Linux Services et administration

David Zarebski

IMIE

2026



[0.0] ⋙ Programme I

- Objectifs
- Préparation de l'environnement
  - Installation d'Open Suse leap
  - Récupération de vikunja
- Systemd
  - Bases
  - Hardening
- Cron: tâches journalisées
- Exposition par un reverse-proxy nginx
- Conteneurisation Linux
  - Différentes technos pour différents usages
  - Prise en main de docker

Objectifs

[1.0] ⋙ Objectifs généraux

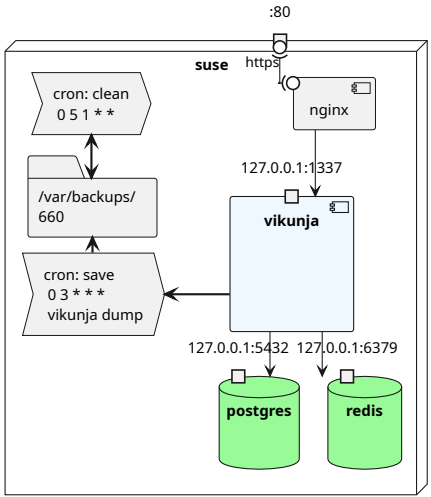
Prise en main de:

- `systemd`
- `journald`
- `cron`

[1.0] ⋙ Exercice fil-rouge

Sur une VM Open Suse Leap v16, nous allons installer un applicatif du nom de vikunja

- comme un service (`systemd`), lancé au démarrage et dépendant
  - d'une base de donnée `postgres`
  - d'un server de cache `redis`
- exposé en HTTP via `nginx` (selon le temps disponible)
- nous veillerons à l'aggregation des logs dans `journald`
- avec une tâche journalisée de sauvegarde de la base de donnée (`cron`)
- une tâche de nettoyage de ces sauvegardes (rétention 2 mois)



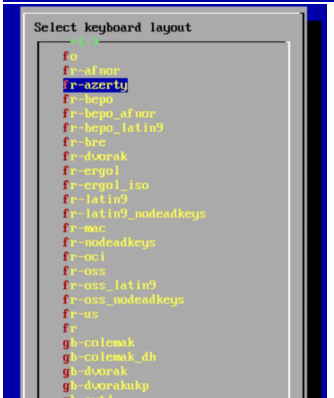
Préparation de l'environnement

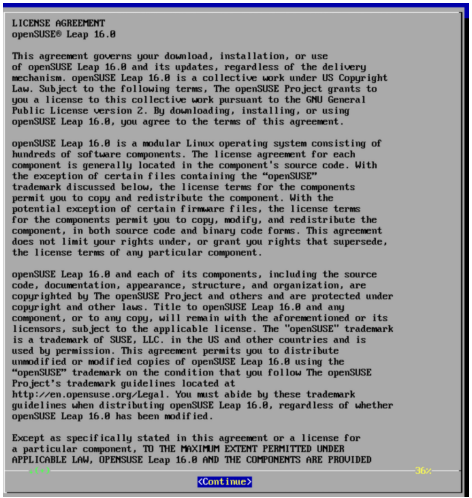
Préparation de l'environnement
≫Installation d'Open Suse leap

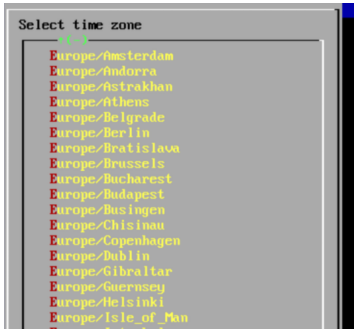
Des images VM prêtes à l'emploi sont disponibles ici:
https://get.opensuse.org/leap/16.0/?type=server#download

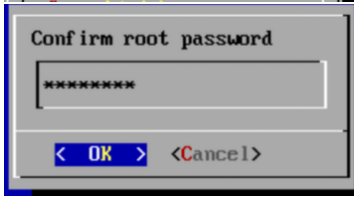
[2.1] ⋙ Setting des locale, timezone et MDP root

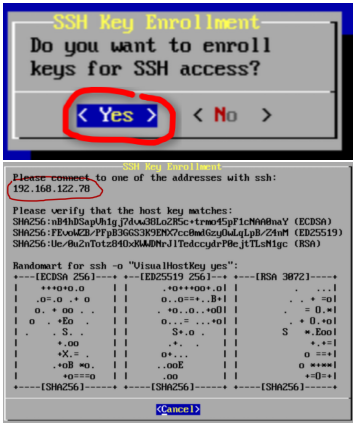








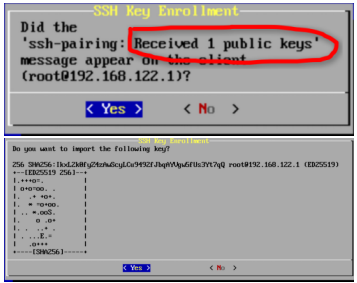


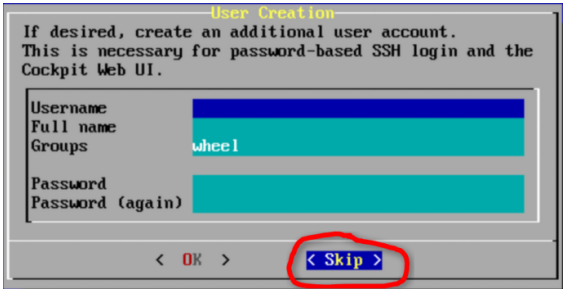


Suse a mis en place un mécanisme assez commode pour **importer les clefs**. Il suffit de

```
ssh root@<VM_IP>
```

Comme quoi, on peut faire sans `cloud-init`





Nous ferons l'ensemble des TP comme **root** donc pas besoin d'un utilisateur additionel

```
ssh root@<VM_IP>
```

`zypper` est le **gestionnaire de packages**.

```
zypper update
```

Les packages sont des `.rpm`

Mandatory Access Control: OpenSuse utilise `SELinux`

```
sestatus
SELinux status:                 enabled
SELinuxfs mount:                /sys/fs/selin
SELinux root directory:         /etc/selinux
Loaded policy name:             targeted
Current mode:                   enforcing
```

firewalling Le gestion du firewall se fait par défaut avec `firewalld`

```
firewall-cmd --info-zone=public
public (default, active)
  target: default
  ingress-priority: 0
  egress-priority: 0
  icmp-block-inversion: no
  interfaces: enp1s0
  sources:
  services: dhcpv6-client ssh
```

- Beaucoup de similitudes avec **RedHat**, **Fedora**, *etc*

Préparation de l'environnement
≫Récupération de vikunja

Nous allons partir de la procédure d'installation manuelle décrite ici

- téléchargement d'une archive 2.5.0 compatible
- décompression de l'archive en `/opt/vikunja`

[2.2] ⋙ Création d'un user

Notre applicatif tournera avec un utilisateur `vikunja` (PID:GID = 901)

```
useradd -d /opt/vikunja -m -r -s /bin/false vikunja
cat /etc/passwd
...
vikunja:x:901:901::/opt/vikunja:/usr/sbin/nologin
```

> Sauf cas exceptionnel, un service exposé à des utilisateurs ne doit **JAMAIS** tourner avec `root`, raison d'être de cet utilisateur

Systemd

Systemd
≫Bases

On sait déjà que les processus dépendent de parents les ayant lancés.

```
root    921 ...  ?      Ss  0:00 sshd: /usr/sbin/sshd -D [listener] 0 of 10-10
root    996 ...  ?      Ss  0:00  \_ sshd-session: root [priv]
root   1011 ...  ?      S   0:00      \_ sshd-session: root@pts/0
root   1013 ...  pts/0  Ss  0:00          \_ -bash
root   1817 ...  pts/0  R+  0:00              \_ ps faux
```

Ici, `ps faux` ▷ `bash` ▷ `sshd-session root@pts` ▷ `sshd-session: root [priv]` ▷ `sshd: /usr/sbin/sshd`

Puisque l'on ne peut remonter à l'infini, **par quoi commence ce graph**?

[3.1] ⋙ Le premier processus du user-space

Un processus un peu special occupe toujours le PID 1

```
root 1 ... Ss 14:38 0:01 /usr/lib/systemd/systemd --switched-root --system --
```

Il n'est lancé par aucun processus du **user-space**. Si l'on regarde les options passées au **kernel** par le **bootloader**

```
cat /boot/grub2/grub.cfg | grep -B 5 systemd
...
linux   /boot/vmlinuz-6.12.0-160000.35-default
   root=UUID=41b87736-08cc-6bd0-85de-d844cd161676 ${extra_cmdline}
   rw quiet systemd.show_status=1 console=ttyS0,115200 console=tty0 security=
...
```

Il ressort que le **kernel** lance **systemd**, confiant à ce dernier la responsabilité de **démarrer les autres services du user-space**

- **Système d'initialisation** moderne utilisé par la plupart des distributions Linux
- Remplace l'ancien système SysVinit
- Gère les services, le monitoring du système et la gestion des dépendances

> Aux antipodes de la philosophie UNIX

- ⇒ sources de critiques dans l'écosystème
- certaines distros résistent toujours à l'envahisseur

systemd	OpenRC
Parallélisation native	Séquentiel (optionnellement parallèle)
Dépendances déclaratives	Scripts init avec vérifications
Journalisation intégrée (journald)	Logs séparés
Units unifiées	Scripts spécifiques par service
Gestion d'état complexe	Plus simple

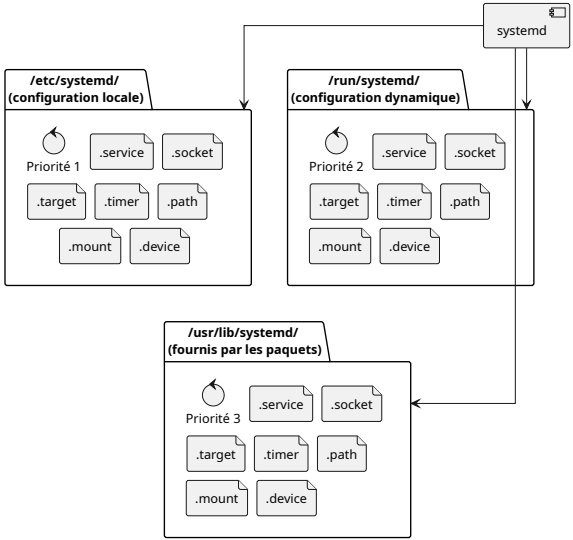
- démarrer/arrêter/redémarrer un service:
  `systemctl start|stop|restart <unit_name>`
- activer/désactiver une unité au démarrage: `systemctl enable|disable <unit_name>`
- recharger la définition des units: `systemctl daemon-reload`
- lire les logs d'une unités: `journalctl -u <unit_name>`

`systemd` charge ses **units**

- à partir de ces trois emplacements
- dans cet ordre de priorité
- en incluant tout fichier portant ces extensions

...

sachant que `systemd` cache en mémoire ces informations, toute édition doit s'accompagner de `systemctl daemon-reload`



- **Démarrage parallèle:** les units non-dépendantes démarrent simultanément
- **Cycles de vie:** chaque unit a un état (`active`, `inactive`, `failed`, `activating`)
- **Ordre de démarrage:**
  1. Units système critiques (mount, network)
  2. Units de services
  3. Units de login (getty, graphical)
- **Arrêt ordonné:** respect de l'ordre inverse des dépendances

[3.1] ⋙ Types d'Units et structure minimale

Liste non exhaustive

- **.service:** services et applications
- **.target:** groupes d'units (équivalent des runlevels)
- **.socket:** sockets pour activation à la demande
- **.timer:** planificateur de tâches (équivalent `cron`)
- **.mount:** points de montage
  - et oui, `systemd` parse `/etc/fstab`
- **.path:** surveillance de chemins de fichiers
- **.device:** périphériques système

```
[Unit]
Description=usbguard

[Service]
ExecStart=/usr/sbin/usb-daemon

[Install]
WantedBy=multi-user.target
```

Exemple minimal de `[Service]`.
selon l'unité vous trouverez:

- `[Timer]`
- `[Mount]`
- ...

[3.1] ⋙ Structure d'un fichier Unit et principaux champs

```
[Unit]
Description=Description du service
After=network.target
Requires=postgresql.service

[Service]
Type=simple
User=www-data
ExecStart=/usr/bin/mon-service
ExecStartPre=/usr/bin/préparation

[Install]
WantedBy=multi-user.target
```

champs principaux

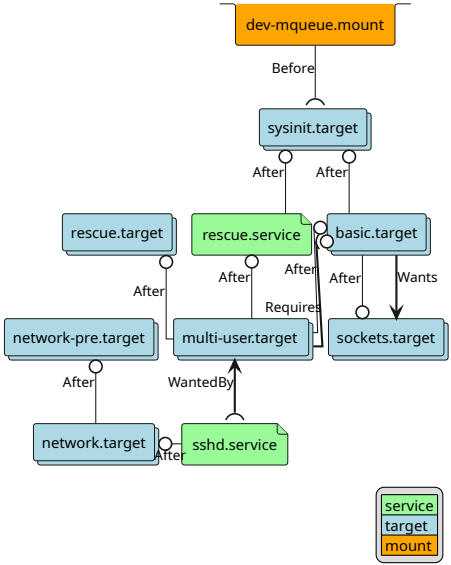
- **Description:** description textuelle du service
- **After=:** démarre APRÈS les units spécifiées
- **Before=:** démarre AVANT les units spécifiées
- **Requires=:** dépendance forte (échec = échec du service)
- **Wants=:** dépendance douce (échec ignoré)
- **Documentation=:** URL de documentation

[3.1] ⋙ Illustration de la dépendance entre des unités

Si l'on remonte la chaine de dépendance de `sshd`, sans suprise:

- ≫ dépend des services liés à `network.target`

`systemctl list-dependencies sshd.service` ne livre qu'une vue partielle



[3.1] ⋙ [Service] - Champs d'Execution

```
[Unit]
Description=Description du service
After=network.target
Requires=postgresql.service

[Service]
Type=simple
User=www-data
ExecStart=/usr/bin/mon-service
ExecStartPre=/usr/bin/préparation

[Install]
WantedBy=multi-user.target
```

- **Type=:** simple, forking, notify, oneshot, dbus
- **User=** exécution avec ce `UID`
- **Group=:** exécution avec ce `PID`
- **ExecStart=:** commande principale
- **ExecStartPre=:** commandes avant ExecStart
- **ExecStartPost=:** commandes après ExecStart
- **ExecStop=:** commande d'arrêt
- **ExecReload=:** commande de rechargement
- **WorkingDirectory**

path de notion de `PATH` ⇒ utilisez des **chemins absolus**

- `ReadWritePaths` chemins accessibles en écriture
- `ReadOnlyPaths` chemins en lecture seule
- `InaccessiblePaths` chemins complètement inaccessibles
- `NoNewPrivileges` empêche l'escalade de privilèges
- `PrivateTmp` espace temporaire isolé
- `ProtectSystem` protège les systèmes de fichiers
  - **yes**: RO `/usr`, `/boot`, `/efi`
  - **full**: RO `/usr`, `/boot`, `/efi`, `/etc`
  - **scrict**: RO `/` sauf `/dev`, `/proc`, `/sys` (et `ReadWritePaths`)

[3.1] ⋙ Avec ces éléments, créons l'unité de vikunja

Installation des dépendances

Après avoir exporté `vikunja` et sa configuration dans `/opt/vikunja/`

```
zypper install postgresql-server, redis
```

Analyse des units:

- **/usr/lib/systemd/system/redis@.service**. Nous cherchons:
  - edition de **/var/lib/pgsql/data/pg_hba.conf** =>

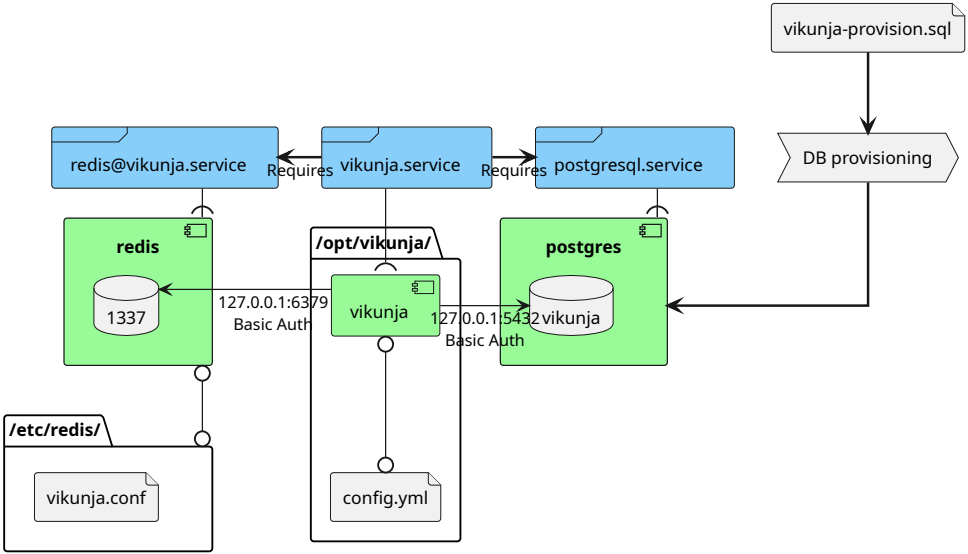
    ```
    host    all    all    127.0.0.1/32    password
    ```

  - comment initializer une BDD: https://www.postgresql.org/docs/current/app-initdb.html
- **/usr/lib/systemd/system/postgresql.service**
  - identification et identification des confs pertinentes

Création de notre [Service]

Dans **/etc/systemd/system/**, création de `vikunja.service` que nous allons confiner à **/opt/vikunja/**

[3.1] ⋙ Vue d'ensemble de l'attendu



Systemd
≫Hardening

- prévoir le redémmarage en cas d'échec (`Restart`)
- nécessite d'autres services ⇒ dépendance explicite des unités
- sauf cas exceptionnel, un service exposé à des tiers ne doit **jamais tourner avec root**
  - compte locaux spécifiques à chaque service
- principe de **moindre privilège**:
  - ne doit pouvoir lire/écrire que le scrict minimum
  - `ProtectSystem`
  - capabilities minimales (voir ci-après)

Niveau paranoïa, on peut aller plus loin que cela en analysant notre unit avec `systemd-analyze security vikunja` et nous rendre compte de ce qui manque

[3.2] ⋙ Hardening - Isolation et Privilèges (quelques exemples)

```
[Service]
NoNewPrivileges=true

PrivateDevices=true

PrivateTmp=true

PrivateNetwork=false

MemoryLimit=1G
```

- désactive les `setuid` `setgid` lors des appels `execve`
- cache le hardware en exposant un `/dev` minimal. ∅ `CAP_MKNOD`
- `/tmp` et `/var/tmp` isolés du reste du système (basé FS `namespace`)
- **NIC** virtuelle dédiée (un peu comme les network docker)
- limitation des resources matérielles accessibles (`cgroups`) ⇒ **OOM Error**

```
[Service]
# Réduire les capabilities
CapabilityBoundingSet=CAP_NET_BIND_SERVICE

# Namespaces
ProtectHome=true
ProtectKernelTunables=true
ProtectKernelModules=true
ProtectKernelLogs=true
RestrictRealtime=true
RestrictSUIDSGID=true
RestrictNamespaces=yes

# Appels systèmes
SystemCallFilter=@system-service
```

[3.2] ⋙ Exercice

Durcissons le service à l'aide de ces options. Nous allons prioriser:

- la limitation des capacités (`SystemCallFilter`, `CapabilityBoundingSet`)
- restriction de la racine `RootDirectory`

expérimenter avec

- `PrivateNetwork`
- `RestrictAddressFamilies`

à la recherche d'un compromis entre:

- sécurité: `systemd-analyze security vikunja`
- bon fonctionnement `journalctl -r -u vikunja`

Cron: tâches journalisées

`cron` est un scheduler basé temps que l'on retrouve sur tout système basé UNIX. Il permet de planifier l'**exécution récurrente** de commandes ou de scripts shell. Parce qu'il dépend du temps, `cron` nécessite l'accès à un serveur NTP (dans une configuration standard)

Il existe plusieurs implémentations de ce scheduler, la plus commune étant `cronie` (que nous allons **installer**).

`systemd` couvre également ce besoin par le biais de ses `timers` mais nous allons nous concentrer sur `cron`

> dans ce contexte, '*tâches journalisées*' et '*jobs*' seront des synonymes

[4.0] ⋙ Cron: exprimer la récurrence

Dans un fichier de configuration `cron`, une tâche se specifie comme suit

```
*/5 9-16 * 1-5,9-12 1-5 ~/bin/i_love_cron.sh
```

et se décompose de la manière suivante

$$\underbrace{*/5}_{minutes}\ \underbrace{9-16}_{heures}\ \underbrace{*}_{jours}\ \underbrace{1-5,9-12}_{mois}\ \underbrace{1-5}_{\text{jour de la semaine}}\ \underbrace{\sim /bin/i_love_cron.sh}_{script/commande}$$

Ici nous disons *exécute* `~/bin/i_love_cron.sh`

- du lundi au vendredi
- de Janvier à Mai et Septembre à Décembre
- de 9H à 16H
- toutes les 5 minutes

pattern	correspondance
`<d>-<d>`	range
`<d>,<d>`	liste
`*`	$\forall x$
`*/<d>`	$\forall x \mid x \equiv 0 \pmod{d}$

Commme tous processus, les jobs `cron` appartiennent à des users (`UID`).

utilisateurs

Chaque utilisateur peut définir ses tâches journalisées

- éditées via `crontab -e`
- lues via `crontab -l`

pattern

```
* * * * * <script/cmd>
```

système

- exécutées par `<user>`
- définies en:
  - `/etc/cron.d`
  - `/etc/crontab`

pattern

```
* * * * * <user> <script/cmd>
```

[4.0] ⋙ Cron: $PATH

Le **contexte d'exécution** d'un job diffère de celui d'un shell ouvert par l'utilisateur cible

Pour le mettre en évidence, nous pouvons créer en `/etc/cron.d/uptime.cron` un job qui:

- pour chaque minute
- ajoutera sur `/tmp/uptime.log`
- une ligne constituée de:
  - la `date` au format rfc3339 (granularité : date)
  - les infos renvoyées par `uptime`

format attendu

```
...
2026-09-02  10:07:00  up   0:40,  1 user,  load average: 0.01, 0.01, 0.00
2026-09-02  10:08:00  up   0:41,  1 user,  load average: 0.00, 0.00, 0.00
```

inutile de redémmarer `cron.service`: ce dernier détecte automatiquement les changements de */etc/cron.d* par **inode**

Il est donc nécessaire d'utiliser des chemins absolus...

```
* * * * * root /usr/bin/echo "$(/usr/bin/date --rfc-3339=date)
            $(/usr/bin/uptime)" >> /tmp/uptime.log
```

... à moins de ne préciser le `PATH`

```
PATH=/usr/bin
* * * * * root echo "$(date --rfc-3339=date) $(uptime)" >> /tmp/uptime.log
```

> En soi, même logique que dans un shell: `printenv | grep PATH` liste les dossiers dans lesquels chercher les binaires

[4.0] ⋙ Cron: $USER

Nous allons maintenant tester la possibilité de définir des jobs pour un utilisateur quelconque en créant une tâche `/etc/cron.d/whoami.cron`:

- exécutée chaque minute
- par vikunja
- ajoutant `whoami` à `/tmp/whoami.log`

> Quelques pièges ici: pensez à consulter `systemctl status cron` pour comprendre

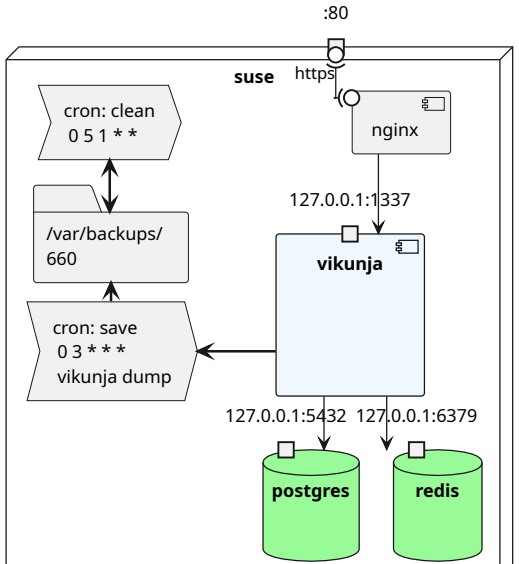
vikunja n'a pas les droits d'écriture sur `/tmp`. Il convient alors de

```
touch /tmp/whoami.log
chmod 777 /tmp/whoami.log
```

Nous pouvons alors

```
PATH=/usr/bin
* * * * * vikunja echo "$(whoami)" >> /tmp/whoami.log
```

[4.0] ⋙ Création de nos tâches journalisées



Deux tâches journalisées

- **save**:
  - `/etc/cron.d/vikunja-save.cron`
  - tous les jours, à 3H
  - exécution de `vikunja dump`
  - écriture des zips dans `/var/backups`
- **clean**:
  - `/etc/cron.d/vikunja-clean.cron`
  - tous les 1er du mois, à 5H
  - supprimer les archives de `/var/backups` vieilles de plus de 2 mois

NB: Il n'y a pas de notion de "**dossier courant**" en `cron` ⇒ il va falloir ruser et une date = *nombre de secondes depuis l'epoch*

[4.0] ⋙ Correction cron: save

Rien n'oblige à faire les sauvegardes avec l'utilisateur **vikunja**

Pour le changement de dossier courant, on peut recourir à un script forcer les deux commandes à fonctionner dans le **même sub-shell** avec `()`

`/etc/cron.d/vikunja-save.cron`

```
0 3 * * * root (cd /opt/vikunja && ./vikunja dump -p /var/backups)
```

[4.0] ⋙ Correction cron: clean

/etc/cron.d/vikunja-clean.cron

```
0 5 1 * * vikunja find /var/backups -maxdepth 1 \
   -name 'vikunja-dump_*.zip' ! -newermt '2 month ago' -delete
```

Exposition par un reverse-proxy nginx

```
zypper install nginx
```

Crée le service `nginx.service` correspondant. Pour pouvoir assurer son rôle de reverse-proxy, `nginx` aura besoin:

1. que `SELinux` le lui permette

```
setsebool -P httpd_can_network_connect on
```

> Nous reparlerons de **SELinux** et, plus généralement, de *Mandatory Access Control* dans de futurs modules

2. Que le firewall autorise les packets TCP entrants sur :80

```
firewall-cmd --zone=public --add-service=http --permanent
firewall-cmd --reload
```

[5.0] ⋙ Configuration du VHost

Il nous suffit de créer `/etc/nginx/vhosts.d/site.conf`

```
server {
    listen       80;
    server_name  ${DOMAIN};
    location / {
        proxy_pass http://127.0.0.1:1337;
        client_max_body_size 20M;
        proxy_set_header Host \$host;
        proxy_set_header X-Real-IP \$remote_addr;
        proxy_set_header X-Forwarded-For \$proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto \$scheme;
    }
}
```

et redémarrer le service

Conteneurisation Linux

Conteneurisation Linux
≫Différentes technos pour différents usages

Depuis l'émergence de **docker** en 2013, nombreuses alternatives sont apparues.

Dans les quelques slides à venir, nous:

- présenterons les principaux moteurs de conteurisation
- indentifierons ce qu'ils ont en commun
- pointerons leurs principales différences

> Nous laisserons sciemment les conteneurs BSD (`jails`) et les différentes méthodes de sandboxing de côté.

[6.1] ⋙ Motivations et raison du succès de ce paradigme

Délivrabilité

Une solution efficace au **dependency hell**: l'applicatif vient avec

- ses dépendances systèmes
- ses scripts d'init
- ses librairies, ses versions

qui ne rentreront pas en conflit avec celles d'autres applicatifs. *build once, runs everywhere*

> promesses tenues

Sécurité

Parce qu'il s'agit d'un environnement d'exécution minimal isolé du reste du système, **docker** a pu être assimilé à du sandboxing

- ne lit que ce qui se trouve dans le conteneur
- ne voit pas les autres processus,
- pas d'accès aux devices

> Plus mitigé, comme nous le verrons

[6.1] ⋙ Primitives systèmes impliquées dans la conteneurisation I

L'ensemble des technologies que nous allons voir dépendent du noyau Linux pour au moins 2 raisons triviales:

- les processus d'un conteneur communiquent avec **le même noyau** que ceux de l'hôte (un conteneur n'est pas une VM)
- l'isolation de ces processus repose sur certaines fonctions exposées par le noyau:
  - restriction des ressources allouées au processus: `cgroups`
  - création d'une *fausse racine* du système de fichier pour un processus: `pivot_root`
  - possibilité de créer des groupes de processus ne pouvant pas voir les autres: `namespace`

En soi, avant **docker**, il était déjà possible d'encapsuler des processus à l'aide de ces primitives appelées *à la mano*. La création d'un `chroot` était l'exercice canonique pour les jeunes admin-sys Linux

Cgroups

Permettent, pour un groupe de processus, de:

- limiter le temps CPU alloué
- limiter la RAM (déclenchement d'un **OOMKilled**)
- le nombre de PID possibles ($\rightarrow$ éviter une *fork bomb*)
- brider les opérations I/O (e.g. limiter la bande passante d'un disque)
- vivent en `/sys/fs/cgroup` (allez y jeter un oeil: c'est instructif)

Namespace

- Isolation de la *vue* d'une ressource permise par `clone()` ⇒ le processus ne voit que des versions modifiées des

Type	Isole
mnt	Points de montage
uts	Hostname
pid	Arbre de PID
net	Interfaces, routes, pare-feu
user	UID/GID, capabilities
cgroup	Racine cgroup visible
time	Horloges monotone / boot

. . . à moins de ne donner des privilèges spécifiques au conteneur

Chroot et pivot_root

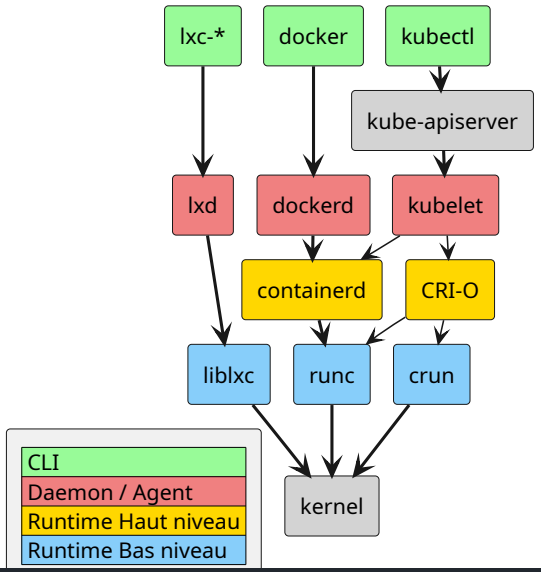
Pour le processus impliqué, la **racine du système de fichier** n'est pas la racine véritable. `chroot()` * vieux mécanisme UNIX des 70' * faillible: * multiples `cd ../` * appel de `chroot()` depuis `chroot()`

Depuis 2025, les moteurs de conteneur utilisent une fonction système plus robuste `pivot_root()` * s'applique à l'intégralité du namespace **mount** * pour le processus * `/` est tout simplement monté sur un autre point de montage du système * le *vrai* `/` n'est pas monté dans son userspace

[6.1] ⋙ Comparatif des moteurs

	Docker	LXC	containerd	Podman
Type	Moteur complet	Conteneurs système	Runtime haut niveau	Moteur complet
Démon	`dockerd`	Aucun	`containerd`	Aucun
Rootless	Option	Oui (unprivileged)	Via nerdctl	Natif
Images OCI	Oui	Non (templates)	Oui	Oui
Runtime bas niveau	runc	liblxc	runc / crun	crun / runc

- **Docker** : écosystème, outillage, Compose
- **LXC** : « mini-OS » avec init, proche VM légère
- **containerd** : brique de Docker, backend Kubernetes (CRI)
- **Podman** : compatible CLI Docker, pods, intégration systemd



- **containerd**: maintenu par Docker
- **CRI-O**: maintenu par RedHat (utilisé par OpenShift)
- communication assez directe de **lxc**

Parmi les **runtimes bas niveau**, citons également:

- **youki**: écrit en Rust, performances natives
- **kata containers**: mini-VMs, isolation matérielle
- **gVisor (runsc)**: sandboxing, aucun **syscall** direct => passage par un **sentry**

Plusieurs `RuntimeClass` peuvent co-exister sur un même cluster kubernetes

Un conteneur ne fourni pas le même niveau d'isolation qu'une VM

- Noyau partagé $\models$ une faille noyau $\Rightarrow$ évasion (ex. Dirty Pipe)
- root du conteneur $\equiv$ root hôte (mod. user namespace)
- `--privileged` : quasi aucune isolation
- `docker.sock` monté (e.g. **traefik**) $\Rightarrow$ n'importe quel client HTTP dans le conteneur $\equiv$ root
- Ressources non namespacées : `/proc`, `/sys`, keyrings, modules
- CVE runtime : CVE-2019-5736, CVE-2024-21626

Menace	Parade
Évasion noyau	gVisor, Kata, microVM
Élévation de privilèges	Rootless, user namespaces
Capabilities excessives	`--cap-drop=ALL`, `no_new_privs`
Syscalls dangereux	Profil seccomp
Accès fichiers	AppArmor, SELinux
Images vulnérables	Scan, signature, images minimales (e.g. images distroless)

Conteneurisation Linux
≫Prise en main de docker

```
zypper install docker
systemctl start docker
systemctl enable docker
```

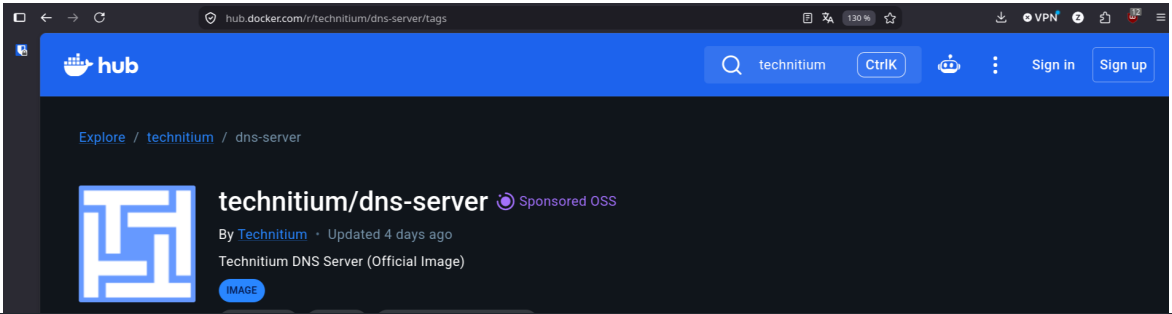
[6.2] ⋙ Instanciation d'un premier conteneur

Pour l'ensemble des démos, nous allons utiliser une image d'un serveur DNS du nom de technitium

Pour instancier un conteneur à partir de son image, il suffit de mentionner celle-ci

```
docker run technitium/dns-server:15.6.1 &
```

`docker` récupérera par défaut l'image depuis **dockerhub**



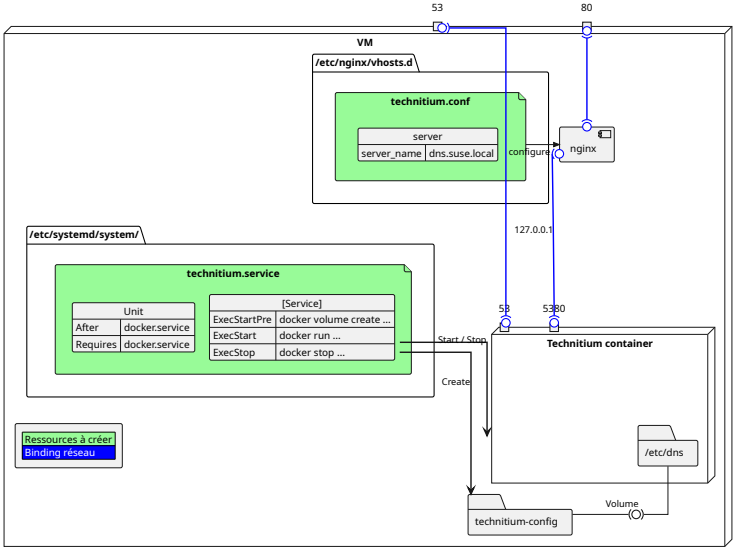
Si nous listons les conteneurs

```
docker container ls
CONTAINER ID   IMAGE    COMMAND   CREATED  STATUS  PORTS   NAMES
a18c5e9678ec   technitium/dns-server:15.6.1   "/usr/bin/dotnet /op..."   Abou
```

Nous pouvons voir celui que nous venons d'instancier et y entrer

```
docker exec -it a18c5e9678ec bash
```

[6.2] ⋙ Intégration de ce conteneur comme un service I



[6.2] ⋙ Intégration de ce conteneur comme un service II

Précisions

- **[reverse proxy]** Ajout d'un **vhost** ⇒ *dns.suse.local* pointe sur 127.0.0.1:5380. Après la création du fichier, il suffira de **redémarrer nginx**
- **[volume]**: **technitium-config** peut être créé à l'aide de `docker volume create`.
- `ExecStartPre` ≡ *exécuté avant*
- il sera nécessaire de **nommer explicitement le conteneur** afin de pouvoir le `docker stop` (option `--name`)
- il sera peut-être nécessaire de rendre certaines opérations **tolérantes à l'échec**

```
# la commande peut échouer sans que le service ne soit en échec
ExecStartPre=-/quelque/chose
# echec de la commande = échec du service
ExecStartPre=/quelque/chose
```