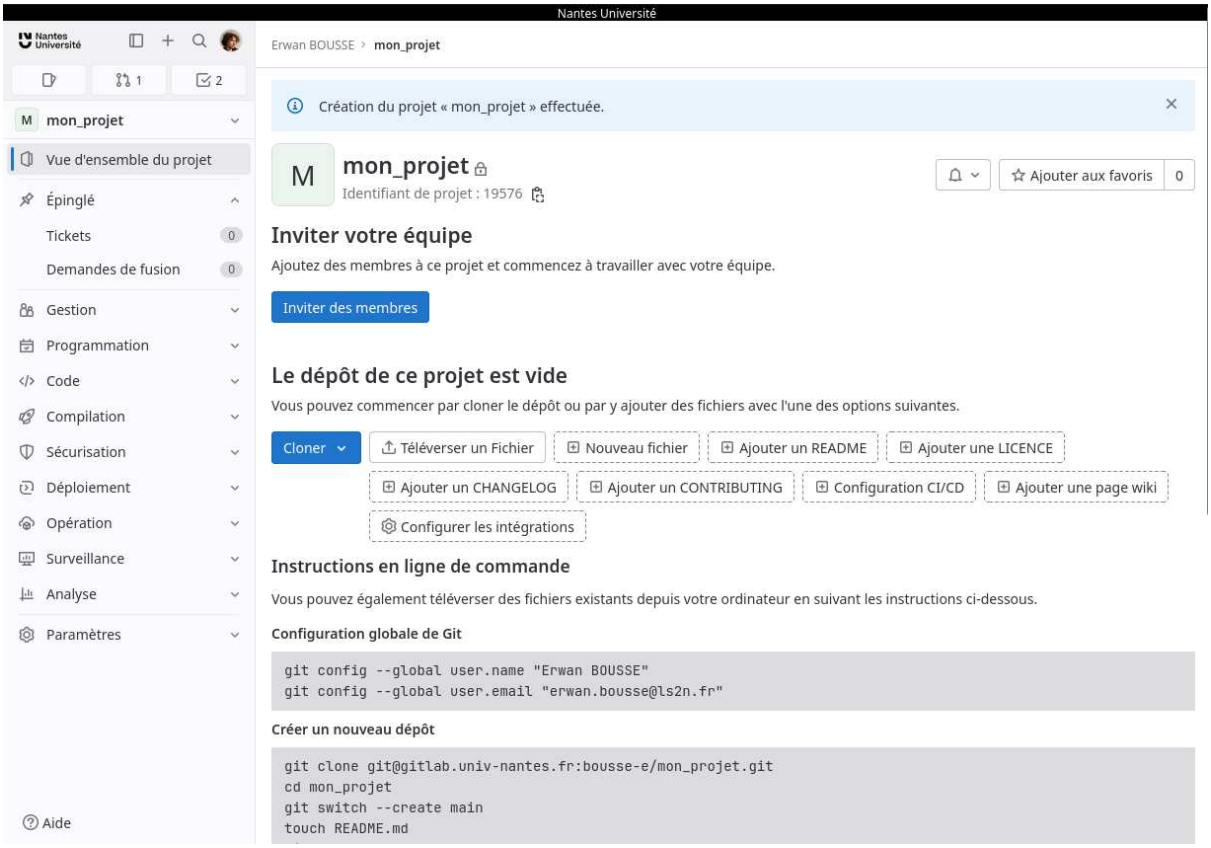


Figure 6. Écran après avoir créé un projet vide

On peut voir que :

- Le panneau de gauche a changé de contexte. Le nouveau contexte est celui du projet tout juste créé.
- La section sélectionnée par défaut est *Vue d'ensemble du projet*.
- Un message vous invite à *Inviter votre équipe*, autrement dit à ajouter des membres au projet pour pouvoir y travailler à plusieurs.
- Un message vous indique que "*Le dépôt de ce projet est vide*", et vous invite à *ajouter des fichiers*. On peut ignorer ces boutons car nous ajouterons les fichiers via Git.
- Une longue liste "*Instructions en ligne de commande*" est fournie. On peut l'ignorer car on ne travaillera pas avec Git.

3.3.2. Publier une image de conteneur au sein d'un projet

Au sein d'un projet Gitlab, il sera possible de *publier* des images de conteneur que vous aurez préparées.

Pour publier une image présente dans votre stockage local d'image, la commande à utiliser sera la suivante :

[sur l'environnement de déploiement distant]

```
$ podman image push <image du stockage local> \
```

```
                    docker-registry.univ-nantes.fr/<chemin projet>/<nom image>:<étiquette
image>
```

où `<chemin projet>` est le chemin de votre projet au sein de l'arborescence Gitlab. Si vous avez le compte étudiant `eXXXXX` et que vous avez nommé votre projet `mon_super_projet`, alors votre chemin de projet aura la forme `eXXXXX/mon_super_projet`.

Il est nécessaire de mettre votre compte étudiant en *minuscules* pour que le chemin cible de publication d'image fonctionne.

3.3.3. Accéder aux images de conteneurs d'un projet

Chaque projet Gitlab se voit attribuer un espace au sein du **registre de conteneurs** de l'instance Gitlab. Cet espace permet de stocker un ensemble d'images de conteneurs associées à ce projet.

Pour accéder aux images de conteneurs, se rendre rendant dans la section *Déploiement* → *Registre de conteneur*

Chapitre 4. 💻 TP : Découverte de l'infrastructure et des outils

4.1. Présentation et objectifs

Les travaux pratiques de ce cours peuvent avoir l'air un peu intimidants : beaucoup d'outils différents, beaucoup d'actions à réaliser dans un terminal Linux, des projets assez complexes à comprendre et à manipuler, etc.

Ce premier TP vise à vous rendre opérationnels quant à l'usage des outils de base, ainsi que l'usage des services fournis par l'infrastructure à notre disposition.

Les objectifs de ce TP sont donc de :

- générer votre première **clé SSH** personnelle,
- vous familiariser avec **l'interaction avec le serveur de déploiement**, avec :
  - une connexion à l'aide du protocole SSH,
  - l'envoi de fichiers avec `scp`,
  - modifier des fichiers texte à distance.
- essayer quelques déploiements "à la main" dans un environnement limité (et sans conteneurisation), et constater certaines difficultés

Non seulement tout cela vous sera utile pour être à l'aise pour les TPs suivants, mais cela vous permettra également de **prendre conscience des tâches manuelles habituellement nécessaires pour manipuler un serveur distant**.

Avant de vous lancer sur le TP, il est **fortement recommandé** de lire à minima le début de la Chapitre 1, et d'ouvrir dans un onglet l'autre notice (Chapitre 2) qui servira tout au long du TP. Gardez également ouvert le chapitre 📎 Déploiement de logiciel [https://gl.univ-nantes.io/developpement_exploitation/workbook/site/2026-2027/cms/deployer-des-logiciels/]

4.2. Usages principaux du serveur de déploiement

Une part importante des TPs nécessite l'usage du serveur de déploiement mis à votre disposition. Apprenons à manipuler manuellement ce serveur.

4.2.1. Connexion SSH par mot de passe

1. Ouvrez un terminal, puis suivez les instructions pour vous connecter par SSH à l'aide d'un mot de passe dans la Chapitre 2.
2. Si tout va bien, vous êtes désormais connecté au serveur ! Par défaut vous êtes dans votre *répertoire utilisateur* spécifique à ce serveur. Pour voir l'emplacement de ce répertoire sur le serveur, faites `pwd`. Ce répertoire est initialement vide, et ne contient pas les données de votre répertoire étudiant habituel. Pour le constater, faites la commande `ls`, et constatez que vous n'avez aucun retour (car il n'y a aucun fichier).
3. Créez un nouveau répertoire appelé `tp-decouverte-serveur` à l'aide de la commande :

[sur l'environnement de déploiement distant]

```
$ mkdir tp-decouverte-serveur
```

4. Rendez vous dans ce répertoire à l'aide de la commande :

   [sur l'environnement de déploiement distant]

   ```
   $ cd tp-decouverte-serveur
   ```

5. Suivez les instructions pour créer et éditer un fichier depuis le terminal toujours dans la Chapitre 2. Préparez un fichier avec pour nom `hello.txt`, et avec pour contenu `"Hello world!"`. Une fois l'édition terminée, utilisez la commande `cat hello.txt` pour vérifier le contenu du fichier.
6. Enfin, fermez votre connexion SSH à l'aide de la commande `exit`.

4.2.2. Générer sa clé SSH personnelle

1. Ouvrez un terminal, puis suivez les instructions pour générer votre clé SSH personnelle dans la Chapitre 1.

   Si vous faites ce TP en binôme, vous devez faire cette étape pour chaque membre du binôme, en se connectant à chaque fois sur la session du membre concerné !

2. Pour vérifier que vous avez réussi cette étape, ouvrez un terminal et faites la commande :

   [sur votre poste de travail local]

   ```
   $ ls ~/.ssh
   ```

   Si vous voyez les fichiers `id_ecdsa.pub` et `.ssh/id_ecdsa`, c'est que tout est OK, et vous pouvez passer à la suite.

4.2.3. Connexion SSH avec clé SSH

Apprenez à simplifier vos connexions SSH en suivant les instructions pour vous connecter à l'aide de votre clé SSH personnelle dans la Chapitre 2.

4.2.4. Envoi de fichiers vers le serveur

Nous avons vu comment créer des fichiers directement depuis le serveur, cependant l'édition depuis le terminal n'est pas le plus pratique. Voyons désormais comment préparer des fichiers sur votre poste Linux, avant de les transférer par le réseau jusqu'au serveur.

1. Sur votre poste Linux, préparez un nouveau répertoire appelé `tp-decouverte-envoi`.
2. Dans ce répertoire, préparez :
   - un premier fichier appelé `envoi1.txt` avec pour contenu `"Ceci est mon premier envoi"`
   - un sous-répertoire appelé `envoi2` et contenant :
     - un fichier appelé `envoi2_a.txt` avec pour texte `"Ceci est le premier fichier de mon`

`second envoi"`,

- un fichier appelé `envoi2_b.txt` avec pour texte `"Ceci est le second fichier de mon second envoi"`.

3. Ouvrez un terminal, puis suivez les instructions pour envoyer un fichier toujours dans la Chapitre 2. Vous devez transférer le fichier `envoi1.txt`, et le déposer dans le répertoire `tp-decouverte-serveur` situé sur le serveur de déploiement.
4. Ouvrez un second terminal, et connectez vous par SSH sur le serveur, puis allez vérifier que votre fichier est bien présent dans le répertoire `tp-decouverte-serveur`. Utilisez la commande `cat` pour vérifier le contenu du fichier.
5. Dans le premier terminal, faites un nouveau transfert afin de copier le répertoire `envoi2` au sein du répertoire `tp-decouverte-serveur` situé sur le serveur de déploiement.
6. Dans le second terminal, allez vérifier que l'envoi du répertoire a bien fonctionné. Utilisez la commande `cat` pour vérifier le contenu des deux fichiers qui étaient dans le répertoire envoyé.

4.3. Comprendre un peu l'environnement de déploiement

4.3.1. Carte d'identité de l'environnement

Il est utile de se renseigner sur l'environnement sur lequel on intervient, notamment connaître son système d'exploitation.

Une première façon de connaître le système d'exploitation d'un environnement Linux est de lire contenu du fichier standard `/etc/os-release`, par exemple avec la commande suivante :

[sur l'environnement de déploiement distant]

```
$ cat /etc/os-release
```

- Exécutez cette commande sur l'environnement de déploiement étudié, et déterminez quel est le système d'exploitation et sa version en lisant les clés `NAME` et `VERSION`.

Une seconde façon de connaître le système d'exploitation, ainsi que d'autres informations utiles, est d'exécuter l'outil `hostnamectl`, souvent (mais pas toujours) disponible dans un environnement Linux :

[sur l'environnement de déploiement distant]

```
$ hostnamectl
```

- Exécutez cette commande sur l'environnement de déploiement étudié, et déterminez quel est le système d'exploitation et sa version en lisant la clé `Operating system`. Observez également les autres clés disponibles, par exemple la clé `Chassis` qui décrit le type de machine (`laptop` pour un ordinateur portable, `vm` pour une machine virtuelle, etc).

4.3.2. Variables d'environnement

Dans le chapitre Logiciel et exécution [https://gl.univ-nantes.io/developpement_exploitation/workbook/site/2026-2027/cms/logiciel-execution/], nous avons rappelé le méchanisme de **variable d'environnement**, qui permet de poser des variables dans un environnement notamment en vue de configurer un logiciel à déployer.

Pour jouer un peu avec ce méchanisme, dans l'environnement de déploiement :

1. Exécutez la commande `printenv` et constatez qu'un ensemble important de variables sont définies par défaut dans l'environnement.
2. Utilisez la commande `export` pour créer une nouvelle variable d'environnement `MAVARIABLE` avec la valeur `"mavaleur"`, puis constatez que votre variable est bien affichée dans `printenv`.
3. Utilisez la commande `echo` pour afficher le contenu de `MAVARIABLE`, en n'oubliant pas de préfixer le nom de la variable avec `$`.

4.3.3. Droits d'accès

Dans le chapitre 📘 Logiciel et exécution [https://gl.univ-nantes.io/developpement_exploitation/workbook/site/2026-2027/cms/logiciel-execution/], nous avons rappelé le méchanisme de **droit d'accès**, qui permet de limiter dans un environnement quelles actions peuvent être réalisées par des utilisateurs et/ou des processus logiciels.

Pour observer grossièrement certains droits d'accès en place sur dans l'environnement de déploiement :

1. Exécutez la commande `touch ~/fichier_temporaire` pour créer un nouveau fichier appelé `fichier_temporaire` dans votre répertoire utilisateur (dénoté par `~`). Puis, utilisez la commande `ls` pour vérifier que ce fichier a bien été créé. Constatez que tout s'est bien passé (le fichier a été créé), puis supprimez le fichier avec `rm`. Cela confirme que *vous avez tous les droits dans votre répertoire utilisateur*.
2. Exécutez la commande `touch /etc/fichier_temporaire` pour créer un nouveau fichier appelé `fichier_temporaire` dans le répertoire système `/etc`. Constatez que cette action n'est pas autorisée, car *votre compte de possède pas les droits suffisants.*
3. Exécutez la commande `apt install git` pour installer un nouveau logiciel (à savoir `git`) dans les logiciels système de l'environnement de déploiement. *Constatez à nouveau que cette commande échoue* : en effet, `apt` cherche à installer le logiciel au sein des données sytème (répertoire `/usr/bin` notamment), or votre compte utilisateur ne possède pas le droit d'écrire dans les données systèmes.

En résumé, **vos droits sur l'environnement de déploiement sont très limités** ! En particulier, sans les droits `root`, il n'est pas facilement possible d'installer des logiciels ou des dépendances logicielles…

4.4. Quelques petits déploiements (sans conteneurs)

Dans cette dernière partie, tentons de **déployer** des petits programmes sur l'environnement de déploiement.

Les programmes peuvent être trouvés en visitant cette page web [https://gitlab.univ-nantes.fr/gl/developpement_exploitation/resources/small_programs]. Dans cette partie on ne s'intéresse qu'aux **quatre premiers programmes** (on ne traitera pas `5_economic_dispatch`).

Ici le terme "déploiement" est légèrement exagéré, car il sera question de *très* petits logiciels jouets qui ne rendent peu ou pas de services concrets, et qui ne fonctionnent pas en service continu.

4.4.1. `replace_string`

Sur la page web avec les différents programmes, rendez-vous dans le répertoire `1_replace_string`.

1. Lisez bien le README qui vous est présenté, et qui affiche la documentation technique et utilisateur de l'outil.
2. Téléchargez `replace_string.py` sur votre machine de TP, puis vous transférez ce fichier dans l'environnement de déploiement.
3. Testez l'outil sur l'environnement de déploiement :
   a. Créez un nouveau fichier de votre choix, et mettez-y le contenu exemple proposé dans le README
   b. Exécutez l'outil pour remplacer le mot `bottle` par le mot de votre choix.

Tout fonctionne comme prévu ? Si oui, répondez aux questions suivantes :

1. Quelle dépendance est requise pour faire fonctionner `1_replace_string` ?
2. Avez-vous vérifié que cette dépendance était présente dans l'environnement de déploiement avant de déployer ?
3. Comment vérifier que cette dépendance est bien présente dans l'environnement de déploiement ?

La plupart des logiciels exécutables dans un terminal savent répondre à l'option `--version`, qui permet de juste vérifier que le logiciel existe est quelle version est installée.

4.4.2. `compress_file`

Sur la page web avec les différents programmes, rendez-vous dans le répertoire `2_compress_file`.

Suivez le même principe que pour le programme précédent (`replace_string`) : lisez bien la documentation, puis déployez le logiciel sur l'environnement de déploiement, et enfin tentez de l'exécuter sur un exemple que vous aurez préparé.

1. Que se passe-t-il à l'exécution ?
2. Est-ce que la dépendance requise est bien présente comme souhaité ?
3. Peut-on installer la dépendance requise ? Si non, pourquoi ?

La fonction `zstd.compress` a été ajoutée uniquement dans la version Python 3.14, et n'est pas disponible avec Python 3.13.

4.4.3. `print_colored_string`

Sur la page web avec les différents programmes, rendez-vous dans le répertoire `3_print_colored_string`.

Suivez le même principe que pour les programmes précédents : lisez bien la documentation, puis déployez le logiciel sur l'environnement de déploiement, et enfin tentez de l'exécuter sur un exemple que vous aurez préparé.

1. Que se passe-t-il à l'exécution ?

2. Est-ce que les dépendances requises sont bien présentes comme souhaité ?
3. Peut-on installer les dépendances requises ? Si non, pourquoi ?

4.4.4. `TicTacToe`

Sur la page web avec les différents programmes, rendez-vous dans le répertoire `4_tictactoe`.

Suivez le même principe que pour les programmes précédents : lisez bien la documentation, puis déployez le logiciel sur l'environnement de déploiement, et enfin tentez de l'exécuter sur un exemple que vous aurez préparé.

1. Que se passe-t-il à l'exécution ?
2. Est-ce que la dépendance requise est bien présente comme souhaité ?
3. Peut-on installer la dépendances requise ? Si non, pourquoi ?

4.4.5. Synthèse

Avec ces petits déploiements, vous avez pu commencer à découvrir les besoins que peut avoir un logiciel lorsqu'il faut le déployer, quelques exemples de problèmes de dépendances (mauvaise version, bibliothèque manquante, logiciel manquant, etc.) ainsi que les caractéristiques de l'environnement de déploiement (droits limités, logiciels pré-installés limités).

Le prochain TP vous montrera comment se libérer de ces contraintes en créant *vos propres environnements* sous la forme d'un *conteneurs* !

Chapitre 5. 💻 TP : Créer et manipuler des conteneurs

5.1. Présentation et objectifs

L'objectif de ce TP est d'apprendre à utiliser Podman pour créer et manipuler des conteneurs de manière interactive à partir d'images de conteneurs pré-existantes, de gérer le cycle de vie d'un conteneur, et de manuellement déployer des petits programmes au sein des dits conteneurs.

Lorsque le sujet de TP vous propose d'exécuter une commande, il est demandé que vous arriviez à comprendre tout ce que comporte cette commande. **Prenez le temps de bien lire et décortiquer chaque commande qui vous est proposée**, et n'hésitez pas à poser des questions.

5.2. Connexion au serveur de déploiement

Toute la suite du TP devra être effectuée sur l'environnement de déploiement. Vous pouvez donc commencer par ouvrir un terminal et ouvrir une session SSH à distance sur ce dernier, comme vous avez appris à le faire dans le Chapitre 4.

Il est tout à fait possible d'ouvrir plusieurs terminaux en parallèle. Vous pouvez ainsi, si nécessaire ouvrir plusieurs connexions vers le serveur de déploiement. Cela peut être utile par exemple pour lancer un conteneur dans un premier terminal (avec `run`), et l'observer (avec `ps` ou `logs`) ou le manipuler (avec `stop` ou `rm`) depuis un second. Vous pouvez également garder un autre terminal ouvert pour exécuter des commandes sur votre poste Linux.

5.3. Tutoriel : un premier conteneur Ubuntu

5.3.1. Exécution du conteneur

Dans cette première partie, on se propose de créer un conteneur qui n'embarque aucun logiciel particulier, et qui permet simplement d'interagir avec un système d'exploitation restreint Linux classique. On choisit pour cela d'utiliser l'image de conteneur officielle Ubuntu [https://hub.docker.com/_/ubuntu].

Dans un premier terminal connecté au serveur de déploiement, exécutez la commande suivante :

[sur l'environnement de déploiement distant]

```
$ podman container run --name mon_premier_conteneur --tty --interactive
docker.io/ubuntu:24.04
```

Les options `--tty` et `--interactive` nous permettent de lancer le conteneur en mode interactif, de manière à pouvoir saisir des commandes au clavier une fois le conteneur créé.

Cette commande indique à Podman de créer un nouveau conteneur à partir de l'image de conteneur ayant pour référence `docker.io/ubuntu:24.04`, ce qui correspond à l'image :

- située sur le registre de conteneur `docker.io` (autrement dit, le Docker Hub)
- nommée `ubuntu`
- avec pour étiquette `24.04`, ce qui correspond à la version d'Ubuntu souhaitée.

Une fois ce conteneur créé, Podman va automatiquement exécuter le *point d'entrée* de l'image. Dans le cas de l'image Ubuntu choisie, ce point d'entrée est l'interpréteur de commandes Bash, à savoir le même que celui que vous utilisez par défaut dans le terminal sur un poste informatique Linux de l'Université.

Ainsi, une fois le conteneur lancé, on se retrouve face à un invite de commande (ou *prompt*) qui nous invite à exécuter des commandes.

Vous pouvez célébrer cela par une commande comme :

[sur le conteneur]

```
# echo "Bonjour depuis un conteneur !"
```

qui affichera `Bonjour depuis un conteneur !` de l'interpréteur de commandes du conteneur. Ce n'est pas grand chose, mais c'est un début !

Vous pouvez ensuite vérifier que ce nouvel environnement possède bien Ubuntu 24.04 comme système d'exploitation, à l'aide de la commande suivante (déjà vue dans le TP précédent) :

[sur le conteneur]

```
# cat /etc/os-release
```

Malheureusement la commande `hostnamectl`, plus informative que celle ci-dessus, n'est généralement pas pré-installée dans les systèmes d'exploitation restreints des conteneurs.

Enfin, il est utile de vérifier que votre conteneur est bien créé et en cours d'exécution. Pour cela, **à l'aide d'un second terminal**, exécutez la commande :

[sur l'environnement de déploiement distant]

```
$ podman container list
```

Vous verrez votre conteneur `mon_premier_conteneur` apparaître.

5.3.2. Installation d'un paquet comme administrateur du conteneur

> Hourra, je suis `root` !
>
> — Étudiant exécutant son premier conteneur

On peut voir dans le prompt affiché que l'utilisateur actif dans le conteneur s'appelle `root`, autrement dit un utilisateur qui possède les droits administrateur !

Mais comment est-ce possible, alors que vous n'avez normalement jamais de tels droits sur une

poste informatique de l'Université ?

Un conteneur est un environnement différent et dissocié de l'environnement de déploiement sur lequel vous travaillez (même si hébergé par ce dernier), et fonctionne via un système d'exploitation *restreint* sur lequel vous possédez tous les droits. Au sein de ce système exploitation restreint, vous êtes par défaut `root`, et vous pouvez ainsi y faire ce que vous souhaitez, même si vous n'êtes pas `root` dans le système d'exploitation de l'environnement de déploiement.

Par exemple, les commandes suivantes permettent, depuis un environnement Ubuntu, d'utiliser le gestionnaire de paquets `apt` pour installer un utilitaire appelé `cowsay` (utilitaire très utile permettant, dans un terminal, de dessiner une vache qui parle) :

[sur le conteneur]

```
$ apt update
$ apt install cowsay
```

Sur votre poste Linux de l'Université, si vous ouvrez un nouveau terminal et que vous exécutez ces commandes, alors une erreur de type `Permission denied` sera affichée, car vous n'avez pas les droits administrateurs.

En revanche, si vous exécutez ces mêmes commandes depuis votre conteneur, cela fonctionnera, et l'utilitaire `cowsay` sera installé dans le mini-environnement. Vous pouvez célébrer cette installation en exécutant cette commande dans votre conteneur, et en admirant le résultat :

[sur le conteneur]

```
# /usr/games/cowsay "Bonjour depuis un conteneur !"
```

5.3.3. Copier un fichier dans le conteneur

Nous souhaitons désormais personnaliser `cowsay` avec un *fichier de configuration* qui irait remplacer le dessin de vache par un nouveau dessin avec un autre personnage. Cela peut être accompli à l'aide de l'option `-f` de `cowsay`, qui permet d'indiquer à `cowsay` de lire un fichier de configuration spécifique.

Mais malheureusement le conteneur que vous avez créé est en **vase clos** : votre conteneur ne peut pas accéder aux fichiers de votre environnement de déploiement, et toutes les commandes que vous exécutez ne peuvent influencer que l'intérieur du conteneur.

Heureusement, nous avons vu qu'il était possible d'échanger des fichiers avec un conteneur à l'aide de la commande `podman container cp`. Voyons comment utiliser cette commande pour transférer un fichier vers notre conteneur Ubuntu.

1. Commencez par préparer un fichier texte appelé `r2d2.cow` sur votre poste de travail, et contenant exactement ceci :

   Fichier `r2d2.cow`

   ```
   $the_cow = <<EOC;
            $thoughts      ___
             $thoughts    /() \\
              $thoughts _|_____|_
               | | === | |
   ```

```
            |_|  O  |_|
             ||  O  ||
             ||__*__||
            |~ \\___/ ~|
            /=\\ /=\\ /=\\
            [_] [_] [_]
EOC
```

2. Ouvrez un nouveau terminal qui va vous servir à effectuer le transfert de fichier vers le conteneur, tout en gardant ouvert votre second terminal dans lequel vous êtes encore dans votre conteneur.
3. Transférez le fichier `r2d2.cow` dans un répertoire `~/tp-conteneurs/` sur l'environnement de déploiement à l'aide de l'outil `scp`.
4. Exécutez ensuite la commande suivante pour transférer ce fichier depuis l'environnement de déploiement, vers le répertoire `/root` au sein du conteneur Ubuntu que nous avons nommé `mon_premier_conteneur` lors de sa création :

   [sur l'environnement de déploiement distant]

   ```
   $ podman container cp ~/tp-conteneurs/r2d2.cow mon_premier_conteneur:/root/
   ```

Vous pouvez désormais vous rendre à nouveau dans le terminal que vous aviez laissé de côté dans lequel vous êtes dans le conteneur Ubuntu, et vous pouvez utiliser le fichier nouvellement ajouté dans le conteneur :

1. Vérifiez que le fichier est bien arrivé dans le répertoire `/root` :

   [sur le conteneur]

   ```
   # cd /root
   # ls
   ```

2. Exécutez `cowsay` en lui indiquant d'utiliser le fichier de configuration nouvellement ajouté au conteneur :

   [sur le conteneur]

   ```
   # /usr/games/cowsay -f /root/r2d2.cow "Beep. boop."
   ```

Vous obtenez normalement un superbe dessin en ASCII-art [https://fr.wikipedia.org/wiki/Art_ASCII] montrant R2-D2 [https://fr.wikipedia.org/wiki/R2-D2] s'exprimant à sa manière ◻.

5.3.4. Stopper, lister et supprimer le conteneur

Pour terminer l'exécution de l'interpréteur de commandes du conteneur, vous pouvez faire :

[sur le conteneur]

```
# exit
```

Le conteneur se retrouve alors *stoppé* – puisque son exécution est terminée – et vous arrivez de retour dans l'environnement d'accueil, à savoir le serveur de déploiement.

Mais le conteneur n'a pas disparu pour autant. Pour lister les conteneurs existants, on peut faire :

[sur l'environnement de déploiement distant]

```
$ podman container list --all
```

L'option `--all` permet d'afficher les conteneurs qui sont stoppés, et pas seulement les conteneurs en cours d'exécution.

On voit alors apparaître le conteneur Ubuntu avec pour état `Exited`.

On peut supprimer ce conteneur avec :

[sur l'environnement de déploiement distant]

```
$ podman container rm <identifiant>
```

Puis, si on reliste les conteneurs, il aura normalement disparu.

Nous venons ici d'illustrer le principe du *cycle de vie d'un conteneur*, tel que présenté dans le chapitre dédié à la création de conteneurs [https://gl.univ-nantes.io/developpement_exploitation/workbook/site/2026-2027/cms/creer-conteneurs/]. N'hésitez pas à aller y revoir la machine à états qui décrit les différents états et transitions possibles d'un conteneur.

5.4. Exercice : Asciidoctor depuis un conteneur Alpine

5.4.1. Énoncé

On se donne l'objectif d'exécuter un conteneur à partir de l'image officielle Alpine Linux [https://hub.docker.com/_/alpine], puis d'installer le logiciel Asciidoctor [https://asciidoctor.org/] au sein du conteneur.

Alpine Linux [https://www.alpinelinux.org/] est un système d'exploitation Linux différent de Ubuntu.

Asciidoctor est un logiciel qui permet de convertir un fichier écrit à l'aide du langage Asciidoc [https://asciidoc.org/] (similaire au langage *Markdown*), afin de produire un document HTML en sortie.

Exemple 1. Simple fichier Asciidoctor et rendu HTML correspondant

On écrit le fichier `financiers.adoc` avec le langage Asciidoc :

Contenu `financiers.adoc`

```
= Financiers aux amandes

== Ingrédients

- 200gr de yaourt végétal
```

```
- 1 pincée de sel
- 145g sucre
- 5gr de poudre à lever
- 60gr de huile de tournesol
- 100gr de farine T45
- 120gr de poudre d'amande
- amandes effilées
- margarine végétale

== Étapes

. Dans un récipient, fouettez le yaourt avec le sucre.
. Ajoutez l'huile, la farine, la poudre d'amandes, le sel et la poudre à lever.
La pâte obtenue doit être lisse.
. Graissez un moule à financiers avec un peu de margarine végétale et versez-y la
pâte.
Parsemez d'amandes effilées.
. Enfournez pour 20 à 25 minutes Ã 180 °C. Les financiers doivent avoir une belle
couleur dorée.
```

Si on utilise le logiciel Asciidoctor sur ce fichier, on obtient le HTML suivant (une fois ouvert dans un navigateur) :

Financiers aux amandes

Ingrédients

- 200gr de yaourt végétal nature
- 1 pincée de sel
- 145g sucre
- 5gr de poudre à lever
- 60gr de huile de tournesol
- 100gr de farine T45
- 120gr de poudre d'amande
- amandes effilées
- margarine végétale

Étapes

1. Dans un récipient, fouettez le yaourt avec le sucre.
2. Ajoutez l'huile, la farine, la poudre d'amandes, le sel et la poudre à lever. La pâte obtenue doit être lisse.
3. Graissez un moule à financiers avec un peu de margarine végétale et versez-y la pâte. Parsemez d'amandes effilées.
4. Enfournez pour 20 à 25 minutes Ã 180 °C. Les financiers doivent avoir une belle couleur dorée.

Last updated 2023-11-07 15:11:15 UTC

Asciidoctor s'utilise depuis un terminal à l'aide d'une commande qui peut prendre la forme suivante :

```
$ asciidoctor mondocument.adoc
```

et qui produit en résultat un fichier `mondocument.html`

Pour notre conteneur, on fait les choix suivants :

- On utilisera comme image de base l'image officielle Alpine disponible sur le Docker Hub [https://hub.docker.com/_/alpine]. En lisant la documentation de l'image, on en extrait les informations utiles suivantes :
  - L'image étant sur le Docker Hub, le registre de conteneur est accessible via l'adresse `docker.io`.
  - Le nom de l'image est simplement `alpine`.
  - Une étiquette valide pour cette image est `3.22`.
- On installera Asciidoctor simplement à l'aide de l'utilitaire `apk` disponible sur Alpine Linux. Avec `apk`, Asciidoctor peut être installé à l'aide du paquet appelé `asciidoctor`. `apk` s'utilise principalement avec deux actions :
  - `apk update` permet de récupérer la liste des paquets disponibles (similaire à `apt update`),
  - `apk add <nom du paquet>` permet d'installer un nouveau paquet (similaire à `apt install`).

Tout comme l'image Ubuntu, le point d'entrée de l'image Alpine est un interpréteur de commande, à savoir `sh`, qui a un comportement identique à Bash.

5.4.2. Questions

1. Préparez et exécutez une commande pour créer le conteneur Alpine souhaité (cf l'énoncé plus haut), avec les options nécessaires pour lui donner le nom `alpine_asciidoctor` et permettre les interactions clavier avec le conteneur. On utilisera l'image de conteneur accessible par l'étiquette `3.22`.
2. Une fois dans le conteneur, exécutez les commandes nécessaires pour installer `asciidoctor`. Vérifiez que l'outil est bien installé en l'exécutant avec l'option `--version`.
3. Sur votre poste de travail, préparez un fichier `financiers.adoc` contenant la recette de financiers montrée en exemple plus haut.
4. À l'aide d'un second terminal, copiez `financiers.adoc` au sein de votre conteneur.
5. Dans votre conteneur, exécutez `asciidoctor` de manière à convertir le fichier `financiers.adoc` en HTML. Utilisez `ls` pour vérifier que vous avez bien produit un fichier `financiers.html`.
6. À l'aide d'un second terminal, copiez le fichier `financiers.html` depuis conteneur, vers l'environnement de déploiement.

Pour copier un fichier *depuis* un conteneur, on utilise à nouveau `podman container cp`.

7. Vérifiez avec `ls` que vous avez bien récupéré le fichier qui était dans le conteneur.
8. Quittez le conteneur, puis supprimez le.
9. Selon vous, qu'est devenu le fichier `financiers.adoc` que vous aviez copié au sein du conteneur ?

5.5. Tutoriel : utiliser un conteneur pour déployer `replace_string`

Jusqu'ici nous avons surtout vu comment créer des conteneurs et installer des logiciels pré-existants dans ces conteneurs. Ces logiciels pre-existants étaient à chaque fois disponible sous forme de *paquets* fournis par les systèmes d'exploitations que nous avons considéré, et donc très faciles à installer.

Cependant, **que faire lorsqu'on développe fraichement un logiciel et qu'il n'est donc pas directement installable par un système d'exploitation ?** Dans la suite de ce TP on se concentrera sur des scénarios de ce type, dans lequels nous avons accès au code source et à la documentation technique d'un logiciel développé, mais où ce logiciel n'est pas disponible sous forme de paquet facilement installable dans un système d'exploitation.

Dans le présent tutoriel, nous allons refaire le déploiement de l'outil `replace_string` vu dans le TP précédent [https://gitlab.univ-nantes.fr/gl/developpement_exploitation/resources/small_programs/-/tree/main/1_replace_string]. Même si nous avions réussi à exécuter cet outil dans l'environnement de déploiement, nous allons voir ici *comment re-faire ce déploiement à l'aide d'un conteneur*. Ce tutoriel sera présenté en deux versions :

1. d'abord une version où on part *d'une image de base avec le système d'exploitation Ubuntu*,
2. ensuite une version où on part *d'une image de base avec Python pré-installé*.

5.5.1. Stratégie 1 : depuis une image de base Ubuntu

Le README de `replace_string` nous indique que la seule dépendance requise est un interpréteur Python en version 3. Il nous faut donc préparer un conteneur dans lequel cette dépendance est disponible.

Une première stratégie possible est de préparer un conteneur à partir d'une image de base contenant un système d'exploitation retreint, puis d'installer un interpréteur Python dans cet environnement, et enfin d'installer notre logiciel.

1. Commençons par créer un nouveau conteneur sur une base Ubuntu, avec les options `-ti` (équivaut à `--tty --interactive`) pour permettre les interactions au clavier :

   [sur l'environnement de déploiement distant]

   ```
   $ podman container run -ti --name replace_string_container docker.io/ubuntu:24.04
   ```

   Nous aurions pu choisir un autre système d'exploitation que Ubuntu, avec pour seule contrainte que Python doit être facilement installable dans le système d'exploitation choisi.

2. Dans ce conteneur, installons un interpréteur Python 3 à l'aide du gestionnaire de paquets `apt`, en sachant que le paquet Python 3 s'appelle `python3` :

   [sur le conteneur]

   ```
   # apt update
   # apt install -y python3
   ```

   Il est possible qu'apt vous demande de répondre à des questions concernant le

> fuseau horaire. Ces questions sont causées par l'installation d'une dépendance transitive de `python3` appelée `tzdata`, une bibliothèque pour gérer les fuseaux horaires. Vous pouvez répondre n'importe quoi à ces questions, cela n'impactera pas le TP.

3. À l'aide d'un second terminal, copiez `replace_string.py` dans le conteneur, ainsi qu'un fichier texte exemple qui servira à tester l'outil (reprenez par exemple `song.txt` du README, comme dans le TP précédent).
4. Dans le conteneur, testez `replace_string.py` sur votre fichier exemple.
5. Si tout est bon, quittez votre conteneur, puis supprimez-le une fois que vous avez terminé.

5.5.2. Stratégie 2 : depuis une image de base Python

Bien que la première stratégie fonctionne sans problème, **il est parfois possible de gagner du temps en choisissant l'image de base la plus adaptée**. Comme vu à la toute fin du chapitre 📎 Créer et manipuler des conteneurs [https://gl.univ-nantes.io/developpement_exploitation/workbook/site/2026-2027/cms/creer-conteneurs/], il existe des *images dédiées à des technologies*, c'est à dire qui viennent avec le nécessaire pour travailler avec un certain langage de programmation ou un certain framework logiciel.

Et nous avons de la chance : on peut trouver sur le registre Docker Hub une image de conteneur officielle pour le langage Python [https://hub.docker.com/_/python] ! Cette image possède déjà un interpréteur Python 3 pré-installé et pré-configuré, ce qui nous permet de sauter cette étape.

Refaisons l'exercice avec ce nouveau type d'image de base :

1. On crée un nouveau conteneur cette fois sur une base Python ; on utilise l'étiquette `3-slim` pour s'assurer d'avoir Python en version 3 dans une image légère, et on met `bash` comme point d'entrée explicite pour éviter de démarrer le conteneur dans l'interpréteur Python :

   [sur l'environnement de déploiement distant]

   ```
   $ podman container run -ti --name replace_string_container_v2 docker.io/python:3-slim
   bash
   ```

2. À l'aide d'un second terminal, copiez `replace_string.py` dans le conteneur, ainsi qu'un fichier texte exemple qui servira à tester l'outil (reprenez par exemple `song.txt` du README, comme dans le TP précédent).
3. Dans le conteneur, testez `replace_string.py` sur votre fichier exemple.
4. Si tout est bon, quittez votre conteneur, puis supprimez-le une fois que vous avez terminé.

5.6. Exercice : utiliser des conteneurs pour déployer différents programmes

5.6.1. `compress_file`

Vous devez déployer l'outil `compress_file` vu dans le TP précédent [https://gitlab.univ-nantes.fr/gl/developpement_exploitation/resources/small_programs/-/tree/main/2_compress_file].

Travail à effectuer:

1. Lisez bien à nouveau la documentation fournie dans le README.

2. Préparez un conteneur avec la dépendance requise (en faisant bien attention à la version demandée !), avec un nom et avec les options permettant les interactions clavier.
3. Dans votre conteneur, vérifiez que votre dépendance est bien installée (pensez à utiliser l'option `--version`).
4. Copiez `compress_file.py` sur le conteneur, puis testez-le sur un fichier de votre choix.
5. S'il n'y a pas d'erreur, c'est bon ! Quittez le conteneur puis supprimez-le lorsque vous avez terminé.

5.6.2. `print_colored_string`

Vous devez déployer l'outil `print_colored_string` vu dans le TP précédent [https://gitlab.univ-nantes.fr/gl/developpement_exploitation/resources/small_programs/-/tree/main/3_print_colored_string].

Travail à effectuer:

1. Lisez bien à nouveau la documentation fournie dans le README.
2. Préparez un conteneur, avec un nom, avec les options permettant les interactions clavier, et avec les dépendances requises, en sachant qu'une bibliothèque python peut être installée à l'aide de la commande:

```
# pip install <nom de la librarie>
```

3. Copiez `print_colored_string.py` sur le conteneur, puis testez-le sur une chaîne de caractères de votre choix.
4. S'il n'y a pas d'erreur, c'est bon ! Quittez le conteneur puis supprimez-le lorsque vous avez terminé.

5.6.3. `TicTacToe`

Vous devez déployer le jeu `TicTacToe` vu dans le TP précédent [https://gitlab.univ-nantes.fr/gl/developpement_exploitation/resources/small_programs/-/tree/main/4_tictactoe].

Quelques informations pour vous aider :

- Il existe une image de base `docker.io/eclipse-temurin` [https://hub.docker.com/_/eclipse-temurin] qui contient les outils de base pour le développement et l'exécution de programmes Java (commandes `javac` et `java` notamment).
- Une étiquette valide pour cette image est `21-jdk`, qui permet d'avoir Java en version 21 et avec tous les outils de développement (JDK).
- Cette image de base a pour point d'entrée l'interpréteur de commandes Java appelle `jshell`. Malheureusement, `jshell` ne permet pas d'aller exécuter un fichier Java spécifique, et est donc peu commode pour exécuter un conteneur de manière interactive.

Travail à effectuer:

1. Lisez bien à nouveau la documentation fournie dans le README.
2. Préparez un conteneur, avec un nom, avec les options permettant les interactions clavier, et avec la dépendance requise.

Si besoin, n'hésitez pas à démarrer votre conteneur avec un point d'entrée plus adapté à vos besoins, par exemple `bash`.

3. Dans votre conteneur, vérifiez que votre dépendance est bien installée (pensez à utiliser l'option `--version`).
4. Copiez `TicTacToe.java` sur le conteneur, puis testez-le.
5. S'il n'y a pas d'erreur, c'est bon ! Quittez le conteneur puis supprimez-le lorsque vous avez terminé.

5.6.4. `economic_dispatch`

Vous devez déployer l'outil `economic_dispatch` [https://gitlab.univ-nantes.fr/gl/developpement_exploitation/resources/small_programs/-/tree/main/5_economic_dispatch], qui n'a pas été vu au TP précédent.

Quelques informations pour vous aider :

- Il existe une image de base `docker.io/julia` [https://hub.docker.com/_/julia] qui contient les outils de base pour le développement et l'exécution de programmes Julia.
- Une étiquette valide pour cette image est `1.11-trixie`, qui permet d'avoir Julia en version 1.11.
- Cette image de base a pour point d'entrée l'interpréteur de commandes Julia, peu commode pour exécuter un conteneur de manière interactive.
- Il est possible d'installer une bibliothèque Julia avec la commande suivante:

```
# julia -e 'import Pkg; Pkg.add("<nom de la bibliothèque>")'
```

Travail à effectuer:

1. Lisez bien à nouveau la documentation fournie dans le README.
2. Préparez un conteneur, avec un nom, avec les options permettant les interactions clavier, et avec la dépendance requise.

À nouveau, si besoin, n'hésitez pas à démarrer votre conteneur avec un point d'entrée plus adapté à vos besoins, par exemple `bash`.

3. Dans votre conteneur, vérifiez que votre dépendance est bien installée (pensez à utiliser l'option `--version`).
4. Copiez `economic_dispatch.jl` sur le conteneur, ainsi qu'un fichier de test reprenant l'exemple donné dans le README, puis testez l'outil.
5. Exécutez l'outil à l'aide du fichier test, et observez que vous obtenez les mêmes résultats que ceux montrés dans le README.
6. S'il n'y a pas d'erreur, c'est bon ! Quittez le conteneur puis supprimez-le lorsque vous avez terminé.

5.7. Faire le ménage sur le serveur de déploiement

Une fois tous ces tutoriels et exercices terminés, il est bien de faire un peu de ménage sur l'environnement de déploiement ! Pour cela, référez-vous à la section dédiée dans la notice [notice-serveur#sec-infra-donnees].

Chapitre 6. TP : Montage de répertoires/fichiers

6.1. Présentation et objectifs

L'objectif de ce TP est d'apprendre à utiliser le montage de répertoire/fichier avec Podman pour plus facilement gérer les données de logiciels exécutés dans des conteneurs.

Comme pour le TP précédent, lorsque le sujet de TP vous propose d'exécuter une commande, il est demandé que vous arriviez à comprendre tout ce que comporte cette commande. **Prenez le temps de bien lire et décortiquer chaque commande qui vous est proposée**, et n'hésitez pas à poser des questions.

6.2. Connexion au serveur de déploiement

Comme pour le TP précédent, toute la suite du TP devra être effectuée sur l'environnement de déploiement. Vous pouvez donc commencer par ouvrir un terminal et ouvrir une session SSH à distance sur ce dernier, comme vous avez appris à le faire dans le Chapitre 4.

Pour rappel, il est tout à fait possible d'ouvrir plusieurs terminaux en parallèle. Vous pouvez ainsi, si nécessaire ouvrir plusieurs connexions vers le serveur de déploiement. Cela peut être utile par exemple pour lancer un conteneur dans un premier terminal (avec `run`), et l'observer (avec `ps` ou `logs`) ou le manipuler (avec `stop` ou `rm`) depuis un second. Vous pouvez également garder un autre terminal ouvert pour exécuter des commandes sur votre poste Linux.

6.3. Tutoriel : premiers montages dans un conteneur Ubuntu

6.3.1. Un premier montage de répertoire

Nous allons commencer ce TP de manière similaire au TP précédent, en créant un conteneur à partir d'une image Ubuntu. Cependant, cette fois-ci nous allons mettre en place un **montage de répertoire** permettant d'échanger facilement des données avec le conteneur.

Commençons par préparer un nouveau répertoire dans votre répertoire utilisateur au sein de l'environnement de déploiement, et ajoutons un fichier appelé `mon_fichier` dans ce répertoire :

[sur l'environnement de déploiement distant]

```
$ mkdir ~/tp-montage/tutoriel_1
$ echo "Ce fichier a été créé en dehors du conteneur" > ~/tp-
montage/tutoriel_1/mon_fichier
```

Ensuite on crée le conteneur de manière similaire à la dernière fois, mais en ajoutant cette fois-ci le montage :

[sur l'environnement de déploiement distant]

```
$ cd ~
$ podman container run --name conteneur_avec_montage \
                       --tty \
                       --interactive \
```

```
                    --mount type=bind,source="./tp-
montage/tutoriel_1",target="/repertoire_montage" \
                    docker.io/ubuntu:24.04
```

L'option `--mount` ici déclare trois choses :

- `type=bind` déclare le type de montage, à savoir un montage de répertoire/fichier (le seul type de montage étudié dans ce cours),
- `source="./tp-montage/tutoriel_1"` déclare que nous souhaitons monter le répertoire `./tp-montage/tutoriel_1` de l'environnement de déploiement,
- `target="/repertoire_montage"` déclare que le point de montage au sein du conteneur sera le répertoire `/repertoire_montage`.

Faites très attention lors de vos montages de répertoire : pensez comme montré ici à **toujours préparer un nouveau répertoire pour chacun de vos conteneurs** !

Cela vous évite d'une part de mélanger les données de vos différents déploiements, et vous évite d'autre part de partager par erreur un répertoire critique avec votre conteneur, ou bien entre différents conteneurs.

Par exemple, **il ne faut *surtout* pas monter l'intégralité de votre répertoire utilisateur au sein de votre conteneur** (en faisant par exemple `--mount type=bind,source="/comptes/EXXXXXX",target="/chemin/dans/mon/conteneur"`), car le logiciel s'exécutant dans le conteneur pourrait faire n'importe quoi à vos fichiers – par exemple en modifiant les droits d'accès de tous vos fichiers !

Une fois le conteneur créé et démarré, on obtient comme la fois précédente un interpréteur de commande interactif. On peut alors regarder les répertoires disponibles à la racine du système de fichiers :

[sur le conteneur]

```
# ls /
```

Parmi la liste des répertoires on voit apparaître `repertoire_montage`, le point de montage choisi dans l'option `--mount` !

On peut se rendre dans ce répertoire et constater que notre fichier est bien présent, avec le bon contenu :

[sur le conteneur]

```
# cd /repertoire_montage
# ls
# cat ./mon_fichier
```

6.3.2. Deux répertoires très très liés 🔗🔗

Vérifions maintenant que le répertoire `/repertoire_montage` du conteneur est bien *lié* au répertoire `~/tp-montage/tutoriel_1` de l'environnement de déploiement. Pour cela nous allons faire deux modifications dans ce répertoire *depuis le conteneur* :

1. Éditez le fichier `~/tp-montage/tutoriel_1/mon_fichier` pour y ajouter le texte de votre choix :

   [sur le conteneur]

   ```
   # nano ./mon_fichier
   ```

   La commande `nano` n'est pas installée par défaut dans un environnement Ubuntu : vous devez d'abord l'installer ! Le paquet `apt` correspondant s'appelle simplement `nano`.

2. Puis créez un nouveau fichier à côté de ce premier fichier :

   [sur le conteneur]

   ```
   # echo "Ceci est un second fichier" > ./mon_second_fichier
   ```

Observons maintenant les conséquences de ces changements. À l'aide d'un second terminal, rendez-vous dans le répertoire monté de l'environnement de déploiement, et vérifiez son contenu ainsi que celui des deux fichiers :

[sur l'environnement de déploiement distant]

```
$ cd ~/tp-montage/tutoriel_1
$ ls
$ cat ./mon_fichier
$ cat ./mon_second_fichier
```

Tous les changements effectués dans le conteneur sont bien apparents dans l'environnement de déploiement !

On pourrait à nouveau modifier le contenu du répertoire `~/tp-montage/tutoriel_1` depuis l'environnement de déploiement, constater que les changements apparaissent dans le répertoire `/repertoire_montage` du conteneur, et ainsi de suite…

En résumé : on observe bien de chaque côté le *même répertoire* accessible depuis les deux environnements.

6.3.3. Un montage de fichier

Il est également possible de faire un montage de *fichier* dans le conteneur. Créons un fichier à monter dans l'environnement de déploiement :

[sur l'environnement de déploiement distant]

```
$ echo "Ce fichier va être monté dans un conteneur" > ~/tp-
montage/tutoriel_1/mon_troisième_fichier
```

Puis créons un second conteneur en déclarant un montage avec ce nouveau fichier :

[sur l'environnement de déploiement distant]

```
$ cd ~
$ podman container run --name conteneur_avec_montage_2 \
                    --tty \
                    --interactive \
                    --mount type=bind,source="./tp-
montage/tutoriel_1/mon_troisième_fichier",target="/root/fichier_dans_conteneur" \
                    docker.io/ubuntu:24.04
```

Une fois dans le conteneur, on peut vérifier que le fichier monté est bien présent à l'emplacement du point de montage, avec le bon contenu :

[sur le conteneur]

```
# cd /root
# ls
# cat ./fichier_dans_conteneur
```

Et normalement tout est bon !

Pour aller plus loin, on peut modifier ce fichier depuis le conteneur (par exemple à l'aide de `nano`), et constater que ce changement est bien visible au sein du fichier `~/tp-montage/tutoriel_1/mon_troisième_fichier` de l'environnement de déploiement.

6.4. Ménage

Avant de passer aux exercices suivant, pensez à supprimer les conteneurs que vous avez créé dans ce tutoriel !

6.5. Exercice : faciliter l'usage du conteneur Asciidoc avec un montage

6.5.1. Énoncé

Dans cet exercice on se propose de reprendre l'exercice Asciidoc du TP précédent, mais cette fois-ci sans exécuter la commande `podman container cp` : on souhaite cette fois-ci reposer sur un *montage de répertoire* afin de passer des fichiers au conteneur, et afin de récupérer les fichiers générés au sein du conteneur.

Commencez par **relire l'énoncé** (qui reste le même pour cet exercice) avant de passer aux questions qui suivent.

6.5.2. Questions

1. Sur l'environnement de déploiement, préparez un répertoire vide que vous monterez dans le conteneur que vous allez créer.
2. Préparez et exécutez une commande pour créer un conteneur Alpine avec l'image de conteneur accessible par l'étiquette `3.22`, et avec les options nécessaires pour :
   - lui donner le nom `alpine_asciidoctor`,
   - permettre les interactions clavier avec le conteneur,
   - définir un *montage de répertoire* depuis le répertoire créé dans la question 1, vers un

répertoire de votre choix au sein du conteneur.

3. Une fois dans le conteneur, exécutez les commandes nécessaires pour installer `asciidoctor`. Vérifiez que l'outil est bien installé en l'exécutant avec l'option `--version`.
4. Sur votre poste de travail, préparez un fichier `financiers.adoc` contenant la recette de financiers montrée en exemple plus haut.
5. À l'aide d'un second terminal, déplacez le fichier `financiers.adoc` au sein du répertoire que vous avez préparé sur l'environnement de déploiement.
6. Dans votre conteneur, vérifiez que le fichier `financiers.adoc` est bien apparu dans votre point de montage, puis exécutez `asciidoctor` de manière à convertir le fichier en HTML. Utilisez `ls` pour vérifier que vous avez bien produit un fichier `financiers.html`.
7. À l'aide d'un second terminal, vérifiez sur l'environnement de déploiement que le fichier HTML produit est bien accessible dans le répertoire qui a été monté dans le conteneur.
8. Quittez le conteneur, puis supprimez le.
9. Cette fois-ci, qu'est devenu le fichier `financiers.adoc` que vous avez manipulé au sein du conteneur ?

6.6. Exercice : faciliter le déploiement de `economic_dispatch`

6.6.1. Énoncé

Dans cet exercice on se propose de reprendre l'exercice `economic_dispatch` du TP précédent, mais là aussi sans exécuter la commande `podman container cp` : on souhaite cette fois-ci reposer sur un *montage de répertoire* afin de passer le programme ainsi que le fichier de test dans le conteneur.

Relisez bien l'énoncé de l'exercice du TP précédent, et pour la suite n'hésitez pas à partir de la commande que vous aviez déjà préparé lorsque vous aviez fait cet exercice.

6.6.2. Questions

1. Sur l'environnement de déploiement, préparez un nouveau répertoire que vous monterez au sein de votre conteneur.
2. Dans ce répertoire, ajoutez le fichier `economic_dispatch.jl`, ainsi qu'un fichier de test permettant de tester l'outil (se référer au README).
3. Préparez un conteneur, avec un nom, avec les options permettant les interactions clavier, la dépendance requise par `economic_dispatch`, et avec un montage du répertoire préparé à la question précédente.

Comme dans le TP précédent, n'hésitez pas à démarrer votre conteneur avec un point d'entrée plus adapté à vos besoins, par exemple `bash`.

4. Dans votre conteneur, vérifiez que votre dépendance est bien installée (pensez à utiliser l'option `--version`).
5. Exécutez l'outil à l'aide du fichier test, et observez que vous obtenez les mêmes résultats que ceux montrés dans le README.
6. S'il n'y a pas d'erreur, c'est bon ! Quittez le conteneur puis supprimez-le lorsque vous avez terminé.

6.7. Exercice : déployer le logiciel *docker-ffmpeg-converter*

Jusqu'ici nous n'avons effectué que des déploiements de logiciels "jouets" permettant de manipuler les notions du cours en douceur. Dans ce dernier exercice, l'objectif est de **déployer un "vrai" logiciel existant**, qui n'a pas été préparé juste pour les objectifs de ce cours.

6.7.1. Énoncé

Nous souhaitons déployer le logiciel docker-ffmpeg-converter [https://github.com/KennethWussmann/docker-ffmpeg-converter]. Ce logiciel se charge de convertir automatiquement des vidéos dans un format souhaité à l'aide de l'outil linux `ffmpeg` [https://ffmpeg.org/], en lisant automatiquement et continuellement les vidéos d'un répertoire dit "source", et en produisant automatiquement les vidéos converties dans un autre répertoire dit "destination".

docker-ffmpeg-converter est similaire au cas d'étude *Automatic-Image-Converter* étudié en cours, mais convertit des *vidéos* et non pas des *images*.

Une image de conteneur a déjà été mise au point pour facilement déployer *docker-ffmpeg-converter*. En lisant la documentation de l'image [https://github.com/KennethWussmann/docker-ffmpeg-converter], on découvre qu'elle possède les caractéristiques suivantes :

- Le point d'entrée de l'image est l'exécution du logiciel *docker-ffmpeg-converter*.

Comme le point d'entrée n'est *pas* un interpréteur de commandes, il n'est pas pertinent de rendre interactif (via les options `--tty` et `--interactive`) un conteneur créé à partir de cette image. En effet, le processus de *docker-ffmpeg-converter* n'attend aucune entrée clavier pour pouvoir fonctionner.

- Lorsqu'il est exécuté au sein de cette image, *docker-ffmpeg-converter* est conçu pour comprendre notamment les variables d'environnement suivantes :
  - `SOURCE_DIRECTORY_PATH` : le répertoire surveillé par le logiciel, où les utilisateur·ices doivent déposer les fichiers vidéos qu'iels souhaitent convertir.
  - `DESTINATION_DIRECTORY_PATH` : le répertoire où le logiciel va écrire les vidéos une fois converties, et où les utilisateur·ices pourront récupérer le résultat.
  - `GLOB_PATTERNS` : liste de *motifs glob* [https://en.wikipedia.org/wiki/Glob_(programming)] qui définissent quels fichiers du répertoire surveillé sont pris en considération par le logiciel. Par exemple le motif glob `*.avi` fera que seuls les fichiers se terminant par l'extension `.avi` seront convertis.
  - `SCAN_INTERVAL` : le nombre de secondes que va attendre *docker-ffmpeg-converter* avant de surveiller à nouveau le répertoire surveillé.
  - `FFMPEG_ARGS` : arguments confiés à l'outil `ffmpeg` lorsqu'il est appelé par *docker-ffmpeg-converter* pour convertir chaque fichier vidéo découvert. Ces arguments doivent être conformes aux options permises et reconnues par `ffmpeg` [https://ffmpeg.org/ffmpeg.html#Options]. Si la valeur `%s` est utilisée dans un argument, *docker-ffmpeg-converter* la remplacera automatiquement par le nom du fichier d'entrée.

`ffmpeg` est un outil *très* complexe de traitement vidéo ; il ne vous est pas demandé de comprendre comment il fonctionne ou quelles options peuvent lui être confiées.

  - `LOG_FORMAT` : le format d'affichage des messages de logging affiché par le logiciel au cours de

son exécution. Peut prendre la valeur `json` pour un affichage en JSON [https://en.wikipedia.org/wiki/JSON], ou `text` pour un affichage en texte simple.

- L'image de conteneur de *docker-ffmpeg-converter* est disponible sur le registre `ghcr.io` (qui est celui de Github) sous le nom `kennethwussmann/docker-ffmpeg-converter`.
- Une étiquette valide pour cette image est `1.2.0`.

On souhaite réaliser un déploiement de *docker-ffmpeg-converter* sur l'environnement de déploiement de TP avec les contraintes suivantes :

- Le logiciel doit convertir tous les fichiers avec l'extension `.webm` déposés dans le répertoire `~/tp-montage/docker-ffmpeg-converter/inputs` de l'environnement de déploiement,
- Le logiciel doit déposer les fichiers convertis dans le répertoire `~/tp-montage/docker-ffmpeg-converter/outputs` de l'environnement de déploiement,
- On souhaite couvertir les fichiers d'entrée dans le format MP4, en fixant le nombre d'images par seconde à 24, et en produisant un fichier avec le suffixe `.mp4`, et avec des message de logging réduits ; pour accomplir cela on souhaite passer les arguments suivants à `ffmpeg` :

```
-loglevel error -y -fflags +genpts -i %s -r 24 %s.mp4
```

Comme déjà expliqué, vous n'avez pas besoin de comprendre en détail les arguments passés à `ffmpeg`.

- On souhaite que le logiciel vérifie la présence de nouveaux fichiers toutes les 5 secondes.
- On souhaite que les messages de logging soient affichés en texte simple

6.7.2. Questions

1. À l'aide d'une création de conteneur, effectuez le déploiement demandé dans l'énoncé.

Vous allez avoir besoin de confier des *variables d'environnement* à votre conteneur, afin de configurer le logiciel déployé. N'hésitez pas à vous référer au le chapitre sur l'exécution des logiciels [https://gl.univ-nantes.io/developpement_exploitation/workbook/site/2026-2027/cms/logiciel-execution/] pour revoir cette notion, et au le chapitre dédié à la création de conteneurs [https://gl.univ-nantes.io/developpement_exploitation/workbook/site/2026-2027/cms/creer-conteneurs/] pour revoir comment transmettre une variable d'environnement à un conteneur donné.

2. Constatez que le processus exécuté dans le conteneur affiche des messages de logging dans le terminal où vous avez créé le conteneur. Ces messages nous informent sur les actions réalisées par le logiciel.
3. Vérifiez que votre déploiement est conforme aux attentes :
   a. Sur votre poste de travail, téléchargez la vidéo WebM accessible via ce lien [https://upload.wikimedia.org/wikipedia/commons/8/87/Schlossbergbahn.webm] (une fois face à la vidéo, utilisez le raccourci clavier *Ctrl*+S de votre navigateur pour l'enregistrer).
   b. Transférez cette vidéo sur l'environnement de déploiement distant, puis copiez cette vidéo dans le répertoire `~/tp-montage/docker-ffmpeg-converter/inputs`
   c. Constatez dans les messages de log du logiciel que le fichier a été découvert et a été traité.

d. Constatez dans le répertoire `~/tp-montage/docker-ffmpeg-converter/inputs` de l'environnement de déploiement qu'une conversion en MP4 du fichier d'entrée est bien apparente.

e. Transférez le résultat de la conversion sur votre poste de travail, et ouvrez le pour vérifier que la vidéo convertie est bien la même que la vidéo d'origine.

Chapitre 7. TP : Publication de ports réseaux

Dans les TPs jusqu'ici, la seule manière que nous avons eu à disposition pour intéragir avec un logiciel exécuté au sein d'un conteneur était par le biais de *répertoires et fichiers*, que ce soit à l'aide de simples copies (`podman container cp`) ou bien à l'aide de montages (`--mount`).

Dans le présent TP, vous allez voir comment permettre des intéractions avec un logiciel conteneurisé en reliant un conteneur à l'interface réseau (et donc au réseau local) de l'environnement de déploiement, à l'aide de **publications de ports**.

7.1. Tutoriel : déployer un serveur web *nginx* avec publication de port

On se propose de déployer un serveur web *nginx* qui aura pour mission de publier des pages HTMLs simples. Pour cela, on souhaite utiliser l'image de conteneur *nginx* officielle [https://hub.docker.com/_/nginx], à savoir une image de conteneur qui a déjà été préparée avec le logiciel *nginx* et toutes ses dépendances.

7.1.1. Exécution avec publication de port

En lisant la documentation de l'image *nginx*, on observe deux choses utiles :

- Le point d'entrée de l'image est l'exécution d'un processus serveur serveur web *nginx*. Ainsi, si on exécute un conteneur à partir de l'image, cela démarrera directement un processus *nginx*.
- Un processus serveur web *nginx* écoute sur le port réseau `80`. Ainsi, pour que ce serveur web soit accessible depuis un navigateur exécuté sur votre poste de travail, il va être nécessaire d'effectuer une **publication de port** qui rende le conteneur accessible depuis le réseau local de l'université. Cette publication de port va permettre de relier un port réseau *de votre conteneur* (à savoir, dans le cas présent, le port `80`) à un port réseau de l'interface réseau *de son environnement d'accueil* – à savoir le serveur de déploiement.

Pour rappel, le serveur de déploiement est partagé par tous les étudiants de la formation. Ainsi, **il est nécessaire que chaque étudiant·e ne fasse des publications de port que sur les ports réseaux qui lui sont attribués !** Merci de lire attentivement la section correspondante de la notice serveur.

Préparez et exécutez la commande suivante afin de créer un conteneur *nginx* appelé `serveur_web_1`, et afin d'effectuer une publication de port sur le port `80` du conteneur :

[sur l'environnement de déploiement distant]

```
$ podman container run --rm \
                       --detach \
                       --name serveur_web_1 \
                       --publish <mon port réseau>:80 \
                       docker.io/nginx:1.25
```

en remplaçant `<mon port réseau>` par un port de votre choix de votre intervalle de port attribué.

On choisit d'utiliser l'option `--rm` pour que notre conteneur s'autodétruise à son arrêt, ainsi que l'option `--detach` pour ne pas attacher notre terminal au conteneur.

Pour vérifier que votre conteneur est bien en exécution, effectuez la commande suivante :

[sur l'environnement de déploiement distant]

```
$ podman container ps
```

Vous devez voir que :

- l'image utilisée est la bonne,
- le statut est `Up`,
- la publication de port est bien celle demandée,
- le nom du conteneur est bien `serveur_web_1`.

Et pour uniquement vérifier la liste des publications de ports du conteneur, on peut faire :

[sur l'environnement de déploiement distant]

```
$ podman container port serveur_web_1
```

Affichage de la publication de port

Supposons que votre publication du port `80` du conteneur `mon_conteneur` de fasse sur le port `7777` de l'environnement d'accueil. Si on utilise `podman container port mon_conteneur`, on verra alors apparaître :

```
80/tcp -> 0.0.0.0:8081
```

Décortiquons cela :

- `80/tcp` désigne le port réseau du conteneur que nous souhaitons publier. Le suffixe `/tcp` signifie que la publication de port est effectuée à l'aide du protocole réseau TCP. En effet, il est également possible de publier un port à l'aide du protocole UDP, mais cela ne sera pas traité non plus dans le cours.
- `0.0.0.0:8081` désigne le port réseau de l'environnement d'accueil sur lequel nous souhaitons faire la publication de port. La notation `0.0.0.0` signifie que la publication de port est effectuée sur *toutes les interfaces réseau* de l'environnement d'accueil. Il est en effet possible de restreindre une publication de port pour cibler à une ou plusieurs interfaces réseaux choisies, mais ce cas ne sera pas traité dans le cours.

Pour vérifier que le conteneur est bien en train de travailler, regardez les logs écrits par le conteneur à l'aide de la commande suivante :

[sur l'environnement de déploiement distant]

```
$ podman container logs serveur_web_1
```

Cela devrait vous afficher un grand nombre de lignes peu évidentes à comprendre, mais qui semblent

indiquer que le serveur est bien en train de fonctionner !

On peut désormais tester notre conteneur : sur votre poste informatique Linux, ouvrez le navigateur web Firefox, et accédez à l'adresse suivante : `http://podman:<mon port réseau>`, avec `<mon port réseau>` le même port que celui choisi pour la publication lors de la création du conteneur.

Dans cette adresse HTTP, on écrit `podman` car c'est le *nom d'hôte* de l'environnement de déploiement utilisé pour les TPs. Comme vos machines de TP sont sur un même réseau informatique que cet environnement de déploiement, elles peuvent le contacter via son nom d'hôte.

Vous devriez normalement accéder à la page par défaut qu'affiche un conteneur *nginx*, avec pour titre "*Welcome to nginx!*".

Si tout est OK, on peut désormais stopper et supprimer notre conteneur avant de passer à la suite :

[sur l'environnement de déploiement distant]

```
$ podman container stop serveur_web_1
```

Le conteneur ici est automatiquement supprimé suite au `stop`, car nous avions spécifié `--rm` lors de sa création.

7.1.2. Exécution avec montage de répertoire

En lisant la documentation de l'image *nginx*, on observe que, par défaut, la page HTML affichée est le fichier `/usr/share/nginx/html/index.html` situé à l'intérieur du conteneur. Afin d'afficher la page HTML de notre choix, il va être nécessaire d'effectuer un **montage de répertoire** qui nous permette de personnaliser le contenu de `/usr/share/nginx/html`.

Commencez par préparer une page HTML extrêmement simple à publier à l'aide de notre serveur web :

- Sur le serveur de déploiement, créez un répertoire `mon_site_web`.
- Dans ce répertoire, utilisez `nano` pour créer un fichier `index.html`, et mettez-y le contenu suivant :

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8">
    <title>Premier déploiement</title>
</head>

<body>
    <h1>Bienvenue !</h1>
    <p>Ce déploiement a été réalisé par XXXXXXXXXX</p>
    <p>Si vous voyez cette page, c'est que tout a bien fonctionné.</p>
</body>

</html>
```

en remplaçant `XXXXXXXXXXX` par votre nom (ou vos noms, si le travail est fait à plusieurs).

Vous pouvez personnaliser le contenu de cette page à loisir, elle sert simplement de prétexte pour ce tutoriel.

- Enfin, effectuez la commande `pwd` pour afficher le *chemin absolu* du répertoire `mon_site_web`, et notez ce chemin pour l'étape suivante.

Maintenant que le contenu du site est prêt, préparez et exécutez la commande suivante afin de créer à nouveau un conteneur *nginx* similaire au précédent, mais avec le montage de répertoire souhaité :

[sur l'environnement de déploiement distant]

```
$ podman container run --rm \
                       --detach \
                       --name serveur_web_2 \
                       --publish <mon port réseau>:80 \
                       --mount type=bind,source="<chemin absolu mon_site_web>
",target="/usr/share/nginx/html" \
                       docker.io/nginx:1.25
```

avec `<chemin absolu mon_site_web>` le chemin absolu vers le répertoire `mon_site_web`, obtenu précédemment à l'aide de la commande `pwd`.

Testons ce nouveau conteneur : sur votre poste informatique Linux, utilisez Firefox pour accéder à nouveau à l'adresse : `http://podman:<mon port réseau>`. Vous devriez normalement accéder à une page avec pour titre "*Bienvenue* !" et avec le contenu que vous aviez écrit dans le fichier `index.html`. **Si oui, bravo, vous avez déployé votre premier site web à l'aide d'un conteneur !**

Pour faire un bon rappel du fonctionnement d'un montage de répertoire avec `--mount`, vous pouvez modifier le fichier `index.html` dans l'environnement d'accueil *alors que votre conteneur est toujours en exécution*, puis rafraîchir la page web affichée dans Firefox sur votre poste Linux : le changement sera instantanément visible ! En effet, le répertoire monté existe simultanément dans l'environnement d'accueil et dans le conteneur, ainsi un changement effectué dans l'un est directement visible dans l'autre.

Pour la suite, ne stoppez pas et ne supprimez pas tout de suite votre conteneur `serveur_web_2`.

7.1.3. Exécution de commandes dans un conteneur déjà en exécution

On souhaite comprendre ce qu'il se passe au sein du conteneur `serveur_web_2` créé dans la section précédente. Pour cela, exécutez la commande suivante :

[sur l'environnement de déploiement distant]

```
$ podman container exec -ti serveur_web_2 bash
```

La notation `-ti` est la combinaison des options `-t` (version abrégée de `--tty`) et `-i` (version abrégée de `--interactive`). Pratique dès qu'on souhaite créer une session

> interactive.

Suite à cette commande, on se retrouve dans une session interactive de l'interpréteur de commandes Bash au sein du conteneur `serveur_web_2`, et tout cela sans avoir interrompu l'exécution du processus *nginx*. On peut alors par exemple aller voir le conteneur du répertoire `/usr/share/nginx/html` du conteneur :

[sur le conteneur]

```
# ls /usr/share/nginx/html
```

et ainsi vérifier qu'il contient bien un fichier `index.html`.

Puis, on peut afficher le contenu de ce fichier `index.html` :

[sur le conteneur]

```
# cat /usr/share/nginx/html/index.html
```

et constater que c'est bien le contenu du fichier HTML que nous avions préparé initialement dans le répertoire `mon_site_web` de l'environnement d'accueil !

Vous pouvez désormais quitter cette session interactive dans le conteneur avec `exit`, puis stopper et supprimer le conteneur `serveur_web_2`.

> Le répertoire `mon_site_web` ne sera pas supprimé lors de la suppression de votre conteneur, puisqu'il était simplement rendu disponible par un montage de répertoire.

7.2. Exercice : déployer un service Gokapi avec publication de port

7.2.1. Énoncé

On souhaite déployer une instance de Gokapi [https://github.com/Forceu/gokapi], un logiciel libre permettant de fournir un service d'envoi de fichiers par URL.

On trouve sur le Docker Hub l'image de conteneur Gokapi officielle [https://hub.docker.com/r/f0rc3/gokapi]. En lisant la documentation Gokapi [https://gokapi.readthedocs.io/], on découvre les informations utiles suivantes :

- L'image étant sur le Docker Hub, le registre de conteneur est accessible via l'adresse `docker.io`.
- Le nom de l'image est `f0rc3/gokapi`.
- Le point d'entrée de l'image est l'exécution du serveur web *Gokapi*.
- Ce serveur web écoute par défaut sur le port `53842` du conteneur.
- Le logiciel stocke ses données dans deux répertoires différents *au sein du conteneur* :
  - le répertoire `/app/data`, dans lequel les fichiers et les métadonnées sont stockés,
  - le répertoire `/app/config`, dans lequel la configuration de *Gokapi* est conservée.
- Lors de son tout premier lancement (et *uniquement* lors de son premier lancement), un service *Gokapi* affiche une page d'accueil dédiée uniquement à l'administrateur·ice système, et qui

permet à la fois de créer un compte d'administration et de saisir facilement toutes les variables de configuration souhaitées.

7.2.2. Questions

1. Préparez et exécutez une commande pour créer un conteneur *Gokapi*, en suivant avec les contraintes suivantes:
   - Le nom du conteneur doit être `gokapi_instance`,
   - L'image de conteneur utilisée doit être celle correspondant à l'étiquette `v2.1.0`,
   - L'application doit être accessible par une publication de port,
   - Toutes données créées et stockées par le logiciel doivent être persistées dans l'environnement d'accueil. Autrement dit, on doit pouvoir être capable de sereinement détruire et recréer le conteneur utilisé pour le déploiement, sans perdre la moindre donnée.
2. Une fois le conteneur en exécution, utilisez Firefox depuis votre poste de travail Linux pour accéder à la page d'accueil de l'application, en utilisant le nom d'hôte du serveur de déploiement (`podman`) ainsi que la publication de port que vous avez réalisée, et en ajoutant `/setup` à la fin de votre URL. Cela devrait ressembler à :

```
http://podman:<port>/setup
```

3. Configurez le service que vous avez déployé à l'aide de la page d'accueil de premier lancement. Laissez les valeurs par défaut pour toutes les questions, SAUF pour les questions suivantes :
   - Page *Webserver 2/2* :
     - *Public Name* : `Gokapi de NOM PRENOM` (à remplacer avec votre nom et prénom)
     - *Public Facing URL*: `http://podman:<port>`, en remplaçant `<port>` par votre publication de port.

La valeur que vous mettez ici doit normalement correspondre à celle dans votre barre d'adresse Firefox, sans le `/setup/start`.

   - Page *Credentials*:
     - *Username* : `admin`
     - *Password* : `my_password`
4. Testez l'application :
   a. Téléchargez sur votre poste de travail le logo du cours [https://gl.univ-nantes.io/developpement_exploitation/site/2026-2027/cms/_images/logo.drawio.png].
   b. Ouvrez la page d'accueil de votre instance *Gokapi*, puis authentifiez-vous avec le compte administrateur.
   c. Cliquez sur le (très gros) bouton "*Drop files, paste or click here to upload*", puis sélectionnez le fichier logo téléchargé précédemment. Le fichier sera téléversé dans l'espace *Gokapi* du compte `admin`, et sera visible dans la liste de fichiers affichée dans la page d'accueil.
   d. Vérifiez que vous pouvez correctement télécharger votre fichier : cliquez sur *l'ID* de votre fichier pour ouvrir une page avec un bouton "*Download*" permettant de le télécharger.
   e. Stoppez et supprimez votre conteneur.

f. Recréez votre conteneur avec la même commande que dans la première étape.

g. Connectez-vous, et vérifiez que le compte administrateur fonctionne toujours, et que le fichier que vous avez déposé est toujours là. Si oui, tout est OK ! Dans le cas contraire, avez-vous bien vérifié votre *montage de répertoire* ... ?

Si tout s'est bien passé, vous pouvez stopper et supprimez votre conteneur avant de passer à la suite.

7.3. Exercice : déployer un service Kanboard avec publication de port

7.3.1. Énoncé

On souhaite déployer une instance de Kanboard [https://kanboard.org/], un logiciel libre reposant sur la méthode Kanban [https://fr.wikipedia.org/wiki/Kanban_(d%C3%A9veloppement)] pour faire de la gestion de projet.

Pour simplifier le déploiement, on souhaite reposer sur la possibilité offerte par *Kanboard* de stocker ses données sous la forme d'un fichier SQLite [https://fr.wikipedia.org/wiki/SQLite], afin d'éviter d'avoir à gérer un système de gestion de base de données externe. C'est d'ailleurs le mode de fonctionnement par défaut de *Kanboard*.

Nous verrons plus loin dans le TP un exercice avec un système de gestion de base de données externe.

On trouve sur le Docker Hub l'image de conteneur Kanboard officielle [https://hub.docker.com/r/kanboard/kanboard]. En lisant la documentation Kanboard [https://docs.kanboard.org/v1/admin/docker/], on découvre les informations utiles suivantes :

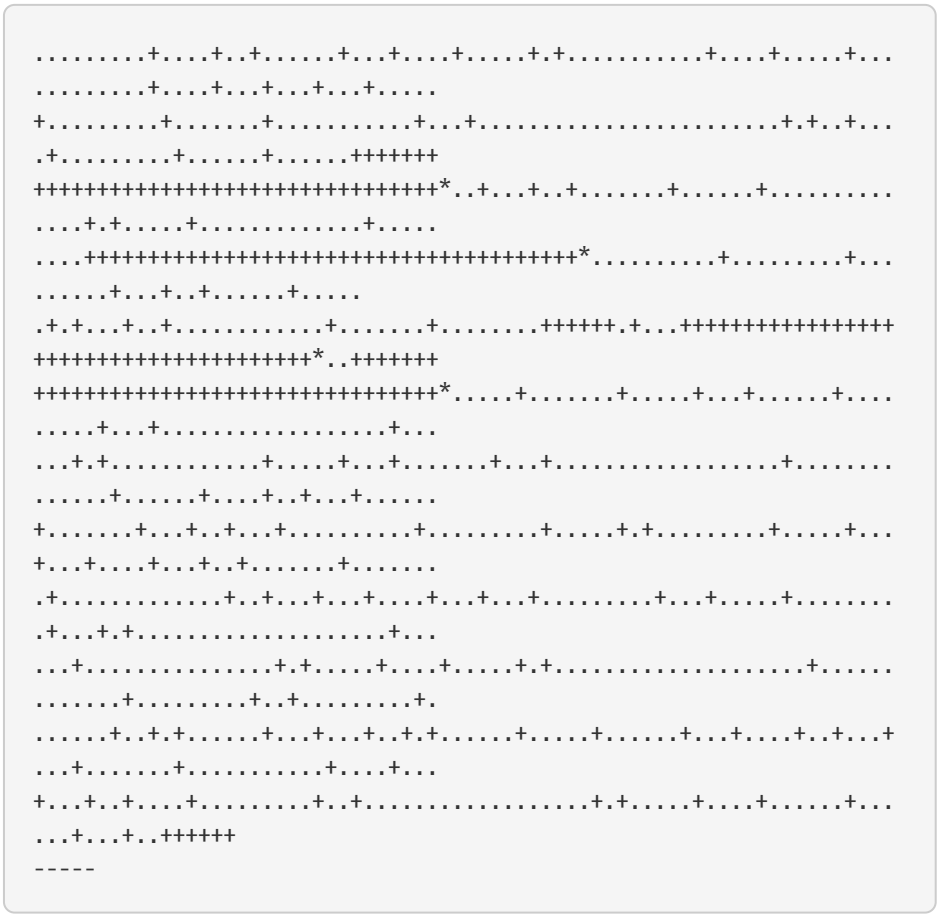
- L'image étant sur le Docker Hub, le registre de conteneur est accessible via l'adresse `docker.io`.
- Le nom de l'image est `kanboard/kanboard`.
- Le point d'entrée de l'image est l'exécution du serveur web *Kanboard*,
- Ce serveur web écoute par défaut sur le port `80` du conteneur,
- Le logiciel stocke ses données dans le répertoire `/var/www/app/data` du conteneur.

7.3.2. Questions

1. Préparez et exécutez une commande pour créer un conteneur *Kanboard*, en suivant avec les contraintes suivantes:
   - Le nom du conteneur doit être `kanboard_instance`.
   - L'image de conteneur utilisée doit être celle correspondant à l'étiquette `v1.2.32`
   - L'application doit être accessible par une publication de port,
   - Les données de l'application doivent être stockées dans l'environnement d'accueil.

Relisez bien cette note qui concerne le montage de répertoire.

Kanboard est un logiciel très peu bavard, et qui donc ne dit presque rien à son démarrage. Ainsi, si vous regardez les logs de votre conteneur (soit en exécutant sans `--detach`, soit à l'aide de `podman container logs`), vous verrez simplement apparaître quelque chose comme ceci :



Cette étrange séquence de caractères signifie simplement que Kanboard génère une clé cryptographique à son démarrage, et donc que tout fonctionne bien. La présence de la ligne `-----` à la toute fin signifie que Kanboard est près à fonctionner, et que vous pouvez donc passer à la question suivante.

2. Une fois le conteneur en exécution, utilisez Firefox depuis votre poste de travail Linux pour accéder à la page d'accueil de l'application, en utilisant la publication de port que vous avez réalisée.
3. Testez l'application :
   a. Par défaut, l'application met à disposition un compte administrateur avec pour nom d'utilisateur `admin` et pour mot de passe `admin`. Connectez-vous à l'aide de ce compte.
   b. Utilisez l'interface pour créer un nouveau projet, et donnez lui le nom de votre choix.
   c. Une fois le projet créé, allez dans l'onglet *Board*, et utilisez l'interface pour créer un ticket dans la colonne *Backlog* avec le contenu de votre choix.
   d. Une fois le ticket créé, déplacez ce ticket vers la colonne *Ready*.
4. Si vous êtes arrivé jusqu'ici, c'est que le déploiement est déjà un succès ! On souhaite tout de même vérifier que vous avez bien configuré correctement le stockage des données, à l'aide des actions suivantes :
   a. Stoppez et supprimez votre conteneur.
   b. Recréez votre conteneur avec la même commande que dans la première étape.
   c. Connectez-vous, et vérifiez que le ticket que vous aviez créé est toujours là. Si oui, tout est OK ! Dans le cas contraire, avez-vous bien vérifié votre *montage de répertoire* … ?

Si tout s'est bien passé, vous pouvez stopper et supprimez votre conteneur avant de passer à la suite.

7.4. Exercices additionnels

Pour ceux qui auraient terminé ce travail avant le temps alloué en séance, voici quelques exercices supplémentaires moins guidés et plus "libres" que ceux demandés précédemment.

7.4.1. Déploiement d'un serveur de discussion IRC

IRC est un protocole permettant à des logiciels clients IRC de communiquer avec des logiciels serveur IRC, afin de permettre à de nombreux utilisateurs connectés au même serveur de communiquer entre eux par messagerie instantanée. Il existe de très nombreux logiciels clients et serveurs compatibles avec le protocole IRC.

L'exercice est ici pour vous de déployer un logiciel serveur IRC InspIRCd [https://www.inspircd.org/] sur le serveur de déploiement. Vous utiliserez l'image docker officielle, disponible sur le Docker Hub [https://hub.docker.com/r/inspircd/inspircd-docker/]. Vous vous assurerez que les données du serveur sont bien stockées dans l'environnement d'accueil, et que votre serveur soit accessible via un de vos ports personnels. Pour l'essentiel, vous pouvez garder les options de configuration de base de InspIRCd.

Pour vérifier que votre déploiement fonctionne, vous pouvez télécharger sur votre poste de travail la commande `kirc` (*KISS for IRC*) à l'aide de ce lien [https://gitlab.univ-nantes.fr/gl/developpement_exploitation/resources/kirc-mirror/-/jobs/artifacts/master/raw/kirc?job=build]. C'est un client IRC minimaliste capable de se connecter à un serveur IRC. Pour se connecter à votre serveur, vous pouvez vous baser sur ce format de commande :

[sur votre poste de travail local]

```
$ ./kirc -s <nom d'hôte> -p <port> -c <canal> -n <pseudo>
```

avec :

- `<nom d'hôte>` : le nom d'hôte auquel se connecter,
- `<port>` : le port auquel se connecter sur l'hôte ciblé,
- `<canal>` : le nom d'un canal auquel se connecter au sein du serveur IRC (si le canal n'existe pas, il sera créé),
- `<pseudo>` : le pseudo à utiliser pour envoyer des messages.

Si tout fonctionne, vous devriez avoir un retour texte assez verbeux de la part du serveur, puis vous devriez arriver face à un invite de la forme :

```
nom_du_canal>
```

à partir duquel vous pouvez envoyer des messages texte.

Pour pousser vos tests plus loin, vous pouvez tester un *même* serveur IRC à plusieurs en vous connectant dessus avec différents postes Linux simultanément, et en vérifiant que vous arrivez à échanger des messages sur un même canal de discussion !

7.4.2. Déploiement de Automatic-Image-Converter

Faites un déploiement par vous même d'une instance de Automatic-Image-Converter [https://gitlab.univ-nantes.fr/gl/developpement_exploitation/resources/automatic-image-converter].

Vous pouvez bien entendu vous baser sur les commandes déjà vues pendant le cours.

Sur la page du projet, vous pouvez accéder à l'ensemble des images de conteneurs disponibles en vous rendant dans *Déploiement → Registre de conteneur*.

Chapitre 8. 💻 TP : Conteneuriser (manuellement) un logiciel

L'objectif de ce TP est d'apprendre à utiliser Podman pour transformer un conteneur en une *image de conteneur*, et pour *publier une image* au sein d'un registre de conteneur distant.

8.1. Préparatifs

8.1.1. Création d'un projet Gitlab

Les premiers énoncés de ce TP nécessitent d'être réalisés au travers d'un projet sur le Gitlab de l'Université. Reprenez la notice Gitlab et suivez les instructions pour créer un nouveau projet privé avec pour nom `developpement_exploitation-tp-images`.

Ce projet vous permettra de publier vos images de conteneurs au sein du registre de conteneur du Gitlab de l'université.

Pour des raisons pratiques liées à l'infrastructure disponible à l'Université, on utilisera le serveur de déploiement (`podman`) pour créer des images de conteneurs. Cependant, on notera qu'en pratique il est bien plus courant de créer une image directement depuis poste de travail utilisé pour le développement, ou bien à l'aide d'un serveur d'intégration continue [https://fr.wikipedia.org/wiki/Int%C3%A9gration_continue] dédié à ce genre de tâches.

8.1.2. Authentification Podman pour le registre de conteneurs de l'Université

Ce TP va vous apprendre à publier vos images sur le registre de conteneurs du Gitlab de l'Université. Cela nécessite au préalable de donner à Podman vos informations de connexion pour qu'il puisse se connecter au Gitlab.

Pour cela, depuis le serveur de déploiement, faites :

```
$ podman login docker-registry.univ-nantes.fr
```

Puis saisissez votre identifiant et votre mot de passe que vous utilisez pour vous connecter habituellement sur Gitlab, et validez.

Cette commande devra être re-exécutée au début de chaque TP, car une fois effectuée l'authentification n'est valide que durant la session en cours !

8.2. Tutoriel : une première image Ubuntu avec `cowsay`

8.2.1. Rappel du précédent tutoriel Ubuntu

Pour débuter, nous allons repartir du premier tutoriel du second TP. Pour rappel, nous avions créé un conteneur à partir d'une image Ubuntu, puis nous avions exécuté un interpréteur de commandes dans ce conteneur afin d'installer un logiciel appelé `cowsay` à l'aide des commandes suivantes :

```
$ apt update
```

```
$ apt install cowsay
```

Un inconvénient de cette approche est que si l'on détruit le conteneur ainsi obtenu, alors on perdra au passage l'installation de logiciel réalisée. Ainsi, si on devait recréer un conteneur identique, il faudrait alors à nouveau effectuer manuellement cette installation.

8.2.2. Création d'une image de conteneur `cowsay`

L'objectif ici est de créer une *image de conteneur* dans laquelle le logiciel souhaité (à savoir `cowsay`) est déjà installé. Ainsi, tout conteneur créé à partir de cette image possédera directement le logiciel souhaité.

Tout d'abord, préparons un nouveau conteneur à partir d'une image de base, à savoir l'image `docker.io/ubuntu:24.04` (la même que dans le premier tutoriel) :

[sur l'environnement de déploiement distant]

```
$ podman container run -ti --name cowsay_template \
                       docker.io/ubuntu:24.04
```

Nous avons nommé notre conteneur `cowsay_template` pour bien signifier que notre conteneur est un *template* (c'est à dire un *gabarit* ou un *patron*) qui sera ensuite transformé en une nouvelle image de conteneur.

Au sein de ce nouveau conteneur, on installe la commande `cowsay` :

[sur le conteneur]

```
$ apt update
$ apt install cowsay
```

Enfin, on peut quitter le conteneur (ce qui l'arrête au passage) :

[sur le conteneur]

```
$ exit
```

Il ne nous reste plus qu'à créer une nouvelle image de conteneur à partir de notre conteneur *template* :

[sur l'environnement de déploiement distant]

```
$ podman container commit cowsay_template \
                          tutoriel-cowsay:v1
```

Ici on a choisi de nommer notre nouvelle image de conteneur `tutoriel-cowsay`, et de lui donner l'étiquette `v1`.

Une fois l'image construite, vérifiez qu'elle est bien présente dans votre stockage d'image local :

[sur l'environnement de déploiement distant]

```
$ podman image list
```

On notera que la colonne REPOSITORY indique la référence d'image, qui dans notre cas commence par localhost puisque l'image est sur le stockage local.

Il est possible que vous ayez également dans cette liste un reliquat d'images téléchargées dans les précédents exercices, ce qui est normal.

Il ne nous reste plus qu'à tester notre nouvelle image, on peut donc faire :

[sur l'environnement de déploiement distant]

```
$ podman container run --rm -ti \
                       localhost/tutoriel-cowsay:v1
```

On se retrouve dans un interpréteur de commandes au sein du conteneur. On peut alors directement exécuter la commande souhaitée, sans avoir à l'installer au préalable :

[sur le conteneur]

```
$ /usr/games/cowsay "Bonjour depuis notre première image de conteneur !"
```

8.2.3. Publication de l'image

On souhaite publier notre image sur le registre de conteneurs associé au Gitlab de l' Université, dans le but (hypothétique) de pouvoir réutiliser cette image pour différents déploiements.

Dans un registre de conteneurs Gitlab, une image ne peut être stockée que au sein d'un projet déjà créé. Nous allons ici utiliser le projet créé dans la Section 8.1.1 (TODO).

Pour publier votre image, préparez et exécutez la commande suivante :

[sur l'environnement de déploiement distant]

```
$ podman image push localhost/tutoriel-cowsay:v1 \
                    docker-registry.univ-nantes.fr/<chemin projet>/tutoriel-cowsay:v1
```

où <chemin projet> est le chemin de votre projet au sein de l'arborescence Gitlab. Si vous avez le compte étudiant eXXXXX et que vous avez nommé votre projet developpement_exploitation-tp-images, alors votre chemin de projet aura la forme eXXXXX/developpement_exploitation-tp-images.

Il est nécessaire de mettre votre compte étudiant en *minuscules* pour que le chemin cible de publication d'image fonctionne.

Dans cette commande, on observe que la dernière partie de la cible de publication reprend le nom et l'étiquette de l'image (tutoriel-cowsay:v1) de manière identique au nom de l'image que nous avions créée dans le stockage local. En effet, le souhait ici est de garder le même nom et la même étiquette sur le registre distant. On

> notera cependant que ce n'est pas forcément le cas : on peut très bien publier une image sous un nom et/ou une étiquette différente de son nom et son étiquette du stockage local.

Pour vérifier que la publication a bien fonctionné, utilisez un navigateur pour vous rendre sur la page du projet Gitlab, et rendez-vous dans la section *Déploiement → Registre de conteneur*. La liste doit contenir un élément correspondant aux images nommées `tutoriel-cowsay`. En cliquant sur cet élément, on accède à la liste des étiquettes, qui doit contenir simplement l'étiquette `v1`.

Pour tester votre image publiée, on peut de nouveau exécuter un conteneur mais avec une référence d'image correspondant à l'image distante :

[sur l'environnement de déploiement distant]

```
$ podman container run --rm -ti \
                       docker-registry.univ-nantes.fr/<chemin projet>/tutoriel-cowsay:v1
```

Et vous devriez avoir le même résultat que lors de l'exécution de l'image avec la référence locale.

8.2.4. Ajout d'un point d'entrée

On souhaite maintenant améliorer notre image en modifiant quel logiciel est exécuté lors du démarrage d'un conteneur créé avec l'image. Cela nécessite de spécifier un *point d'entrée* pour l'image à l'aide de la métadonnée `CMD`.

> Comme la `v1` de notre image ne spécifiait pas de point d'entrée, elle héritait par défaut de celui spécifié dans l'image de base (`docker.io/ubuntu:24.04`), ce qui correspond ici à l'interpréteur de commandes `bash`. C'est pour cela que la commande `podman container run` a directement démarré un interpréteur de commande à partir de notre image.

Pour ce tutoriel, on souhaite que la commande `cowsay` soit exécutée au démarrage d'un conteneur. **Problème : cette commande nécessite un argument, à savoir le message à afficher**. *Comment faire pour qu'un conteneur créé à partir de notre image puisse recevoir le message à afficher ?*

Comme on a pu le voir dans le TP précédent, la technique la plus courante pour confier des informations à un conteneur est sous la forme de *variables d'environnement* passées à l'aide de l'option `--env` lors de la création du conteneur.

Si on revient à la création de l'image, il est possible dans une métadonnée `CMD` de faire référence à une variable d'environnement avec la syntaxe `$VARIABLE`.

On va donc re-créér notre image de conteneur en ajoutant cette métadonnée :

[sur l'environnement de déploiement distant]

```
$ podman container commit cowsay_template \
                          --change 'CMD /usr/games/cowsay $MESSAGE' \
                          tutoriel-cowsay:v2
```

On peut vérifier que l'image est bien créée à nouveau à l'aide de la commande :

```
$ podman image list
```

On peut voir que les deux images créées sont présentes dans le stockage d'image local, la première avec l'étiquette `v1`, la seconde avec l'étiquette `v2`.

On peut ainsi désormais exécuter un conteneur de la sorte :

```
$ podman container run --rm \
                       --env MESSAGE="Re-bonjour depuis notre première image !" \
                       localhost/tutoriel-cowsay:v2
```

Et voilà notre première image paramétrable lors de son exécution en conteneur !

8.2.5. Re-publication de l'image

Pour finir, on publie cette deuxième version de notre image aux côtés de la première version :

```
$ podman image push localhost/tutoriel-cowsay:v2 \
                    docker-registry.univ-nantes.fr/<chemin projet>/tutoriel-cowsay:v2
```

Allez de nouveau vérifier la liste des images au sein de la page de votre projet, et vous devez constater qu'il a désormais deux étiquettes pour les images nommées `tutoriel-cowsay`.

8.3. Exercice : image Alpine avec Asciidoctor

8.3.1. Énoncé

On souhaite préparer une image de conteneur qui permette d'exécuter facilement le logiciel Asciidoctor [https://asciidoctor.org/]. Asciidoctor est un logiciel qui permet de convertir un fichier écrit à l'aide du langage Asciidoc [https://asciidoc.org/] (similaire au langage *Markdown*), afin de produire un document HTML en sortie.

Asciidoctor s'utilise depuis un terminal à l'aide d'une commande qui peut prendre la forme suivante :

```
$ asciidoctor mondocument.adoc
```

et qui produit en résultat un fichier `mondocument.html`

Pour notre image de conteneur, on fait les choix suivants :

- On utilisera comme image de base l'image officielle Alpine disponible sur le Docker Hub [https://hub.docker.com/_/alpine], en version 3.22.
- On installera Asciidoctor simplement à l'aide de l'utilitaire `apk` disponible sur Alpine Linux. Avec `apk`, Asciidoctor peut être installé à l'aide du paquet appelé `asciidoctor`. Comme déjà présenté dans le TP précédent, `apk` s'utilise principalement avec deux actions :
  - `apk update` permet de récupérer la liste des paquets disponibles (similaire à `apt update`),
  - `apk add <nom du paquet>` permet d'installer un nouveau paquet (similaire à `apt install`).

En outre, on souhaite que le point d'entrée de notre image soit l'exécution de la commande `asciidoctor` sur un fichier situé dans un répertoire `/data` situé dans le conteneur.

L'image de conteneur produite devra avoir pour nom `exercice-asciidoctor` et pour étiquette `1.0`.

8.3.2. Questions

1. Faites un travail préliminaire de *conception* de votre future image de conteneur en vous posant notamment les questions suivantes :
   - est-il nécessaire de pouvoir *configurer* l'exécution du logiciel conteneurisé (à savoir Asciidoctor) ? si oui, quel *mode de configuration* est possible et pourrait être mis en place ?
   - quel *point d'entrée* définir pour l'image ? comment articuler ce point d'entrée avec le mode de configuration choisi au point précédent ?
   - est-ce qu'un *montage* sera nécessaire lors de la phase d'usage de l'image ? lequel et pourquoi ? comment rendre cohérent ce point de montage avec le mode de configuration et avec le point d'entrée définis précédemment ?

Pour vous aider, vous pouvez également vous poser les questions suivantes :

- Comment permettre à Asciidoctor de lire le fichier que l'on souhaite convertir en HTML ?
- Comment permettre à Asciidoctor de savoir quel fichier convertir ?

2. Préparez un conteneur *template* dans lequel vous installez tout le nécessaire pour répondre au besoin de l'énoncé. Pour cela, vous pouvez d'abord vous poser les questions suivantes :
   a. Quelle est la référence de l'image de base à utiliser ?
   b. Quelles commandes faut-il exécuter au sein du conteneur pour installer Asciidoctor ?
3. En vous basant sur votre travail préliminaire de conception, transformez votre conteneur *template* en image de conteneur.
4. Vérifiez que l'image créée est bien présente dans votre stockage local d'images.
5. Testez votre nouvelle image de conteneur en exécutant un conteneur qui convertit un fichier Asciidoc de l'environnement d'accueil en HTML.
6. Publiez votre image sur le même projet Gitlab que celui préparé en Section 8.1.1, avec le même nom et la même étiquette que ceux utilisés lors de la création de l'image locale.

8.4. Tutoriel : une image pour déployer un site web avec *nginx*

8.4.1. Rappel du précédent tutoriel *nginx*

Pour débuter, nous allons repartir du premier tutoriel du TP sur la publication de port. Pour rappel, dans ce tutoriel, nous avions créé un conteneur à partir d'une image *nginx* à l'aide de la référence d'image `docker.io/nginx:1.25`, et nous avons exécuté ce conteneur avec à la fois un montage de répertoire pour lui confier des fichiers HTMLs, et avec une publication de port pour que le serveur web soit accessible par le réseau.

Cette approche est tout a fait viable pour déployer un site web. Cependant, si dans le futur il devient nécessaire de migrer notre site web depuis un premier serveur de déploiement vers un nouveau serveur de déploiement, elle a pour inconvénient de nécessiter le déplacement de tous les fichiers du

site web vers le nouveau serveur.

8.4.2. Création d'une image de conteneur *nginx* avec site web intégré

L'objectif ici est de créer une image de conteneur dans laquelle se trouve directement à la fois le logiciel *nginx* et le site web qu'on souhaite déployer. Ainsi, la migration de ce site web ne nécessitera plus qu'à supprimer le conteneur d'un premier serveur de déploiement, et de recréer un conteneur identique sur un nouveau serveur.

Pour accomplir cela, il nous faut, lors de la création de l'image, s'assurer que notre site web (qui contient à minima un fichier `index.html`) soit copié dans le répertoire `/usr/share/nginx/html/` de l'image, car c'est à cet emplacement que *nginx* va charger le site qu'il rendra accessible.

Au sein de l'image *nginx*, le contenu par défaut de `/usr/share/nginx/html/` n'est pas vide, car il contient une page d'accueil d'exemple, qui ne nous intéresse pas ! Il va donc nous falloir *remplacer* ce répertoire par le contenu de notre choix.

On commence par créer un conteneur *template* à partir d'une image de base. Cette fois-ci, on choisit comme base l'image officielle nginx, qui permet de partir d'une base dans laquelle le logiciel nginx est déjà installé :

[sur l'environnement de déploiement distant]

```
$ podman container run -ti --name nginx_template \
                       docker.io/nginx:1.25 \
                       bash
```

On spécifie que l'on souhaite utiliser l'interpréteur de commandes `bash` comme point d'entrée, car par défait le point d'entrée de l'image officielle nginx est le démarrage du serveur web.

Dans notre conteneur *template*, il nous faut d'abord supprimer la page web exemple qui y est fournie par défaut dans l'image de conteneur nginx :

[sur le conteneur]

```
# rm /usr/share/nginx/html/*
```

On va ensuite y mettre à la place notre propre page web. Pour cet exercice, utilisez le fichier HTML que vous aviez généré à l'aide d'Asciidoctor dans l'exercice précédent, et à l'aide d'un second terminal, copiez ce fichier au sein de votre conteneur *template* :

[sur l'environnement de déploiement distant]

```
$ podman container cp <fichier html> nginx_template:/usr/share/nginx/html/index.html
```

Lors de cette copie, on renomme notre fichier HTML en `index.html`, car c'est le fichier qui sera utilisé par défaut par nginx comme point de départ de notre site web.

Dans le conteneur, on peut vérifier que notre fichier est bien arrivé à destination :

[sur le conteneur]

```
# cd /usr/share/nginx/html/
# ls
```

Si tout est bon, on peut quitter le conteneur (ce qui le stoppe au passage) :

[sur le conteneur]

```
# exit
```

Il ne nous reste plus qu'à transfomer notre conteneur *template* en nouvelle image de conteneur. Il nous faut pour cela spécifier un *point d'entrée*. On choisit ici de réutiliser à l'identique le même point d'entrée que l'image de conteneur officielle nginx, qui consiste juste à démarrer le serveur web :

[sur l'environnement de déploiement distant]

```
$ podman container commit  nginx_template \
                           --change 'CMD nginx -g "daemon off;"' \
                           tutoriel-nginx:1
```

Enfin, testez votre image en créant un conteneur :

[sur l'environnement de déploiement distant]

```
$ podman container run --rm \
                       --detach \
                       --name nouveau_serveur_web \
                       --publish <mon port réseau>:80 \
                       localhost/tutoriel-nginx:1
```

Cette fois-ci, pas besoin de montage de répertoire pour déployer notre site web : ce dernier est déjà inclus dans l'image, et donc dans le conteneur !

Vérifiez que votre déploiement fonctionne en accédant depuis un navigateur à l'adresse `http://podman:<mon port réseau>` : normalement votre HTML s'affiche comme prévu !

8.4.3. Publication de l'image

Enfin, publiez votre image à l'aide de la commande :

```
$ podman image push localhost/tutoriel-nginx:1 \
                    docker-registry.univ-nantes.fr/<chemin projet>/tutoriel-nginx:1
```

en remplaçant à nouveau `<chemin projet>` par l'emplacement de votre projet sur Gitlab.

8.5. Exercice : une image de conteneur pour `economic_dispatch`

8.5.1. Énoncé

On souhaite préparer une image de conteneur qui permette d'exécuter facilement le l'outil `economic_dispatch` [https://gitlab.univ-nantes.fr/gl/developpement_exploitation/resources/small_programs/-/tree/main/5_economic_dispatch?ref_type=heads] vu dans les premiers TPs. Pour rappel, c'est un outil écrit en Julia qui permet de faire un calcul d'optimisation à partir d'un modèle de scénario.

Pour notre image de conteneur, on fait les choix suivants :

- On utilisera comme image de base `docker.io/julia`, avec l'étiquette `1.11-trixie`.
- Lors de la phase d'usage de l'image, on souhaite pouvoir confier un modèle d'entrée à notre outil.
- L'image de conteneur produite devra avoir pour nom `exercice-economic-dispatch` et pour étiquette `a`.

Pour rappel, il est possible d'installer une bibliothèque Julia avec la commande suivante:

```
# julia -e 'import Pkg; Pkg.add("<nom de la bibliothèque>")'
```

8.5.2. Questions

1. Faites un travail préliminaire de *conception* de votre future image de conteneur.

   Réfléchissez bien au *point d'entrée* que vous souhaitez avoir.

2. Préparez un conteneur *template* dans lequel vous installez tout le nécessaire pour répondre au besoin de l'énoncé.

   Pour récupérer le code source du projet Gitlab, vous pouvez suivre les instructions dédiées de la notice Gitlab.

3. En vous basant sur votre travail préliminaire de conception, transformez votre conteneur *template* en image de conteneur.
4. Vérifiez que l'image créée est bien présente dans votre stockage local d'images.
5. Testez votre nouvelle image de conteneur en exécutant un conteneur qui s'exécute sur le modèle exemple fourni dans la documentation de l'outil.
6. Publiez votre image sur le même projet Gitlab que celui préparé en Section 8.1.1, avec le même nom et la même étiquette que ceux utilisés lors de la création de l'image locale.

8.6. Exercice : créer une image pour *Web Calculator* (Python)

8.6.1. Énoncé

On souhaite créer une image de conteneur pour un logiciel fraîchement développé appelé *Web Calculator*. C'est une application web développée en Python, dont le code source est disponible sur le projet Gitlab suivant [https://gitlab.univ-nantes.fr/gl/developpement_exploitation/resources/web-calculator].

Une fois exécutée, l'application peut être accédée via un navigateur web, et fournit une interface pour faire des calculs arithmétiques entre deux nombres (addition, soustraction, etc.).

Lisez la documentation de l'application [https://gitlab.univ-nantes.fr/gl/developpement_exploitation/resources/web-calculator/-/blob/main/README.adoc], et en particulier les sections 3 et 4. On y trouve des informations importantes, notamment :

- Le logiciel nécessite un certain nombre de dépendances Python.
- Ces dépendances peuvent être installées à l'aide de l'utilitaire `pip`.
- Une commande `uvicorn` permet de démarrer le logiciel.
- Le logiciel nécessite le port réseau `8000` pour recevoir des connexions.

La documentation fait référence à un concept Python appelé *venv*. C'est une manière pratique pour créer des environnements Python dans lesquelles on installer des dépendances spécifiques à un développement ou un déploiement particulier. **Dans notre cas, nous pouvons ignorer l'utilisation de *venv*** car cela n'est pas nécessaire dans un conteneur, qui fourni déjà un environnement isolé adapté.

8.6.2. Questions

1. Faites un travail préliminaire de *conception* de votre future image de conteneur.

À nouveau, réfléchissez bien au *point d'entrée* que vous souhaitez avoir.

2. Préparez un conteneur *template* dans lequel vous installez tout le nécessaire pour répondre au besoin de l'énoncé.

Pour récupérer le code source du projet Gitlab, vous pouvez suivre les instructions dédiées de la notice Gitlab.

3. En vous basant sur votre travail préliminaire de conception, transformez votre conteneur *template* en image de conteneur, avec pour nom `webcalculator` et pour étiquette `dev`.
4. Vérifiez que l'image créée est bien présente dans votre stockage local d'images.
5. Testez votre application à l'aide d'un navigateur.
6. Publiez votre image sur le même projet Gitlab que celui préparé en Section 8.1.1, avec le même nom et la même étiquette que ceux utilisés lors de la création de l'image locale.

Chapitre 9. 💻 TP : Scripter la conteneurisation d'un logiciel

9.1. Présentation et objectifs

L'objectif de ce TP est d'apprendre à utiliser le language *Containerfile* ainsi que l'outil Podman pour créer de nouvelles images de conteneurs.

Tout pour tous les TPs, lorsque le sujet de TP vous propose d'exécuter une commande, il est demandé que vous arriviez à comprendre tout ce que comporte cette commande. **Prenez le temps de bien lire et décortiquer chaque commande qui vous est proposée**, et n'hésitez pas à poser des questions.

9.2. Préparatifs

Ce TP est similaire au TP précédent sur le principe, à savoir la *création* et la *publication* d'images. La seule différence est que la création d'images va cette fois se faire avec `Containerfile`.

Pour la publication d'images, vous allez pouvoir réutiliser le projet Gitlab `developpement_exploitation-tp-images` créé lors du TP précédent.

Il vous faut également ne pas oublier de vous ré-authentifier auprès du registre de conteneurs pour les exercices qui vous demandent de publier l'image produite :

```
$ podman login docker-registry.univ-nantes.fr
```

9.3. Tutoriel : image `cowsay` via un `Containerfile`

9.3.1. Rappel

Dans le TP précédent, nous avions vu comme premier exemple comment créer une image de conteneur qui met à disposition la commande `cowsay`.

Dans ce tutoriel, nous allons refaire exactement la même chose, mais en l'automatisant à l'aide d'un script `Containerfile`.

9.3.2. Premier `Containerfile` avec `FROM` et `RUN`

Sur l'environnement de déploiement, créez un nouveau répertoire `tutoriel-cowsay-containerfile`, et préparez le `Containerfile` suivant :

```
FROM docker.io/ubuntu:24.04

RUN apt update
RUN apt install -y cowsay
```

On peut observer l'ajout de l'option `-y` lors de l'appel à `apt install`. La raison est que durant l'exécution d'un `Containerfile`, il n'est pas possible d'interagir avec le constructeur d'image. Or, par défaut, la commande `apt` peut demander à

> l'utilisateur de confirmer l'installation d'un paquet à l'aide de la touche clavier `y` (pour *yes*) ou `n` (pour *no*). Pour éviter cela, l'option `-y` indique qu'il faut automatiquement répondre par `y` à toutes les questions posées par `apt`.

Faites ensuite appel à Podman pour créer une nouvelle image avec pour nom `tutoriel-cowsay` et pour étiquette `v3` :

```
$ podman image build --tag tutoriel-cowsay:v3 .
```

La dernière partie d'une commande `podman image build` doit être le chemin vers un répertoire dit de *contexte*. Dans notre cas, il suffit donc de pointer vers le répertoire `tutoriel-cowsay-containerfile`, qui est normalement ici votre répertoire courant, donc `"."`. Si le répertoire de contexte possède un `Containerfile`, il sera utilisé par défaut, sauf si l'option `--file` est spécifiée.

Vous verrez alors apparaître une sortie textuelle complète de ce qui est réalisé pour créer l'image. Chaque instruction du `Containerfile` apparaît sous la forme d'une étape (`STEP`).

Une fois l'image construite, vérifiez qu'elle est bien présente dans votre stockage d'image local :

```
$ podman image list
```

On notera que la colonne `REPOSITORY` indique la référence d'image, qui dans notre cas commence par `localhost` puisque l'image est sur le stockage local.

Il est possible que vous ayez également dans cette liste un reliquat d'images téléchargées dans les précédents exercices, ce qui est normal.

Il ne nous reste plus qu'à tester notre nouvelle image, on peut donc faire :

```
$ podman container run --rm -ti localhost/tutoriel-cowsay:v3
```

Comme lors du TP précédent, on se retrouve dans un interpréteur de commandes au sein du conteneur. On peut alors directement exécuter la commande souhaitée, sans avoir à l'installer au préalable :

```
$ /usr/games/cowsay "Bonjour depuis notre image construite avec un Containerfile !"
```

9.3.3. Ajout d'un point d'entrée avec `CMD`

Comme la dernière fois, on souhaite maintenant améliorer notre image en modifiant quel logiciel est exécuté lors du démarrage d'un conteneur créé avec l'image. Cela nécessite de spécifier un *point d'entrée* pour l'image à l'aide de l'instruction `CMD` dans le `Containerfile`.

Il est possible dans une instruction `CMD` de faire référence à une variable d'environnement avec la syntaxe `$VARIABLE`. On prépare donc la deuxième version suivante pour notre `Containerfile`, qui spécifie un point d'entrée avec la commande `cowsay` prenant comme argument le contenu d'une

nouvelle variable d'environnement `MESSAGE` :

```
FROM docker.io/ubuntu:24.04

RUN apt update
RUN apt install -y cowsay

CMD /usr/games/cowsay $MESSAGE
```

À l'aide de ce `Containerfile` modifié, construisez une nouvelle image de conteneur, cette fois-ci avec l'étiquette `v4` :

```
$ podman image build --tag tutoriel-cowsay:v4 .
```

On peut vérifier que l'image est bien créée à nouveau à l'aide de la commande :

```
$ podman image list
```

On peut ainsi désormais exécuter un conteneur de la sorte :

```
$ podman container run --rm \
                       --env MESSAGE="Re-bonjour depuis notre nouvelle image !" \
                       localhost/tutoriel-cowsay:v4
```

Et voilà en fonctionnement une construction automatisée de notre image paramétrable !

9.4. Exercice : image Asciidoctor via `Containerfile`

Refaites l'exercice Asciidoctor du TP précédent, mais cette fois-ci construisez votre image de conteneur à l'aide d'un `Containerfile`.

9.5. Tutoriel : image site web *nginx* via `Containerfile`

9.5.1. Rappel

Pour rappel, dans le TP précédent nous avions préparé une image de conteneur contenant un petit site web près à être déployé. Nous allons dans ce tutoriel accomplir la même chose, mais à l'aide d'un `Containerfile`.

9.5.2. `Containerfile` avec `FROM`, `RUN` et `COPY`

L'objectif ici est de créer une image de conteneur dans laquelle se trouve directement à la fois le logiciel *nginx* et le site web qu'on souhaite déployer.

Pour accomplir cela, il nous faut, lors de la création de l'image, s'assurer que notre site web (qui contient à minima un fichier `index.html`) soit copié dans le répertoire `/usr/share/nginx/html/` de l'image, car c'est à cet emplacement que *nginx* va charger le site qu'il rendra accessible.

Au sein de l'image *nginx*, le contenu par défaut de `/usr/share/nginx/html/` n'est pas vide, car il contient une page d'accueil d'exemple, qui ne nous intéresse pas ! Il va donc nous falloir *remplacer* ce répertoire par le contenu de notre choix.

Sur l'environnement de déploiement, créez un nouveau répertoire `tutoriel-nginx`. Dans ce répertoire, mettez le HTML que vous aviez généré à l'aide d'Asciidoctor dans l'exercice précédent, et renommez ce fichier en `index.html`. Ensuite, rendez-vous dans le répertoire `tutoriel-nginx` et préparez le `Containerfile` suivant :

```
FROM docker.io/nginx:1.25

RUN rm /usr/share/nginx/html/*

COPY ./*.html /usr/share/nginx/html/index.html
```

On observe que :

- À l'aide de `RUN`, une commande `rm` est utilisée pour supprimer le contenu du répertoire `/usr/share/nginx/html/`, afin de supprimer la page d'accueil d'exemple qui est fournie d'office dans l'image nginx.
- À l'aide de `COPY`, On ajoute à l'image tous les fichiers HTML du répertoire `tutoriel-nginx`, au cas où notre site comporte plusieurs pages.

Il est possible d'écrire un `Containerfile` équivalent à celui ci-dessus en utilisant la commande `WORKDIR` pour changer le répertoire courant, qui permet de simplifier les commandes `RUN` et `ADD` qui suivent. N'hésitez pas à essayer !

Construisez une image de conteneur avec la commande suivante :

```
$ podman image build --tag tutoriel-nginx:2 .
```

Enfin, testez votre image en créant un conteneur :

```
$ podman container run --rm \
                       --detach \
                       --name nouveau_serveur_web \
                       --publish <mon port réseau>:80 \
                       localhost/tutoriel-nginx:2
```

Vérifiez que votre déploiement fonctionne en accédant depuis un navigateur à l'adresse `http://podman:<mon port réseau>` : normalement votre HTML s'affiche comme prévu !

Une fois ce tutoriel terminé, n'oubliez pas d'arrêter et de supprimer le conteneur créé.

9.6. Exercice : créer une image pour *Web Calculator* (Python)

9.6.1. Énoncé

On souhaite créer une image de conteneur pour un logiciel fraîchement développé appelé *Web Calculator*. C'est une application web développée en Python, dont le code source est disponible sur le projet Gitlab suivant [https://gitlab.univ-nantes.fr/gl/developpement_exploitation/resources/web-calculator].

Une fois exécutée, l'application peut être accédée via un navigateur web, et fournit une interface pour faire des calculs arithmétiques entre deux nombres (addition, soustraction, etc.).

Lisez la documentation de l'application [https://gitlab.univ-nantes.fr/gl/developpement_exploitation/resources/web-calculator/-/blob/main/README.adoc], et en particulier les sections 3 et 4. On y trouve des informations importantes, notamment :

- Le logiciel nécessite un certain nombre de dépendances Python.
- Ces dépendances peuvent être installées à l'aide de l'utilitaire `pip`.
- Une commande `uvicorn` permet de démarrer le logiciel.
- Le logiciel nécessite le port réseau `8000` pour recevoir des connexions.

La documentation fait référence à un concept Python appelé *venv*. C'est une manière pratique pour créer des environnements Python dans lesquelles on installe des dépendances spécifiques à un développement ou un déploiement particulier. **Dans notre cas, nous pouvons ignorer l'utilisation de *venv*** car cela n'est pas nécessaire dans un conteneur, qui fourni déjà un environnement isolé adapté.

9.6.2. Préparatifs

Vous allez avoir besoin de récupérer le code source du logiciel afin de l'ajouter dans votre image de conteneur. Pour cela, le plus simple est d'utiliser l'utilitaire `git` de la manière suivante :

```
$ git clone <URL git>
```

avec `<URL git>` l'URL git du projet que l'on souhaite récupérer.

Pour obtenir l'URL dit d'un projet Gitlab, on clique sur le bouton "*Code*" et on copie le contenu "*Cloner avec HTTPS*". On peut ensuite coller l'URL dans un terminal pour faire notre `git clone`.

Par exemple pour le projet *web calculator* on peut faire :

[sur l'environnement de déploiement distant]

```
$ git clone https://gitlab.univ-nantes.fr/gl/developpement_exploitation/resources/web-
calculator.git
```

ce qui va créer un répertoire `web-calculator` où sera téléchargé tout le code source du projet.

9.6.3. Questions

Pour récupérer le code source du projet Gitlab, vous pouvez suivre les instructions dédiées de la notice Gitlab.

1. Préparez un `Containerfile` permettant de créer une image *Web Calculator*. Pour cela, posez-vous les questions suivantes :
   a. Quels outils doivent être présents dans notre image de base ? Par conséquent, quelle image de base choisir ?

> Pour choisir une image de base, n'hésitez pas à relire la section correspondante dans le chapitre sur les images de conteneurs [https://gl.univ-nantes.io/developpement_exploitation/workbook/site/2026-2027/cms/ecosysteme-images/#sec-images-base].

   b. Quels fichiers faut-il mettre dans l'image afin d'exécuter le logiciel ?
   c. Quelles commandes faut-il exécuter lors de la construction de l'image ?
   d. Quelle métadonnée est nécessaire pour le point d'entrée ?
2. Créez une image en exécutant votre `Containerfile` à l'aide de Podman. On souhaite que notre image se nomme `webcalculator` et ait pour étiquette `dev`.
3. Préparez une commande pour créer un conteneur à l'aide de votre image créée localement, puis déployez votre application sur le serveur de déploiement.
4. Testez votre application à l'aide d'un navigateur.
5. Publiez votre image au sein de votre projet Gitlab.

9.7. Exercice : créer une image *webdav-embedded-server* (Java)

9.7.1. Énoncé

On souhaite créer une image de conteneur pour le logiciel *webdav-embedded-server*, développé en Java, dont le code source est disponible au sein d'un projet sur le Gitlab Université [https://gitlab.univ-nantes.fr/gl/developpement_exploitation/resources/webdav-embedded-server].

Ce logiciel permet d'exécuter un serveur de stockage de fichiers, sur lequel il est possible de se connecter à l'aide du protocole Webdav [https://fr.wikipedia.org/wiki/WebDAV]. Ce protocole est aujourd'hui très répandu et de nombreux explorateurs de fichiers sont capables d'accéder à distance au contenu d'un serveur Webdav, afin de lire, modifier, créer ou supprimer des fichiers.

Si on analyse le contenu du projet on découvre plusieurs éléments importants :

- On peut voir la présence d'un fichier `build.gradle`, qui signifie que la compilation du code doit se faire avec l'outil Gradle [https://gradle.org/]. Si on fouille dans le fichier `build.gradle` [https://gitlab.univ-nantes.fr/gl/developpement_exploitation/resources/webdav-embedded-server/-/blob/master/build.gradle], on découvre qu'une commande Gradle intitulée `fatJar` est disponible, et permet de compiler le code du logiciel au sein d'un fichier `.jar` qui embarque automatiquement toutes les dépendances du logiciel – très pratique pour un déploiement !
- On peut voir dans le README une description sommaire du mode *Standalone* du logiciel, qui permet de démarrer un serveur. On peut voir que différentes options sont disponibles pour lancer l'application, notamment `--credentials` qui permet de spécifier un compte utilisateur pour se connecter au serveur, et `--port` qui permet de spécifier le port à utiliser pour se connecter à l'application (par défaut, `8080`). Après les options, il faut spécifier le chemin du répertoire dans lequel le serveur Webdav stockera ses fichiers.

Si on met tout cela bout à bout, cela signifie qu'on peut compiler notre application à l'aide de la commande :

```
$ gradle fatJar
```

Cela produira normalement un fichier `build/libs/webdav-embedded-server-0.2.1-SNAPSHOT-fatjar.jar`.

Malheureusement, le serveur de déploiement de l'Université possède une contrainte qui va entraver l'usage de Gradle : **l'usage obligatoire d'un proxy HTTP pour toute connexion HTTP réalisée (donc pour tout téléchargement de fichier).** Or, Gradle a besoin de télécharger des fichiers pour récupérer toutes les dépendances Java du projet... !

Jusqu'ici le problème ne s'est pas présenté durants les TPs car le serveur de déploiement est bien configuré, et possède une variable d'environnement standard appelée `https_proxy` qui est utilisée automatiquement par tous les logiciels pour utiliser le proxy de manière transparente. Ainsi, on peut généralement ignorer la présence du proxy, et laisser les logiciels l'utiliser par eux-même sans que nous ayons à faire quoi que ce soit.

Mais Gradle n'est pas capable d'utiliser la variable `https_proxy` ! Il est nécessaire de lui donner cette information via deux options :

- `-Dhttps.proxyHost=<nom d'hôte proxy HTTP>`
- `-Dhttps.proxyPort=<port réseau proxy HTTP>`

À l'Université, le serveur de proxy HTTP est accessible avec le nom d'hôte `proxy.ensinfo.sciences.univ-nantes.prive` et avec le port `3128`.

Si on met tout cela bout à bout, il est donc dans notre cas nécessaire d'appeler Gradle de la sorte :

```
$ gradle -Dhttps.proxyHost=proxy.ensinfo.sciences.univ-nantes.prive \
         -Dhttps.proxyPort=3128 fatJar
```

Ensuite, depuis le répertoire `build/libs`, on peut lancer le serveur avec la commande :

```
$ java -jar webdav-embedded-server-0.2.1-SNAPSHOT-fatjar.jar
           --credentials "<utilisateur>:<mot de passe>"
```

Et le serveur Webdav démarrera en écoutant sur le port `8080`, et n'autorisera que les informations de connections spécifiées par `--credentials`.

Contraintes à respecter : Lors de la création d'un conteneur à partir votre l'image, on souhaite pouvoir spécifier :

- un nom d'utilisateur et un mot de passe qui seront les informations de connexion acceptées par le serveur Webdav.
- un emplacement de l'environnement d'accueil dans lequel le serveur stockera ses fichiers.

9.7.2. Préparatifs

Récupérez le code source du logiciel à l'aide de `git clone`.

9.7.3. Questions (création de l'image)

Pour récupérer le code source du projet Gitlab, vous pouvez suivre les instructions dédiées de la notice Gitlab.

1. Préparez un `Containerfile` permettant de créer une image *webdav-embedded-server*. Pour cela, posez-vous les questions suivantes :
   a. Quels outils doivent être présents dans notre image de base ? Par conséquent, quelle image de base choisir ?
   b. Quels fichiers faut-il mettre dans l'image afin de compiler le logiciel ?
   c. Quels fichiers faut-il mettre dans l'image afin d'exécuter le logiciel ? D'où provient ce fichier ?
   d. Quelles commandes faut-il exécuter lors de la construction de l'image ?
   e. Par quels mécanismes gérer les deux contraintes posées plus haut ?
   f. Quelle métadonnée est nécessaire pour le point d'entrée ?
2. Créez une image en exécutant votre `Containerfile` à l'aide de Podman. On souhaite que notre image se nomme `webdavserver` et ait pour étiquette `latest`.
3. Préparez une commande pour créer un conteneur à l'aide de votre image créée localement, puis déployez votre application sur le serveur de déploiement.
4. Testez votre application de la sorte :
   a. Sur votre poste de travail linux, ouvrez l'application *Fichiers* (que vous utilisez quotidiennement pour accéder à vos fichiers).
   b. Utilisez le raccourci clavier *CTRL+L* afin d'accéder à la barre d'adresse en haut de l'a fenêtre.
   c. Dans cette barre d'adresse, saisissez `dav://podman:<port réseau>`, afin de vous connecter à votre serveur Webdav déployé sur le serveur de déploiement. Saisissez les informations de connexion demandées, et validez. Si vous n'avez pas de message d'erreur, vous êtes bien connecté !
   d. Tentez de copier un fichier de votre choix au sein de votre serveur Webdav.
   e. Vérifiez que le fichier copié apparaît bien dans l'emplacement du serveur de déploiement où le serveur stocke ses fichiers.
5. Enfin, publiez votre image au sein de votre projet Gitlab.

9.7.4. Questions (amélioration de l'image)

Il est conseillé (mais pas indispensable) de revenir sur ces questions "avancées" uniquement après voir fini l'exercice *WordcloudGenerator* qui suit, ou bien pendant les longs temps d'attente dûs à l'installation de paquets durant l'exercice *WordcloudGenerator*.

1. Quelle est la taille de l'image *webdav-embedded-server* que vous avez produite ?
2. Qu'est-ce que qui peut expliquer la taille occupée par l'image ?
3. Quelle stratégie étudiée en cours permettrait de réduire la taille de l'image ?

4. Proposez un nouveau `Containerfile` qui mette en œuvre cette stratégie, et comparez la taille de l'image obtenue avec la première version.

Si vous avez déjà mis en place cette stratégie – déjà, bravo ! – vous pouvez bien sûr sauter cette partie.

9.8. Exercice : créer une image *WordcloudGenerator* (langage R)

9.8.1. Énoncé

On souhaite créer une image de conteneur pour le logiciel *WordcloudGenerator* développé en R, dont le code source est là aussi disponible au sein d'un projet sur le Gitlab Université [https://gitlab.univ-nantes.fr/gl/developpement_exploitation/resources/wordcloudgenerator].

Ce logiciel permet de construire un nuage de mots [https://fr.wikipedia.org/wiki/Nuage_de_mots-cl%C3%A9s] à partir d'un fichier texte simple. Il s'exécute sous la forme d'une application web qui peut être accédée depuis un navigateur. L'interface permet d'envoyer un fichier `.txt` et de sélectionner le nombre de mots devant apparaître dans le nuage.

Lisez la documentation technique du projet [https://gitlab.univ-nantes.fr/gl/developpement_exploitation/resources/wordcloudgenerator/-/blob/main/README.adoc]. On peut y trouver les informations utiles suivantes :

- Le logiciel nécessite un ensemble de paquets R.
- Le logiciel nécessite également une bibliothèque logicielle C appelée *libxml2*.
- Une commande simple est fournie pour exécuter le logiciel.

Installer un paquet R

Pour installer un paquet R dans un environnement donné où R est déjà installé, on peut utiliser la commande suivante :

```
$ Rscript -e 'install.packages("<nom du package>")'
```

Installer un paquet R peut prendre beaucoup de temps ! N'hésitez pas à faire la fin de l'exercice précédent pendant la longue construction de votre image.

9.8.2. Préparatifs

Récupérez le code source du logiciel à l'aide de `git clone`.

9.8.3. Questions

Pour récupérer le code source du projet Gitlab, vous pouvez suivre les instructions dédiées de la notice Gitlab.

1. Préparez un `Containerfile` permettant de créer une image *WordcloudGenerator*. Pour cela, posez-vous les questions suivantes :

a. Quels outils doivent être présents dans notre image de base ? Par conséquent, quelle image de base choisir ?

b. Comment faire pour installer la bibliothèque logicielle C nécessaire au sein de l'image de base choisie ?

Indice : vous pouvez allez voir quelle image de base a été utilisée pour construire l'image de base que vous souhaitez utiliser, cela peut vous aider à savoir quel utilitaire de gestion de dépendance (`apt`, ou `apk`, ou `dnf`, ou autre) utiliser !

c. Quels fichiers faut-il mettre dans l'image afin d'exécuter le logiciel ?

d. Quelles commandes faut-il exécuter lors de la construction de l'image ?

e. Quelle métadonnée est nécessaire pour le point d'entrée ?

2. Créez une image en exécutant votre `Containerfile` à l'aide de Podman. On souhaite que notre image se nomme `wordcloudgenerator` et ait pour étiquette `v1`.

3. Préparez une commande pour créer un conteneur à l'aide de votre image créée localement, puis déployez votre application sur le serveur de déploiement.

4. Testez votre déploiement à l'aide d'un navigateur. Vous aurez pour cela besoin d'un fichier texte avec du contenu à analyser. Vous pouvez par exemple utiliser le contenu du livre *The Adventures of Sherlock Holmes* [https://www.gutenberg.org/cache/epub/1661/pg1661.txt], ou encore le contenu du livre *Moby Dick* [https://www.gutenberg.org/cache/epub/2701/pg2701.txt].

5. Enfin, publiez votre image au sein de votre projet Gitlab.

Chapitre 10. 💻 TP : Faire un déploiement multi-conteneurs

Dans un TP précédent, vous avez vu comment rendre un conteneur joignable *depuis l'extérieur de son environnement d'accueil*, à l'aide d'un mécanisme appelé la *publication de port*. Cependant, la publication de port n'est pas adaptée lorsque des logiciels exécutés dans des conteneurs *différents* ont besoin de communiquer entre eux.

Dans ce TP, vous allez voir comment rendre possible la communication inter-conteneurs à l'aide de **réseaux virtuels**.

10.1. Tutoriel : premiers réseaux virtuels

Jusqu'ici, le déploiement de chaque service a nécessité la création d'un seul conteneur à la fois, souvent à l'aide d'une *publication de port* pour rendre ce conteneur accessible depuis un réseau local. Or, il est courant qu'un même service nécessite le fonctionnement de *plusieurs conteneurs* reliés entre eux – typiquement, un logiciel applicatif associé à un système de gestion de base de données. **Voyons comment déployer plusieurs conteneurs mis en réseau.**

10.1.1. Deux conteneurs sur un même réseau virtuel

Mise en place

Pour commencer, créons un *réseau virtuel* `r1` qui permette de relier plusieurs conteneurs entre eux :

[sur l'environnement de déploiement distant]

```
$ podman network create r1
```

On peut vérifier que ce réseau a bien été créé avec la commande suivante, qui liste les réseaux virtuels existants :

[sur l'environnement de déploiement distant]

```
$ podman network ls
```

Un conteneur peut être relié à un réseau virtuel *uniquement lors de sa création*. Dans un premier terminal, créons un premier conteneur interactif Alpine appelé `conteneurA` relié à ce réseau (à l'aide de l'option `--network`) :

[sur l'environnement de déploiement distant]

```
$ podman container run --name conteneurA \
                       --rm -ti \
                       --network r1 \
                       docker.io/alpine:3.22
```

Gardez au chaud ce premier terminal dans lequel s'exécute `conteneurA`, et ouvrez un *second* terminal dans lequel nous allons créer un second conteneur interactif `conteneurB` également connecté au réseau appelé `r1` :

[sur l'environnement de déploiement distant]

```
$ podman container run --name conteneurB \
                       --rm -ti \
                       --network r1 \
                       docker.io/alpine:3.22
```

Ensuite, pour tester leur mise en réseau, nous allons installer dans les deux conteneurs l'utilitaire netcat [https://en.wikipedia.org/wiki/Netcat], qui a l'avantage de permettre de facilement créer un petit processus serveur et un petit processus client.

Au sein d'un conteneur Alpine, *netcat* est disponible via le paquet `netcat-openbsd` installable à l'aide du gestionnaire de paquets `apk`. On commence donc par installer *netcat* dans chacun des deux conteneurs :

[dans `conteneurA`]

```
# apk update && apk add netcat-openbsd
```

[dans `conteneurB`]

```
# apk update && apk add netcat-openbsd
```

Utiliser *netcat* pour du test réseau simple

Une fois *netcat* installé dans un environnement Linux donné, il peut être utilisé à l'aide de la commande `nc`. On peut distinguer deux usages simples de la commande netcat `nc` :

- on peut demander à `nc` de se comporter comme *un processus serveur qui écoute sur un port réseau*, de manière à pouvoir recevoir une connexion requise par un processus client, à l'aide de la commande suivante :

  ```
  nc -l <port d'écoute>
  ```

  L'option `-l` signifie *listen*, ce qui se traduit en "écouter".

- on peut demander à `nc` de se comporter comme *un processus client qui se connecte à un port réseau d'un environnement donné* (désigné par son *nom d'hôte*), de manière pouvoir se connecter à un processus serveur exécuté dans cet autre environnement, à l'aide de la commande suivante :

  ```
  nc <nom d'hôte> <port auquel se connecter>
  ```

  Un processus serveur `nc` ne peut pas recevoir plus d'une seule connexion à la fois, ainsi *un seul* processus client peut se connecter à un processus serveur. Cela n'est pas représentatif du cas général dans le monde du logiciel : il est habituel qu'un processus serveur puisse

recevoir *plusieurs connexions en parallèle* (par exemple un serveur web peut recevoir des connexions de plusieurs navigateurs web).

Si une de ces commandes se termine instantanément, et vous rend la main, c'est que la connexion a échoué – cela peut arriver si le nom d'hôte et/ou le port renseigné est incorrect ou incohérent.

Mais à l'inverse, si les deux commandes ne se terminent pas et ne vous rendent pas la main, c'est généralement qu'**une connexion a été établie entre processus serveur et processus client**.

Il est alors possible d'effectuer une saisie au clavier, puis d'appuyer sur *Entrée*, pour envoyer un texte au processus serveur, qui l'affiche dès réception. À l'inverse, si on écrit un texte côté serveur et qu'on appuie sur entrée, le message s'affiche côté client.

Pour arrêter une commande `nc` en cours d'exécution, on peut faire la combinaison de touches *Ctrl+C* dans le terminal concerné.

Faisons l'inventaire de ce que nous avons en place :

- Un réseau virtuel appelé `r1`,
- Un conteneur appelé `conteneurA`, relié à ce réseau virtuel. Au sein de ce réseau virtuel, `conteneurA` est joignable au travers du nom d'hôte `conteneurA` (identique à son nom de conteneur).
- Un conteneur appelé `conteneurB`, relié au même réseau virtuel. Là aussi, au sein de ce réseau virtuel, `conteneurB` est joignable au travers du nom d'hôte `conteneurB` (identique à son nom de conteneur).

Maintenant, utilisons *netcat* pour tester cette mise en réseau !

Test 1 : serveur dans `conteneurA`, client dans `conteneurB`

Utilisons *netcat* pour démarrer un processus *serveur* qui écoute sur le port `5000` du `conteneurA` :

[dans `conteneurA`]

```
# nc -l 5000
```

Une fois cette cette commande exécutée, **il semble ne rien se passer**. C'est normal : *netcat* a démarré son processus serveur et est en attente d'une connexion, donc il reste (très) silencieux pour le moment.

Puis utilisons *netcat* pour démarrer un processus *client* qui se connecte sur le port `5000` de `conteneurA` :

[dans `conteneurB`]

```
# nc conteneurA 5000
```

Si la connexion est bien établie (lire l'encart sur *netcat* plus haut), vous pouvez alors :

- envoyer `"bonjour depuis conteneurB !"` depuis le processus client exécuté sur le `conteneurB`, et

vérifier que le message s'affiche dans le terminal avec le `conteneurA`,

- envoyer `"bonjour depuis conteneurA !"` depuis le processus serveur exécuté sur le `conteneurA`, et vérifier que le message s'affiche dans le terminal avec le `conteneurB`.

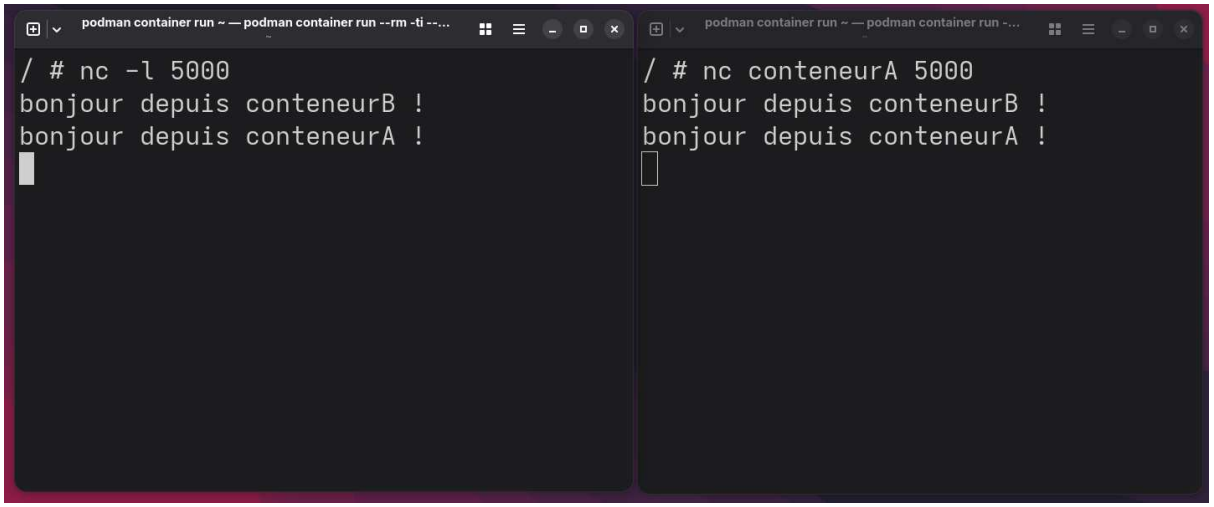


Figure 7. Échange de messages entre les deux conteneurs Alpines

Si tout fonctionne, vous avez fait votre première mise en réseau de plusieurs conteneurs !

Test 2 : client dans `conteneurB`, client dans `conteneurA`

Juste pour démontrer que la connexion entre les deux conteneurs est bien symétrique, reproduisez le même test mais en inversant les deux rôles :

- démarrez un processus `nc` *serveur* depuis `conteneurB`, sur un port d'écoute *différent* de `5000`,
- démarrer un processus `nc` *client* depuis `conteneurA` pour se connecteur au `conteneurB` sur le port choisi à l'étape précédente,
- échangez des messages dans les deux sens pour vérifier que tout fonctionne.

10.1.2. Trois conteneurs sur deux réseaux virtuels

Mise en place

Dans la dernière partie de ce tutoriel, voyons comment deux réseaux virtuels peuvent coexister.

Notre objectif va être de garder le premier réseau, puis de créer un second réseau `r2`, et d'y connecter une partie de nos conteneurs. Nous souhaitons faire en sorte d'avoir :

- un conteneur `conteneurA` connecté au réseau `r1`,
- un conteneur `conteneurB` connecté aux réseaux `r1` **et** `r2`,
- un conteneur `conteneurC` connecté au réseau `r2`.

Commençons par créer le nouveau réseau :

[sur l'environnement de déploiement distant]

```
$ podman network create r2
```

Vérifions que les deux réseaux co-existent bien :

[sur l'environnement de déploiement distant]

```
$ podman network ls
```

On peut garder le `conteneurA` à l'identique, car est déjà relié au réseau `r1` comme souhaité.

Nous devons par contre recréer le `conteneurB` car nous souhaitons qu'il soit relié aux *deux* réseaux virtuels. On peut faire les commandes suivantes dans le terminal dédié au `conteneurB` :

[sur l'environnement de déploiement distant]

```
$ podman container rm --force conteneurB
$ podman container run --name conteneurB \
                       --rm -ti \
                       --network r1 \
                       --network r2 \
                       docker.io/alpine:3.22
```

Enfin, dans un *troisième terminal*, on crée le `conteneurC` de la sorte :

[sur l'environnement de déploiement distant]

```
$ podman container run --name conteneurC \
                       --rm -ti \
                       --network r2 \
                       docker.io/alpine:3.22
```

Sur les deux conteneurs Alpine ainsi (re)créés, installez la commande `nc` comme fait précédemment.

Voyons maintenant ce que ce déploiement jouet permet de faire.

Test 1 : communications depuis le `conteneurB`

- Sur le `conteneurA`, démarrez un processus serveur `nc` qui écoute sur le port `8888`.
- Sur le `conteneurC`, démarrez un processus serveur `nc` qui écoute également sur le port `8888`.

On peut sans problème ici utiliser le même numéro de port, car chacun de ces ports est ouvert dans un environnement différent (et donc sur une interface réseau différente).

- Sur le `conteneurB`, démarrez un processus client `nc` qui se connecte au port `8888` du `conteneurA`, puis échangez des messages dans les deux sens pour vérifier que tout fonctionne.
- Sur le `conteneurB`, démarrez un processus client `nc` qui se connecte au port `8888` du `conteneurC`, puis échangez des messages dans les deux sens pour vérifier que tout fonctionne.

Si tout se passe bien, vous pouvez constater que les messages arrivent à s'échanger dans les deux cas :

- Dans le premier cas, les messages entre `conteneurB` et `conteneurA` transitent via le réseau virtuel `r1`.

- Dans le second cas, les messages entre `conteneurB` et `conteneurC` transitent via le réseau virtuel `r2`.

Test 2 : communication entre `conteneurA` et `conteneurC` ?

- Sur le `conteneurA`, démarrez un processus serveur `nc` qui écoute sur le port `9999`.
- Sur le `conteneurC`, démarrez un processus client `nc` qui tente de se connecter au port `9999` du `conteneurA`, puis tentez d'échanger des messages entre les deux processus.

Si votre mise en place est correcte, les échanges de messages ne devraient ici... *pas fonctionner* !

En effet, même s'il existe un réseau `r1` entre `conteneurA` et `conteneurB`, et un réseau `r2` entre `conteneurB` et `conteneurC`, **il n'existe aucun réseau entre `conteneurA` et `conteneurC`**. Ainsi il est impossible de créer une commumnication directe entre les deux.

À retenir

En réfléchissant bien à quelles connexions sont *nécessaires* entre nos conteneurs, on peut s'assurer avec nos réseaux virtuels quelles communications inter-conteneurs sont possible/autorisées, et lesquelles ne sont impossibles/interdites.

Ce dernier cas est utile pour éviter que deux processus communiquent entre eux "par erreur" (ou par malice) alors qu'ils ne devraient pas.

10.2. Tutoriel : déployer un service Mealie à l'aide d'un réseau virtuel

On se propose de déployer un service Mealie [https://mealie.io/], un logiciel libre permettant de stocker et partager des recettes de cuisine. *Mealie* est capable de fonctionner avec un système de gestion de base de données de type PostgreSQL [https://fr.wikipedia.org/wiki/PostgreSQL], on souhaite donc déployer *Mealie* avec sa propre base de données *PostgreSQL*.

Avant tout, commencez par créer sur l'environnement d'accueil un répertoire `mealie_deployment` qu'on utilisera pour nos prochains montages de répertoire.

10.2.1. Réseau virtuel

Commençons par préparer un **réseau virtuel** qui permettra à nos deux futurs conteneurs de communiquer entre eux.

Exécutez la commande suivante pour créer un réseau virtuel appelé `mealie_network` :

[sur l'environnement de déploiement distant]

```
$ podman network create mealie_network
```

On peut vérifier que ce réseau a bien été créé avec la commande :

[sur l'environnement de déploiement distant]

```
$ podman network ls
```

Le réseau virtuel devrait apparaître dans la liste, avec son propre identifiant unique et le driver `bridge`.

Nous n'utiliserons pas d'autres drivers réseau que `bridge`.

10.2.2. Conteneur *PostgreSQL*

On peut maintenant préparer le système de gestion de base de données PostgreSQL.

On choisit d'utiliser l'image de conteneur officielle PostgreSQL située sur le Docker Hub [https://hub.docker.com/_/postgres]. En lisant la documentation de l'image, on observe que :

- Le point d'entrée de l'image est l'exécution du logiciel PostgreSQL,
- Les données de la base de données sont stockées dans le répertoire `/var/lib/postgresql/data` du conteneur,
- Par défaut, le logiciel PostgreSQL écoute sur le port 5432 du conteneur,
- Par défaut, lors du premier lancement d'un conteneur, un compte PostgreSQL est créé, ainsi qu'une base de données vide à laquelle ce compte peut accéder. Des variables d'environnement sont disponibles pour configurer cela :
  - `POSTGRES_USER` permet de spécifier le nom du compte PostgreSQL,
  - `POSTGRES_PASSWORD`, permet de spécifier le mot de passe du compte `POSTGRES_USER`,
  - `POSTGRES_DB` permet de spécifier le nom de la base de données créée pour le compte `POSTGRES_USER`.

Dans le répertoire `mealie_deployment` que vous avez préparé, créez un sous-répertoire appelé `db-data` dans lequel nous stockerons les données de la base de données. Utilisez la commande `pwd` pour obtenir le chemin absolu du répertoire `db-data`, et notez ce chemin absolu quelque part.

Exécutez ensuite la commande suivante pour créer votre conteneur PostgreSQL, en version `16`, avec pour nom `mealie_db`, en l'ajoutant sur le réseau `mealie_network`, et en préparant un compte PostgreSQL appelé `mealie` avec pour mot de passe `mealie_password` et pour base de données `mealie_db`.

[sur l'environnement de déploiement distant]

```
$ podman container run --rm \
                       --detach \
                       --name mealie_db \
                       --network mealie_network \
                       --mount type=bind,source="<chemin absolu db-data>
",target="/var/lib/postgresql/data" \
                       --env POSTGRES_USER=mealie \
                       --env POSTGRES_PASSWORD=mealie_password \
                       --env POSTGRES_DB=mealie_data \
                       docker.io/postgres:16
```

Mais puisque PostgreSQL écoute sur le port 5432, **pourquoi n'avons-nous pas effectué une publication de ce port** ? La raison est la suivante : *nous ne souhaitons pas que ce conteneur PostgreSQL soit accessible par n'importe quel logiciel sur le réseau du serveur de déploiement*. À la place, nous souhaitons qu'il soit uniquement accessible par notre application, à savoir *Mealie*, au travers du réseau virtuel `mealie_network` que nous avons préparé.

Vérifiez que le conteneur est bien en exécution avec :

[sur l'environnement de déploiement distant]

```
$ podman container ps
```

Puis vérifiez les logs du conteneur pour s'assurer que tout va bien :

[sur l'environnement de déploiement distant]

```
$ podman container logs mealie_db
```

Vers la fin, vous devriez notamment voir la ligne :

```
PostgreSQL init process complete; ready for start up.
```

Et quelques lignes plus loin :

```
database system is ready to accept connections
```

Vérifiez enfin que le conteneur est bien sur le réseau `mealie_network` à l'aide de la commande :

[sur l'environnement de déploiement distant]

```
$ podman container ps --filter network=mealie_network
```

Si le conteneur est bien dans la liste, c'est que tout est OK !

10.2.3. Conteneur *Mealie*

Il ne nous reste plus qu'à lancer un conteneur Mealie, et de lui permettre d'utiliser notre base de données PostgreSQL située dans le premier conteneur.

Mealie est développé à l'aide d'un projet sur la forge logicielle Github [https://github.com/mealie-recipes/mealie], et utilise le registre de conteneurs Github pour stocker les images de conteneur de l'application [https://github.com/mealie-recipes/mealie/pkgs/container/mealie]. Les images seront donc accessibles à l'aide du registre `ghcr.io`, sous le nom `mealie-recipes/mealie`.

En lisant la documentation de Mealie [https://nightly.mealie.io/documentation/getting-started/introduction/], on trouve les informations utiles suivantes :

- Le point d'entrée de l'image est l'exécution du logiciel *Mealie*,
- Le logiciel écoute par défaut sur le port `9000` du conteneur,
- Le logiciel a besoin de stocker des fichiers (notamment des photos téléversées pour illustrer les recettes de cuisine) dans le répertoire `/app/data/` du conteneur,
- On peut configurer l'accès à la base de données à l'aide des variables d'environnement suivantes :
  - `DB_ENGINE` permet d'indiquer le *type de la base de données*, avec soit la valeur `sqlite`, soit la

valeur `postgres`,

- `POSTGRES_SERVER` permet d'indiquer le *nom d'hôte réseau* du serveur PostgreSQL à utiliser pour stocker les données,
- `POSTGRES_DB` permet d'indiquer le *nom de la base de données* à utiliser au sein du serveur PostgreSQL,
- `POSTGRES_USER` permet d'indiquer le *compte* à utiliser pour se connecter au serveur PostgreSQL,
- `POSTGRES_PASSWORD` permet d'indiquer le *mot de passe* à utiliser pour se connecter au serveur PostgreSQL,
- `POSTGRES_PORT` permet d'indiquer le *port* à utiliser pour se connecter au serveur PostgreSQL.

Avec tout cela en tête, préparez et exécutez la commande suivante, afin de créer un conteneur `Mealie`, en version `v1.0.0-RC1.1`, appelé `mealie`, sur le réseau virtuel `mealie_network`, avec une publication de port et avec la bonne configuration pour se connecter à la base de données PostgreSQL préparée à l'étape précédente :

[sur l'environnement de déploiement distant]

```
$ podman container run --rm \
                       --detach \
                       --name mealie \
                       --network mealie_network \
                       --publish <mon port réseau>:9000 \
                       --env DB_ENGINE=postgres \
                       --env POSTGRES_SERVER=mealie_db \
                       --env POSTGRES_DB=mealie_data \
                       --env POSTGRES_USER=mealie \
                       --env POSTGRES_PASSWORD=mealie_password \
                       --env POSTGRES_PORT=5432 \
                       ghcr.io/mealie-recipes/mealie:v1.0.0-RC1.1
```

en remplaçant `<mon port réseau>` par un port réseau de votre choix de votre plage de port attribuée.

Toutes les informations écrites dans les variables `POSTGRES_` du conteneur *Mealie* doivent absolument correspondre exactement à la configuration effectuée du conteneur PostgreSQL au préalable. On notera par exemple que `POSTGRES_SERVER` correspond au nom donné au conteneur PostgreSQL, car ce nom est également un *nom d'hôte réseau* au sein d'un réseau virtuel `mealie_network`.

Vérifiez que le conteneur est bien en exécution avec :

[sur l'environnement de déploiement distant]

```
$ podman container ps
```

Puis vérifiez les logs du conteneur pour s'assurer que tout va bien :

[sur l'environnement de déploiement distant]

```
$ podman container logs mealie
```

Vers la fin, vous devriez notamment voir la ligne :

```
Uvicorn running on http://0.0.0.0:9000
```

Cette ligne indique que *Mealie* écoute sur le port 9000, mais n'oubliez pas que nous avons publié ce port sur un port différent de l'environnement d'accueil !

Vérifiez enfin que le conteneur est bien sur le réseau `mealie_network` à l'aide de la commande :

[sur l'environnement de déploiement distant]

```
$ podman container ps --filter network=mealie_network
```

Si les deux conteneurs sont bien dans cette liste, c'est que tout est OK !

10.2.4. Tests puis nettoyage

On peut désormais tester notre logiciel déployé. Sur votre poste Linux, ouvrez Firefox, et accédez à l'adresse `http://podman:<mon port réseau>`.

Pour rappel, ici `<mon port réseau>` correspond au port de publication choisi pour le conteneur *Mealie* dans l'étape précédente.

Vous devriez faire face à un écran de connexion du logiciel *Mealie*. Vous pouvez vous connecter avec le compte créé par défaut qui a pour utilisateur `changeme@example.com` et pour mot de passe `MyPassword`.

Ensuite, vous pouvez utiliser le bouton *Créer* puis *Importer une recette par son URL* afin d'importer une recette de cuisine à partir d'une page web. Saisissez par exemple cette URL https://deliacious.com/2020/03/babka-cannelle-vegan.html puis validez. L'import devrait fonctionner, et si vous accédez au menu *Rechercher* à fauche, votre recette devrait être listée.

Félicitations, vous avez effectué votre premier déploiement multi-conteneurs !

On peut maintenant faire un peu de nettoyage avant de passer à la suite :

[sur l'environnement de déploiement distant]

```
$ podman container rm -f mealie
$ podman container rm -f mealie_db
$ podman network rm mealie_network
```

10.3. Exercice : déployer un service Kanboard+MariaDB avec un réseau virtuel

10.3.1. Énoncé

On souhaite à nouveau déployer un service *Kanboard*, tout comme dans le TP précédent. Cette fois-ci, on souhaite utiliser le fait que *Kanboard* est capable de stocker des données dans un système de gestion de base de données compatible MySQL. On souhaite donc que notre nouveau déploiement soit composé d'un conteneur *Kanboard* d'une part, et d'un conteneur compatible MySQL d'autre part.

Concernant la base de données, on choisit d'utiliser l'image de conteneur officielle pour le logiciel MariaDB [https://hub.docker.com/_/mariadb], un système de gestion de base de données compatible MySQL. En lisant la documentation de l'image, on trouve les informations utiles suivantes :

- Le point d'entrée de l'image est l'exécution du logiciel MariaDB,
- Le logiciel écoute par défaut sur le port 3306 du conteneur,
- Le logiciel stocke ses données dans le répertoire `/var/lib/mysql` du conteneur,
- Par défaut, lors du premier lancement d'un conteneur, un compte MariaDB est créé, ainsi qu'une base de données vide à laquelle ce compte peut accéder. Des variables d'environnement sont disponibles pour configurer cela (de manière assez similaire à l'image de conteneur PostgreSQL vue précédemment) :
  - `MARIADB_USER` permet de spécifier le nom du compte MariaDB,
  - `MARIADB_PASSWORD` permet de spécifier le mot de passe du compte `MARIADB_USER`,
  - `MARIADB_DATABASE` permet de spécifier le nom de la base de données créée pour le compte `MARIADB_USER`,
  - `MARIADB_ROOT_PASSWORD` est une variable obligatoire qui permet de spécifier un mot de passe administrateur pour tout le logiciel MariaDB.

Concernant le logiciel Kanboard, même si le contexte est le même que dans le TP précédent, **il nous faut également savoir comment configurer un conteneur pour qu'une base de données externe soit utilisée.** En parcourant la documentation, on peut lire qu'il est possible de configurer un conteneur Kanboard à l'aide d'une unique variable d'environnement :

- `DATABASE_URL` permet de spécifier une URL de base de données qui prend la forme suivante :

  ```
  mysql://<nom utilisateur>:<mot de passe>@<nom d'hôte>:<port>/<nom base de données>
  ```

  avec :

  - `<nom utilisateur>` le nom d'utilisateur MariaDB à utiliser,
  - `<mot de passe>` le mot de passe pour l'utilisateur MariaDB
  - `<nom d'hôte>` le nom d'hôte à utiliser pour se connecter au serveur MariaDB,
  - `<port'hôte>` le port réseau à utiliser pour se connecter au serveur MariaDB,
  - `<nom base de données>` le nom de la base de données à utiliser.

10.3.2. Questions

1. Préparez et exécutez une commande pour déployer un service *Kanboard*, sous la forme d'un conteneur *Kanboard* relié à un conteneur *MariaDB* en suivant avec les contraintes suivantes:
   - Le nom du conteneur *Kanboard* doit être `kanboard_instance`.

- Le nom du conteneur *MariaDB* doit être `kanboard_db`.
- L'image de conteneur *Kanboard* utilisée doit être celle correspondant à l'étiquette `v1.2.32`.
- L'image de conteneur *MariaDB* utilisée doit être celle correspondant à l'étiquette `11.1`.
- L'application *Kanboard* doit être accessible par une publication de port,
- Les données du conteneur *Kanboard* doivent être stockées dans l'environnement d'accueil.
- Les données du conteneur *MariaDB* doivent être stockées dans l'environnement d'accueil.

2. Une fois les conteneurs en exécution, utilisez Firefox depuis votre poste de travail Linux pour accéder à la page d'accueil de *Kanboard*, en utilisant la publication de port que vous avez réalisée.
3. Testez l'application de manière similaire à ce qui était demandé dans le TP précédent.
4. Une fois authentifié dans l'application avec le compte par défaut, cliquez tout en haut à droite sur l'icone utilisateur, puis accédez aux *Préférences*. Dans les informations affichées, vérifiez que le type de base de données est bien **mysql**.

Vous allez constater que les données créées par le conteneur MariaDB ne peuvent pas directement être supprimées de l'environnement d'accueil, car votre compte utilisateur ne possède pas les permissions suffisantes. Rendez-vous vers la fin du sujet de TP pour plus d'informations à ce sujet.

10.4. Exercice : déployer *lldap*, *Kanboard* et *Mealie* à l'aide de réseaux virtuels

Cet exercice a été terminé tardivement et ajouté *après* le début des premières séances pour ce sujet de TP. À ce titre il n'est donc pas obligatoire, mais il reste très intéressant à faire pour approfondir les notions de ce chapitre.

Pour finir ce TP, on souhaite effectuer un déploiement un peu plus complexe qui fait principalement collaborer trois logiciels différents :

- Une instance du logiciel *Kanboard* (adossé à une instance MariaDB comme base de données)
- Une instance du logiciel *Mealie* (adossé à une instance PostgreSQL comme base de données)
- Et enfin une instance du logiciel _lldap, un logiciel qui fournit un service d'authentification de type LDAP.

Dans cet exercice, l'idée est de configurer *Kanboard* et de configurer *Mealie* de manière à ce que chacun des deux logiciels délègue sur l'instance *lldap* pour l'authentification d'un utilisateur. Ainsi, un compte utilisateur créé au sein de l'instance *lldap* pourra à la fois être utilisé pour se connecter à *Mealie*, et pour se connecter à *Kanboard*.

On souhaite déployer chaque logiciel sous la forme d'un conteneur, soit un total de 5 conteneurs à déployer !

10.4.1. Présentation de LDAP et *lldap*

Lightweight Directory Access Protocol (LDAP) [https://fr.wikipedia.org/wiki/Lightweight_Directory_Access_Protocol] est une norme qui permet à un logiciel serveur d'agir comme un annuaire d'utilisateurs (le *serveur* LDAP), et qui permet à d'autres logiciels (les *clients* LDAP) de déléguer la gestion et l'authentification de leurs utilisateurs à un serveur LDAP existant. C'est une manière répandue de centraliser un ensemble unique de comptes utilisateurs, qui permettront de se connecter de manière unifier à tout un ensemble de services informatiques.

lldap [https://github.com/lldap/lldap] (qui signifie *Light LDAP implementation for authentication*) est un logiciel libre qui permet de déployer un serveur LDAP simplifié et suffisant pour gérer un annuaire léger d'utilisateurs, et pour gérer l'authentification d'utilisateurs sur différents services.

En lisant la documentation *lldap* [https://github.com/lldap/lldap/blob/main/docs/install.md#with-docker], on découvre les informations utiles suivantes :

- une image de conteneur *lldap* est disponible sur le registre de conteneur Docker Hub (donc `docker.io`) sous le nom `lldap/lldap`.
- Une étiquette valide pour cette image est `2025-10-14`
- Le point d'entrée de cette image est le démarrage d'un processus *lldap*
- Au sein du conteneur, le processus *lldap* stocke ses données au sein du répertoire `/data`.
- Par défaut, un processus *lldap* écoute simultanément sur deux ports réseaux :
  - le port `17170`, via lequel on peut se connecter à *lldap* à l'aide d'un navigateur web, pour pouvoir utiliser l'interface d'administration de *lldap* ; cette interface permet de créer un compte, lister les comptes disponibles, etc.
  - le port `3890`, via lequel un autre logiciel peut se connecter à *lldap* à l'aide du protocole LDAP, et qui permet à cet autre logiciel de déléguer à *lldap* l'authentification utilisateur.
- *lldap* doit être configuré à l'aide des variables d'environnement suivantes :
  - `LLDAP_JWT_SECRET` : doit contenir une chaîne de caractères générée aléatoirement, utilisée à des fins cryptographiques par *lldap*
  - `LLDAP_KEY_SEED` : tout comme `LLDAP_JWT_SECRET`, doit contenir une chaîne de caractères générée aléatoirement, utilisée à des fins cryptographiques par *lldap*
  - `LLDAP_LDAP_USER_PASS` : mot de passe du compte administrateur (dont l'identifiant est `admin`)
  - `LLDAP_LDAP_BASE_DN` : définit le *distinguished name* de base de l'annuaire LDAP, et s'écrit sous la forme `dc=example,dc=com` ou bien juste `dc=example`

10.4.2. Étape 1 : préparer le plan de déploiement

Avant de vous lancer, répondez soigneusement aux questions suivantes :

- Quels sont les conteneurs qu'il va falloir créer ?
- Quels sont les réseaux virtuels qu'il va falloir créer ?

Pour simplifier légèrement, on peut considérer que tous conteneurs ayant besoin de contacter *lldap* peuvent être mis sur un même réseau virtuel.

- Quels sont les publications de port qui vont être nécessaires ?

Il ne faut pas exposer tous les ports réseau sur le réseau local ! Réfléchissez bien à ceux qu'il est nécessaire de publier, et si vous avez un doute là dessus, demandez à votre intervenant de TP.

- Quels sont les montages de fichiers/répertoires qui vont être nécessaires ?

10.4.3. Étape 2 : créer tous les réseaux virtuels nécessaires

Pour chaque réseau virtuel anticipé à l'étape précédente, le créer avec podman.

10.4.4. Étape 3 : déployer *lldap*

Déployez une instance *lldap* qui suit votre plan de déploiement, et avec également les contraintes suivantes :

- le conteneur doit s'appeler *lldap_instance*
- les données de *lldap* doivent être persistées dans l'environnement de déploiement
- vous utiliserez les valeurs suivantes pour les différentes variables d'environnement :
  - `LLDAP_JWT_SECRET` : la valeur de votre choix,
  - `LLDAP_KEY_SEED` : la valeur de votre choix,
  - `LLDAP_LDAP_BASE_DN` : la valeur 'dc=monannuaire'
  - `LLDAP_LDAP_USER_PASS` : la valeur `motdepasse`

Connectez vous ensuite à l'interface web d'administration de *lldap*, connectez vous avec le compte `admin`, puis utilisez le bouton *Create a user* pour créer un compte utilisateur de votre choix

Ne perdez pas le mot de passe que vous aurez choisi, il sera nécessaire aux étapes suivantes !

10.4.5. Partie 4 : déployer Kanboard

En suivant votre plan de déploiement, déployez une base de données PostgresQL, ainsi qu'une instance Kanboard.

Pour le déploiement de Kanboard, ajoutez les variables d'environnement suivantes pour configurer l'authentification au travers du serveur *lldap* :

- `LDAP_AUTH` : mettre la valeur `true`
- `LDAP_BIND_TYPE` : mettre la valeur `user`
- `LDAP_SERVER` : mettre une valeur de la forme `ldap://<nom d'hôte>:<numéro port>`, où `<nom d'hôte>` est le nom d'hôte permettant d'atteindre le serveur LDAP, et `<numero port>` le numéro de port auquel se connecter sur ce serveur pour contacter le serveur LDAP
- `LDAP_USERNAME` : mettre la valeur `uid=%s,ou=people,dc=monannuaire`
- `LDAP_USER_BASE_DN` : mettre la valeur `ou=people,dc=monannuaire`
- `LDAP_USER_FILTER` : mettre la valeur `uid=%s`

Ici nous vous fournissons la solution pour la plupart des valeurs, car configurer une connexion LDAP est une entreprise assez tortueuse.

Une fois votre instance Kanboard démarrée, testez de vous connecter à l'aide du compte utilisateur que vous avez créé dans la partie 3. Si cela fonctionne, c'est bon !

10.4.6. Partie 5 : déployer Mealie

En suivant votre plan de déploiement, déployez une base de données MariaDB, ainsi qu'une instance Mealie.

Pour le déploiement de Mealie, ajoutez les variables d'environnement suivantes pour configurer l'authentification au travers du serveur *lldap* :

- `LDAP_AUTH_ENABLED` : mettre la valeur `true`
- `LDAP_SERVER_URL` : mettre une valeur de la forme `ldap://<nom d'hôte>:<numéro port>`, où `<nom d'hôte>` est le nom d'hôte permettant d'atteindre le serveur LDAP, et `<numero port>` le numéro de port auquel se connecter sur ce serveur pour contacter le serveur LDAP
- `LDAP_BASE_DN` : mettre la valeur `ou=people,dc=monannuaire`
- `LDAP_QUERY_BIND` : mettre la valeur `cn=admin,ou=people,dc=monannuaire`
- `LDAP_QUERY_PASSWORD` : mettre la valeur `motdepasse` (à savoir le mot de passe du compte `admin` créé au lancement de *lldap*)
- `LDAP_ID_ATTRIBUTE` : mettre la valeur `uid`
- `LDAP_NAME_ATTRIBUTE` : mettre la valeur `uid`
- `LDAP_MAIL_ATTRIBUTE` : mettre la valeur `mail`

À nouveau ici nous vous fournissons la solution pour la plupart des valeurs, car configurer une connexion LDAP est une entreprise assez pénible.

Une fois votre conteneur démarré, accédez à Mealie à l'aide d'un navigateur, et tentez de vous connecter à l'aide du compte utilisateur préparé en partie 3. Si vous arrivez à vous connecter, c'est tout bon !

10.5. Prise de recul sur le travail réalisé dans les séances de TP jusqu'ici

Prenons un peu de recul.

La première chose à remarquer est que, **malgré les déboires que vous avez pu rencontrer, les différents déploiements que vous avez réalisé en TP n'ont normalement pas représenté une quantité de travail très importante**. Certes il peut être laborieux de préparer les bonnes commandes et de créer les bons conteneurs dans le bon ordre avec les bonnes configurations – et c'est pour cela que nous avons tout un cours ! – mais *le nombre d'actions à effectuer pour faire le déploiement à proprement parler est généralement très bas* (seulement quelques commandes par situation).

En outre, **vous avez pu déployer ces logiciels sans avoir la moindre connaissance technique de leur fonctionnement interne**, ou des technologies qu'ils utilisent, ou des dépendances qu'ils nécessitent. Sans le savoir, vous avez déployé un logiciel développé en C (nginx), un autre développé en Python + JavaScript (mealie), et un autre développé en PHP (Kanboard), le tout alors que chacune de ces technologies gère totalement différemment ses dépendances logicielles. Vous n'avez jamais eu besoin d'installer un moteur d'exécution PHP, ou un interpréteur Python, ou des bibliothèques natives pour C. Sacrée prouesse !

Tout cela illustre la grande efficacité de la conteneurisation pour le déploiement des logiciels. C'est simple : faire tous ces déploiements n'aurait simplement pas été possible dans un TP de M1 Informatique sans utiliser cette méthode, car cela aurait nécessité de votre part une bien plus grande maîtrise du travail d'administrateur·ice système.

Bien entendu, ce qui a mis ces déploiements à votre portée est le travail important réalisé en amont pour préparer toutes les images de conteneurs que vous avez utilisées. La suite des travaux pratique consistera donc à découvrir comment préparer de telles images pour les logiciels que vous souhaitez déployer.

10.6. Exercices additionnels

Pour ceux qui auraient terminé ce TP avant le temps alloué en séance, voici quelques exercices supplémentaires moins guidés et plus "libres" que ceux demandés précédemment.

10.6.1. Déploiement d'un serveur ForgeJo

ForgeJo [https://forgejo.org/] est une forge logicielle permettant de gérer des développements logiciels. On peut y créer des projets avec référentiel Git et avec des outils d'organisation classiques (tickets, navigation dans le code, revues de code, etc.). C'est donc une alternative au logiciel Gitlab, plus légère mais avec également moins de fonctionnalités.

On notera que ForgeJo a besoin d'écouter sur deux ports réseau différents :

- le port `3000`, qui permet de se connecter à ForgeJo à l'aide d'un navigateur web
- le port `22`, qui permet de se connecter à ForgeJo à l'aide d'une connexion SSH, afin de pouvoir utiliser Git à l'aide du protocole SSH

Pour ce dernier exercice, vous effectuerez un déploiement d'une instance ForgeJo, avec les contraintes suivantes :

- on utilisera une base de données MariaDB (compatible MySQL)
- on configurera ForgeJo pour qu'il utilise correctement le port de publication SSH que vous aurez choisi, à l'aide de l'option `SSH_PORT`
- on utilisera une image avec une étiquette possédant le suffixe `-rootless`, pour s'assurer d'avoir une image compatible Podman
- plutôt que de configurer le logiciel à l'aide de variables d'environnement, on utilisera un fichier de configuration `app.ini`, en se basant sur ce fichier exemple [https://codeberg.org/forgejo/forgejo/src/branch/forgejo/custom/conf/app.example.ini] fourni par les développeurs de ForgeJo.

Cette page de la documentation [https://forgejo.org/docs/latest/admin/config-cheat-sheet/] vous explique comment ForgeJo peut être configuré, et résume toutes les options permettant de configurer le logiciel.

Cette autre page de la documentation [https://forgejo.org/docs/latest/admin/installation/] vous donne des informations sur les images de conteneur ForgeJo disponibles, en particulier un exemple de référence d'image. Elle comporte également des exemples de fichiers écrits avec le langage Compose [https://www.compose-spec.io/], non étudié dans ce cours pour le moment.

Pour valider votre déploiement, vous pouvez :

- créer un compte sur votre instance ForgeJo,
- ajouter votre clé publique SSH à votre compte,
- créer un projet à l'aide de ce compte,
- effectuer un clone Git par SSH de ce nouveau projet,
- ajoutez un fichier, puis effectuez une validation (commit) et envoi (push)

10.6.2. Déploiement de Simplestblog

Faites un déploiement par vous même d'une instance de Simplestblog [https://gitlab.univ-nantes.fr/gl/developpement_exploitation/resources/simplestblog]

Vous pouvez bien entendu vous baser sur les commandes déjà vues pendant le cours.

Sur la page du projet, vous pouvez accéder à l'ensemble des images de conteneurs disponibles en vous rendant dans *Déploiement → Registre de conteneur*.