

# DevOps

Quelques outils utiles pour les développeurs et les sysadmins. Et donc principalement utiles aux DevOps.

- [📄 Git](#)
- [📄 Quelques recettes](#)
- [📄📄📄 Publier et collaborer](#)
- [📄 Gérer des univers parallèles](#)
- [📄 CI/CD avec gitlab](#)
- [📄 Exemple avec Pages](#)
- [📄 Fonctionnement avec SSH](#)
- [📄 Utiliser son propre Runner](#)
- [📄 Containers](#)
- [📄 Utilisation](#)
- [📄 Dockerizer et publier](#)
- [📄 Créer un oeuf pterodactyl](#)
- [📄 Utiliser Docker Compose](#)
- [📄 Réseau, reverse proxy avec Traefik](#)
- [📄 Visualiser tout avec Portainer](#)
- [📄 Archiver un site et des images](#)

# ? Git

Git est un logiciel très puissant et considéré par beaucoup comme un indispensable.

Le logiciel en lui même permet :

- D'avoir un historique de son code et ainsi pouvoir revenir dans un état précédent du projet;
- Gérer plusieurs versions du même code simultanément (avec les branches);
- Faire des backups de son code sur un/des serveur·s distants;
- Définir des releases dans l'historique (avec des tags)
- Collaborer avec d'autres sur le projet (avec les patches)

Mais quand on utilise une "git forge" (un site pour héberger le code tel que Gitlab, GitHub ou autre) on peut avoir encore d'autres avantages :

- Un moyen de partager facilement le code à tout le monde si on le souhaite
- Un "issue tracker" (c'est un genre de forum pour gérer le développement du projet)
- Un moyen pour les gens de faire des copies du code (appelées "fork"), de les modifier et d'ainsi pouvoir fusionner les deux projets en ajoutant les changements en un clic
- Un moyen d'avoir une page d'accueil du projet avec un README formaté
- Un wiki pour gérer la documentation du projet sans se prendre la tête
- Une vision claire du projet, de son historique, etc. Même pour celles et ceux qui ne connaissent pas Git.

Et avec certaines fonctionnalités de plus haut niveau de ces derniers :

- Un hébergement automatique de certain types de sites (avec Pages)
- Une automatisation des tests, build et/ou déploiement du projet en continu (voir chapitre suivant)

## Installation de Git

Sur Linux il suffit d'installer le paquet `git` disponible dans tous les gestionnaires de paquets et parfois même pré-installé dans certaines distributions.

Sous Windows ou macOS, il faut l'installer depuis le site: <https://git-scm.org>

Une fois cela fait, sur Linux ou macOS on peut ouvrir un terminal, et sur Windows on peut faire "clic droit" dans un dossier puis "ouvrir Git Bash".

Ensuite on va donner quelques informations à Git qui vont se retrouver dans l'historique de nos projets.

```
git config --global user.name "Votre nom"
git config --global user.email masuperadresse@mail.com
```

# Petit diagramme

Voici un petit diagramme pour vous aidez à vous retrouver dans les différents termes (HEAD, commit, branche, tag, remote, stash, index, etc)

Zoo diagram Gitlab

Branch diagram Gitlab

# ? Quelques recettes

OK, maintenant que l'on a installé et configuré Git, on peut maintenant passer à comment l'utiliser. On va commencer par les choses les plus communes dans Git.

## Créer un nouveau projet avec Git

Tout d'abord on va créer un nouveau projet Git en créant un nouveau dossier, et en l'initiant avec Git. (`mkdir` permet de créer un dossier, `cd` permet d'aller dedans et `git init` initialise un nouveau projet)

```
mkdir mon-super-projet
cd mon-super-projet
git init
```

Ensuite on va créer un nouveau fichier dans ce dossier, par exemple "README.md"

```
# La commande suivante écrit "Hello World" dans un nouveau fichier "README.md"
echo "Hello World" > README.md
```

Si on veut voir le status actuel de notre projet on peut utiliser les commandes `diff` et `status`

```
# La commande status liste les fichiers modifiés
git status

# La commande diff, liste les changements parmi les fichiers "suivi" (ajouté précédemment
dans l'historique)
git diff

# Cette dernière commande peut aussi servir pour avoir les différences entre plusieurs
fichiers ou plusieurs moments dans l'historique
```

Une fois que l'on a fait des changements on peut ajouter ces changements dans l'historique Git

```
# Git add ajouter le fichier dans la liste des changements qui vont être ajouté à l'historique
# On peut remplacer le nom du fichier par * (étoile) ou . (point) pour ajouter tous les
fichiers et dossiers
git add README.md

# Git commit l'ajoute à l'historique avec un message de description (obligatoire)
git commit -m "J'ai ajouté un nouveau fichier"
```

```
# On peut par après utiliser -am à la place de -m pour ajouter tous les fichiers qui ont déjà été commit
```

On peut ensuite voir que notre "commit" à été ajouté à l'historique du projet avec `log`

```
# Cette commande liste la liste des commits  
git log
```

## Définir un dossier existant comme projet Git

Tout d'abord on va aller dedans, soit en utilisant `cd` soit en faisant "clic droit" > "ouvrir Git bash ici". Ici je vais utiliser `cd`

```
cd /chemin/de/mon/dossier
```

Ensuite on peut initialiser Git dans le dossier

```
git init
```

Enfin on peut ajouter tout le dossier dans l'historique pour commencer

```
# le "." indique tout le dossier. Mais "*" aurait aussi pu fonctionner pour ajouter tous les  
fichiers et sous dossiers.  
git add .  
git commit -m "Premier commit"
```

## Utilisation de Git en temps normal

Maintenant imaginons vous faites vos modifications dans vos fichiers, etc. Et vous souhaitez "commit" les changements.

```
# en utilisant le flag -a, on ajoute automatiquement tous les fichiers "suivis"  
# C'est à dire, tous les fichiers qui ont déjà été ajouté dans le passé avec "git add"  
git commit -am "J'ai modifié des trucs"  
  
# Ou, si on veut tout envoyer d'un coup
```

```
git add .  
git commit -m "J'ai ajouté des trucs"  
  
# Ou encore si on veut faire fichier par fichier  
git add a.txt b.java  
git commit -m "J'ai ajouté des trucs"  
  
# Envoyer les changements sur le serveur une fois satisfait  
# on va voir cela plus en détails dans la section suivante  
git push origin master  
# Note: remplacer "master" par la branche concernée
```

# ???? Publier et collaborer

Maintenant on va voir comment publier son code sur une "git forge" tel que GitHub, Gitlab, Gitea, Codeberg, ou autre.

Tout d'abord il faut se créer un compte sur l'un de ces sites. Dans ce tutoriel on va utiliser Gitlab, car on va le réutiliser dans le chapitre suivant sur l'automatisation.

## Ajouter une clé SSH pour ne pas devoir entrer son mot de passe dans Git

On pourrait simplement donner l'URL de notre projet à Git et lui dire de publier le code. Le problème c'est qu'à chaque fois que l'on voudra faire ça, Git va redemander notre nom d'utilisateur et mot de passe.

Une méthode beaucoup plus sécurisée et beaucoup plus simple est d'utiliser des *clés SSH*.

Si ce n'est pas encore le cas, vous pouvez en créer en utilisant la commande suivante :

```
# Remplissez les champs demandés ou faites ENTER pour avoir les paramètres par défaut
# Faites attention à ne pas mettre de mot de passe en appuyant sur ENTER 2 fois.
ssh-keygen
```

Ensuite vous pouvez afficher votre "clé publique" (ce n'est pas confidentiel et c'est ce que Gitlab va utiliser pour vérifier votre identité)

```
# La commande "cat" lit un fichier. Ce fichier est notre clé publique
cat ~/.ssh/id_rsa.pub
```

Enfin on peut aller dans les paramètres de gitlab et ajouter cette clé.

Page d'ajout de la clé SSH sur Gitlab

## Publier un projet existant sur Git

Lien pour la création d'un repo sur Gitlab

Page de création d'un repo sur Gitlab

*On peut choisir si on veut que le repo soit privé, ou public. Il est aussi préférable de ne pas activer le "README" automatique*

Une fois que le "repo" est créé sur le site on peut soit simplement utiliser les commandes données, ou juste copier le lien SSH. Dans ce cas on va faire cela:

Copier le lien SSH

```
# On suppose que vous êtes déjà dans le projet, mais sinon ouvrez le terminal ou cd comme vu dans la section précédente
git remote add origin git@gitlab.com:julien.lejoly2712/mon-super-projet.git
```

Une fois le lien du "remote" ajouté. On peut maintenant "push" (publier) le code sur le site:

```
git push origin master
```

Et maintenant, sans avoir mis aucun mot de passe, mais en restant sécurisé, le code devrait être accessible sur la page du projet !

La page du projet maintenant

## Ajouter un collaborateur au projet

Maintenant si on veut ajouter un autre collaborateur sur le projet, on peut aller dans les informations du projet puis dans "Membres"

Screenshot du menu

Ensuite on peut cliquer sur "Inviter des membres" et inviter quelqu'un sous le rôle de "développeur".

Screenshot du formulaire

Maintenant le collaborateur peut télécharger le projet en utilisant la commande `git clone`

```
# L'adresse est l'adresse SSH prise plus tôt.
# Si on veut juste télécharger le code sans la possibilité de le modifier, on peut utiliser le lien HTTPS
git clone git@gitlab.com:julien.lejoly2712/course-book.git
cd course-book
```

A présent on peut utiliser Git comme vu précédemment. Mais la différence c'est qu'il faut de temps en temps aussi re-synchroniser le projet en utilisant `git pull`



# Collaborer sur un projet avec des "forks"

Si vous n'êtes pas un membre d'un projet mais souhaitez quand même contribuer, vous pouvez faire un "fork".

Un fork est une copie d'un projet. Vous pouvez ainsi y faire des modifications puis proposer d'ajouter (merge) les modifications dans le projet de base.

Dans cet exemple on va faire un fork du projet [Gitlab](#). Pour cela il suffit de cliquer sur le bouton "fork" en haut à droite.

Screenshot d'un fork de gitlab

Ensuite on peut copier le lien SSH vers notre fork :

Copier le lien vers le fork

Enfin, on peut télécharger le code et y faire nos changements.

```
git clone git@gitlab.com:julien.lejoly2712/gitlab.git
cd gitlab

# Pour l'exemple on va ajouter une ligne à la fin du fichier README.md
echo "Ceci est une ligne inutile" >> README.md

# On va ensuite ajouter les changements (le -a dans la commande commit va "add" tous les
fichiers suivi qui ont été modifiés)
git commit -am "J'ai ajouté une ligne inutile"

# On peut ensuite publier les changements sur notre fork
git push origin master
```

Une fois cela fait, on peut maintenant proposer d'ajouter nos changements sur le projet d'origine en créant un "merge request"



**ATTENTION:** Tout ceci est à titre d'**exemple**, le projet Gitlab est un vrai projet et les développeurs ont d'autre choses à faire que gérer des "merge requests" inutiles. **Merci de ne pas réellement faire ceci**

On peut donc créer une nouvelle "merge request" sur Gitlab.

Screenshot création merge request

Ensuite on précise quel "branche" (on va voir en détails les branches après), et quel projets on va fusionner. Dans ce cas on précise que `gitlab-org/gitlab` est la destination de notre proposition de fusion.

Screenshot merge request

Enfin on peut ajouter une description de nos changements pour finir notre merge request.

Screenshot du formulaire

A présent, les membres du projet de base peuvent ajouter tous vos changements en un clic, si tout se passe bien (on va voir les conflits Git plus tard).

Screenshot pov membre

# ? Gérer des univers parallèles

Maintenant on va voir un autre côté de Git qui est la possibilité de gérer plusieurs version du code simultanément avec des "branches".

## Créer une nouvelle branche

Imaginons que nous voulons tester de nouvelles fonctionnalités sans impacter la branche principale. Pour cela on va créer une nouvelle branche et aller dessus:

```
# "checkout" est utilisé pour voyager entre les différentes branches
# Le flag "-b" est utilisée pour créer une nouvelle branche
git checkout -b ma-super-fonctionalite
```

Maintenant on peut aussi voir la liste des branches et voir sur quelle branche nous sommes actuellement

```
git branch
```

A partir de maintenant tout commit fait ne sera pas enregistré sur la branche principale. Si on veut retourner sur la branche principale on peut alors utiliser un simple checkout

```
git checkout master

# Et puis quand on veut y retourner
git checkout ma-super-fonctionalite
```

## Voyager entre les branches

Vous êtes en train de faire votre projet, mais vous voulez aller voir ou rapidement modifier un truc sur la branche principale. Problème : vous n'avez pas encore fait de commit, donc vous ne pouvez pas changer de branche sinon vous allez perdre vos changements.

Git heureusement nous protège de ce scénario et affichera un message d'erreur si on souhaite changer de branche sans commit les changements.

Mais il y a une autre commande que l'on peut utiliser pour foutre tous les changements en cours dans un coin le temps de voyager entre les branches. C'est `stash`

```
# Cette commande va mettre temporairement vos changements dans un coin pour les retrouver plus tard
git stash

# Vous pouvez maintenant changer de branche sans problème
git checkout master
```

Maintenant vous avez fini vos explorations et voulez retourner à votre nouvelle branche et récupérer vos changements:

```
git checkout ma-super-fonctionalite

# "stash pop" pour retrouver les changements qui avaient été mis de coté avec stash
git stash pop
```

Si à l'inverse on souhaite ne pas les retrouver et les supprimer on peut utiliser "stash drop"

```
git stash drop
```

## Fusionner des branches

Maintenant si on imagine que l'on a fait nos changements sur notre nouvelle branche et que l'on veut ajouter ces changements à la branche principale. On va utiliser `merge`

```
# Changement vers la branche principale
git checkout master
git merge ma-super-fonctionalite
```

Et en théorie ça fonctionne ☐☐

Mais dans certains cas, quand les deux branches ont des modifications différentes pour un même fichier cela peut créer des conflits.

## ? Merge conflict

Pour gérer les conflits il y a plusieurs méthodes.

# ? CI/CD avec gitlab

Dans le chapitre précédent nous avons vu comment utiliser Git pour gérer l'historique de fichiers, s'organiser, collaborer avec d'autres développeurs, etc.

Maintenant nous allons voir un autre aspect très intéressant de Git qui est, l'automatisation, ou *Continuous Integration / Continuous Deployment* qui permet qu'à chaque push sur une branche donnée, un script s'exécute pour tester, build et deployer le projet automatiquement.

Cela est donc un très grand gain de temps et ce n'est pas très compliqué. Pour cela il faut déjà maîtriser Git et se créer un compte sur Gitlab.com.

# ? Exemple avec Pages

Tout d'abord on va créer un nouveau repo sur Gitlab. Puis on va l'ajouter dans un repo local.

```
git init gitlab-pages-test
cd gitlab-pages-test
git remote add origin <lien ssh ou https ici>
```

Ensuite on va créer un nouveau fichier `index.html` dans lequel il est simplement écrit "Hello, World!"

```
echo "Hello, World!" > index.html
```

Maintenant que l'on a créé notre contenu, on va créer le fichier principal `.gitlab-ci.yml` qui va contenir les instructions pour créer notre projet. Ce fichier constitue ce qui s'appelle dans GitLab, une "pipeline"

```
# "pages" est le nom du "job"
pages:
  # Stage indique quel type d'action qui est effectuée (par exemple: test, build, deploy)
  stage: deploy

  # L'image est la base du système dans lequel les commandes d'installation du projet vont
  être lancées. Dans ce cas ci, debian
  # Nous verrons plus en détail cela dans le chapitre sur Docker
  image: debian

  # Les artifacts sont des fichiers ou dossier qui vont être exporté. Dans cet exemple
  `public` va être exportée en dehors de notre "pipeline"
  artifacts:
    paths:
      - public

  # Le "before_script" spécifie les commandes d'installation de l'environnement, par exemple
  ici nous allons utiliser rsync pour déployer notre projet. Donc j'installe rsync
  # Il faut toujours faire en sorte que les commandes ne nécessite pas d'interaction
  (exemple, en ajoutant -y à la commande APT)
  before_script:
    - apt-get update
```

```
- apt-get install -y rsync
```

# Le "script" est la partie principale. Elle indique les commandes à lancer pour déployer notre projet.

# Ici on ne fait que créer un nouveau dossier public et copier tous les fichiers du projets (à l'exception de "public/") dans le dossier "public/"

script:

```
- mkdir public  
- rsync -rv * public/ --exclude=public/
```

# On précise que seul les pushes vers la branche main peuvent appeler le job "pages".

only:

```
- main
```

Une fois nos deux fichiers créés on peut maintenant envoyer le tout sur le serveur

```
git add index.html .gitlab-ci.yml  
git commit -m "Test de CI/CD sur gitlab"  
git push origin main
```

Maintenant on peut aller dans l'onglet "CI/CD" de Gitlab, puis dans "Jobs", puis sur "Running" pour voir les détails de ce qu'il se passe.

Si tout se passe bien après quelques secondes il devrait être écrit "Passed" et le site devrait être accessible. Le lien est trouvable dans l'onglet paramètre du repo, puis dans "Pages".

## Plus d'info

- [Liste de tous les keywords pour .gitlab-ci.yml](#)
- [Gitlab CI/CD Quick Start](#)

# ? Fonctionnement avec SSH

Maintenant si on reprends notre projet précédent, et que l'on veut non pas le déployer avec GitLab Pages, mais sur un serveur personnel.

## Configurer SSH

Pour cela on peut accéder à ce serveur via SSH. Donc à la place de donner un mot de passe à GitLab, on va générer une clé SSH (comme vu dans le chapitre sur Linux)

```
ssh-keygen -f ~/.ssh/id_gitlabci
```

Ensuite on peut envoyer cette nouvelle clé (publique) sur le serveur :

```
ssh-copy-id -i ~/.ssh/id_gitlabci
```

Enfin on peut se connecter pour voir si tout fonctionne

```
ssh -i ~/.ssh/id_gitlabci <utilisateur>@<hôte>
```

En se connectant on a aussi créé le fichier `known_hosts` que l'on va utiliser plus tard.

## Configurer les variables de GitLab-CI

Tout d'abord il faut que l'on récupères quelques informations de notre configuration SSH (la clé privée, et le fichier `known_hosts`)

```
cat ~/.ssh/id_gitlabci
cat ~/.ssh/known_hosts
```

Maintenant on peut aller sur notre projet GitLab, dans les paramètres CI/CD, puis dans la section "Variables".

Ici on va ajouter 2 variables :

Nom de la variable	Contenu	Protégée ?
SSH_PRIVATE_KEY	La clé privée ( <code>cat ~/.ssh/id_gitlabci</code> )	Oui



Nom de la variable	Contenu	Protégée ?
SSH_KNOWN_HOSTS	Le fichier known_hosts ( <code>cat ~/.ssh/known_hosts</code> )	Oui

# Le fichier .gitlab-ci.yml

Maintenant on peut se repencher sur le fichier que l'on a créé un peu plus tot. Dans celui ci on va ajouter ceci dans la section `before_script` :

```
# "pages" est le nom du "job"
prod:

  # Stage indique quel type d'action qui est effectuée (par exemple: test, build, deploy)
  stage: deploy

  # L'image est la base du système dans lequel les commandes d'installation du projet vont
  être lancées. Dans ce cas ci, debian

  # Nous verrons plus en détail cela dans le chapitre sur Docker
  image: debian

  # Plus besoin d'artefacts car on en a pas besoin et que l'on utilise pas GitLab Pages

  # Le "before_script" spécifie les commandes d'installation de l'environnement, par exemple
  ici nous allons utiliser rsync pour déployer notre projet. Donc j'installe rsync
  # Il faut toujours faire en sorte que les commandes ne nécessite pas d'interaction
  (exemple, en ajoutant -y à la commande APT)
  before_script:
    - apt-get update
    - apt-get install -y rsync

  # On va installer ssh-eval et le lancer dans l'environnement de build
  - apt-get install openssh-client -y
  eval $(ssh-agent -s)

  # On va ensuite ajouter notre clé privée
  - echo "$SSH_PRIVATE_KEY" | tr -d '\r' | ssh-add -

  # Et on va mettre les bonnes permissions pour le dossier .ssh
  - mkdir -p ~/.ssh
  - chmod 700 ~/.ssh

  # Enfin on va ajouter le fichier known_hosts pour ne pas créer de problème lors de la
  vérification automatique des clés
```

```
- echo "$SSH_KNOWN_HOSTS" >> ~/.ssh/known_hosts  
- chmod 644 ~/.ssh/known_hosts
```

# Le "script" est la partie principale. Elle indique les commandes à lancer pour déployer notre projet.

# Ici on va envoyer tous les fichiers vers un répertoire d'un serveur distant grâce à rsync et SSH

script:

```
- rsync -rv * <utilisateur>@<hôte>:/<chemin vers le dossier cible>
```

# On précise que seul les pushes vers la branche main peuvent appeler le job "prod"

only:

```
- main
```

Maintenant on peut envoyer le tout au serveur et déclencher la pipeline que l'on vient de modifier.

```
git add .gitlab-ci.yml  
git commit -m "Changement de Pages à un serveur distant"  
git push origin main
```

Et normalement, tout devrait fonctionner !

# ? Utiliser son propre Runner

Toutes les instances n'ont pas des runner partagés. Donc dans certains cas vous allez peut-être avoir besoin de créer votre runner vous-même.

Cela peut aussi être fait quand on veut que le runner tourne dans un certain réseau par exemple (pour avoir accès à certains serveurs, etc)

## Installation

Tout d'abord il faut installer docker et gitlab-runner. Vous pouvez trouver les instructions sur leurs sites respectifs. Mais voici les instructions pour Arch Linux

```
yay -S gitlab-runner docker
```

Ensuite on peut enregistrer un nouveau runner

```
sudo gitlab-runner register
```

Ici, il faut aller dans les paramètres du projet sur GitLab, dans CI/CD, dans la section "Runners" et voir dans les instructions pour les runners spécifiques. Donc:

- GitLab instance URL = voir dans les paramètres
- Registration token = voir dans les paramètres
- Description = N/A
- Tags = N/A
- Maintenance note = N/A
- Executor = docker

Une fois créé on peut le lancer en faisant

```
sudo gitlab-runner start
```

On peut aussi utiliser la commande suivante pour avoir plus de détails

```
sudo gitlab-runner --debug run
```

Et voilà, c'est configuré. Si on installe cela sur un serveur, le runner pourra être disponible 24h/24

# ? Containers

Docker est un logiciel qui est pas mal utilisé pour installer des services sur un serveur.

L'avantage c'est que Docker va virtualiser les services dans des genres de "mini-machine virtuelles". Ce qui permet :

- D'avoir un environnement isolé et constant. Permettant de eployer quelque chose quel que soit l'état de la machine hôte.
- Deployer des services rapidement et facilement
- Permet une plus grande extensibilité
- Plus sécurisé grâce à l'isolation
- Il est aussi possible de revenir dans des états précédents d'un pod

C'est nottament ce qui est utilisé quand on a utilisé Gitlab-CI pour l'automatisation.

Et c'est ce que l'on va de nouveau utiliser quand on va utiliser Kubernetes pour avoir une plus grande extensibilité.

## Pourquoi j'aime Docker

Il y a beaucoup d'avantages que je trouve à Docker.

- La capacité de compiler un programme sans être parasité par d'autres choses installées sur l'ordi (exemple versions plus récentes de certaines librairies)
- La capacité d'avoir une vue sur toute l'infrastructure (environnement, ports, domaines, images, dossiers de configuration, commandes d'installation, versions, lien avec d'autres services, etc)
- La capacité de tester des choses sans avoir à se soucier d'annuler les changements ou de brick son système.
- Pouvoir créer des images qui fonctionne sur n'importe quel serveur tournant n'importe quel OS avec n'importe quel infrastructure (portabilité)
- Pouvoir facilement orchestrer les différents services à l'aide d'un simple fichier. Et même pouvoir gérer plusieurs serveurs en même temps.
- Pouvoir faire facilement des backups de tout, en ayant tout dans une image sans dépendre de dépendences etc

## Quelques concepts à comprendre

Un **volume** est un moyen de rendre certaines données persistantes. Car quand vous re-créez un conteneur, tout le reste sera réinitialisé. Le volume lie un dossier sur la machine hôte à un dossier dans le conteneur. Et il y a deux types de volumes:

- Soit le volume est défini et géré par Docker (et se trouve donc dans `/var/lib/docker/volumes`) ou dans quel cas le contenu initial qui était dans le dossier sur le conteneur prends le dessus
- Soit le volume est un *bind*, c'est à dire un lien direct à un dossier de la machine cible. Dans quel cas le contenu initial sur la machine hôte prends le dessus. Donc si le dossier/fichiers sur la machine hôte n'existe pas, ils se feront aussi supprimer au moment du lancement du conteneur.

Pour utiliser un bind et quand même faire en sorte de garder les données du côté conteneur, il vaut mieux alors définir la mise en place des données dans le dossier au moment du `entrypoint.sh` (on va voir ça quand on vera la dockerization)

Un **réseau** est un lien entre plusieurs conteneurs qui leur permet de communiquer sans avoir besoin d'exposer les ports de ces derniers. On va en savoir plus sur le chapitre sur `traefik` (reverse-proxy)

Une **conteneur** est ce qui tourne sur Docker, un conteneur peut être lié à un volume, à des réseaux, etc. Un conteneur fait tourner une **image** qui aura été préalablement créée.

Une **image** est en quelque sorte l'état "de base" d'un conteneur. Elle contient toutes les informations initiales nécessaires, le script qui a permis de la configurer, la commande de lancement du conteneur et tous les fichiers qui y ont été ajouté. C'est une genre d'"archive" de machine virtuelle si on veut.

Un **registry** est un répertoire pour y envoyer ou télécharger des images. Le plus connus de tous est le [Docker Hub](#)

# Installation

Pour l'installer vous pouvez suivre les [instructions suivantes](#) (évitez Docker Desktop si possible)

# ? Utilisation

Ici on va parler de comment utiliser Docker *en ligne de commandes*, on parlera plus tard de systèmes tel que docker-compose, Kubernetes, Pterodactyl, Portainer ou autre.

## ?? Lancer un conteneur

Pour lancer un conteneur on va utiliser la commande docker `run`, voici un exemple :

```
sudo docker run --name mon-super-docker -e HELLO=world -v ./mon-volume:/home -p 8080:80 -it
docker.io/library/debian
```

Ici dans cet exemple, on a plusieurs options à notre commande :

- `--name` va nomer le conteneur
- `-e HELLO=world` va définir la variable d'environnement "HELLO" pour valoir "world"
- `-v ./mon-volume:/home` va dire que le contenu du dossier local "mon-volume" remplacer le dossier du conteneur "/home" (les dossiers/fichiers hôtes dans un volume passe toujours avant)
- `-p 8080:80` va lier le port `8080` de la machine hôte au port `80` du conteneur.
- `-it` va allouer un `tty` à la machine et l'afficher de manière interactive. On peut donc lancer des commandes dans le conteneur.
- `docker.io/library/debian` défini que l'on va lancer l'image officielle de debian depuis les serveurs de Docker Hub

## ?? Lister les conteneur et les images (ps, images)

Voici quelques commandes pour afficher des informations sur les conteneurs :

```
# Va afficher la liste des conteneurs en cours
sudo docker ps

# Va afficher la liste de tous les conteneurs
sudo docker ps -a
```

```
# Va afficher la liste des images docker  
sudo docker images
```

## ? Gérer des images et des conteneurs (start, stop, rm)

```
# Arrêter un conteneur  
sudo docker stop CONTAINER_ID  
  
# Relancer un conteneur  
sudo docker start CONTAINER_ID  
  
# Supprimer un conteneur (nécessite qu'il soit préalablement arrêté)  
sudo docker rm CONTAINER_ID  
  
# Supprimer une image (nécessite qu'aucun conteneur n'en dépende)  
sudo docker image rm IMAGE_ID  
  
# Note: CONTAINER_ID peut aussi être le nom du conteneur
```

## Créer une nouvelle image et la mettre en ligne (build et push)

On va voir en détail cette partie dans la prochaine partie.

## ? Exécuter des commandes dans un conteneur en cours (exec)

Si on souhaite *inspecter* un conteneur on peut utiliser la commande `exec`

```
# Va entrer dans une shell bash du conteneur "mon-conteneur"  
sudo docker exec -it mon-conteneur /bin/sh  
  
# Va exécuter la commande "ls" dans mon-conteneur
```

# ? Compiler dans un environnement propre avec Docker (exemple)

Si vous souhaitez compiler votre programme, imaginons en Rust, mais que vous souhaitez le faire dans un environnement neuf pour empêcher des incompatibilité pour les futurs utilisateur·ice·s. Vous pouvez utiliser Docker pour ça !

```
docker run --rm -u "$(id -u)": "$(id -g)" -v "$PWD":/usr/src/myapp -w /usr/src/myapp  
rust:bullseye cargo build --release
```

Pour décortiquer un peu la commande:

- `--rm` va automatiquement supprimer le conteneur une fois la compilation terminée
- `-u "$(id -u)": "$(id -g)"` définit l'utilisateur à utiliser dans le conteneur à être celui de la machine hôte (pour les permissions de fichiers)
- `-v "$PWD":/usr/src/myapp` va lier le dossier actuel au dossier `/usr/src/myapp` dans le conteneur
- `-w /usr/src/myapp` va définir ce dossier comme étant le dossier actuel
- `rust:bullseye` est l'image utilisée (c'est debian + les outils de développement de Rust)
- `cargo build --release` est la commande de compilation à lancer dans le conteneur



# ? Dockerizer et publier

Dans ce petit exemple j'ai voulu dockerizer une image de Debian avec `openjdk-18-jdk`.

Généralement quand on dockerize une quelque chose c'est très équivalent à l'installer en *bare* sur un serveur donc connaître Linux est un prérequis pour pouvoir faire ceci. Mais il y a quelques difficultés en plus quand on le crée pour Docker.

- Il n'y a pas de système de init, donc pas de systemd. Par exemple pour utiliser php-fpm et nginx, il faut manuellement les installer, les configurer et les démarrer dans le entryptpoint. Alors que normalement on a pas besoin de ce soucier de ça
- Il est bien de pouvoir connaître alpine Linux car c'est une distribution light très souvent utilisée pour créer des images Docker pour des raisons d'optimisation. C'est différent notamment car beaucoup de fichiers binaires Linux ne sont pas compatibles avec cette distribution et que les configurations de logiciels commun tel que php-fpm ou nginx peuvent être différentes sur celle ci
- On peut optimiser encore plus mais cela nécessite d'avoir de très bonnes connaissances du système que l'on utilise pour savoir ce qui peut être supprimé

## ? Tester des choses dans un pod

On sais que l'on veut partir d'une image debian. Mais on ne sais peut-être pas les instructions exacte pour y installer openjdk-18-jdk. Donc on peut créer un pod Debian et l'utiliser comme environnement de test.

```
docker run --rm -it docker.io/library/debian
```

Maintenant, on a une distribution Debian pour y faire nos tests.

## ? Créer le dockerfile

Pour cela on va d'abord créer un dossier vide pour notre projet

```
mkdir docker-openjdk
cd docker-openjdk
```

Maintenant on peut créer un fichier `Dockerfile` et y ajouter ceci:

```
# On part d'une image de debian
FROM debian

# On ajoute le répertoire SID de debian, met a jour et installe le paquet openjdk-18-jdk
RUN echo "deb http://deb.debian.org/debian sid main contrib non-free" >> /etc/apt/sources.list
RUN apt-get update
# On fait également attention a ce que aucune commande ne nécessite une interaction de
l'utilisateur avec "-y"
RUN apt-get install -y openjdk-18-jdk

# On indique la commande principale qui va être lancée quand on crée un pod avec l'image
CMD [ "/bin/bash" ]
```

Une fois cela fait on peut build l'image, on va mettre comme nom `docker.io/<username>/debian-openjdk-18` car c'est l'adresse vers laquelle on va publier l'image plus tard. Et on va aussi utiliser `--no-cache` dans les cas où le build dépend de ressources externes car sinon Docker ne les reprendra pas.

```
sudo docker build . --no-cache -t docker.io/<username>/debian-openjdk-18
```

## ? Publier dans un registry

Maintenant, on peut aller sur [Docker Hub](https://hub.docker.com/), créer un nouveau repo. Puis ensuite on va se connecter via docker et publier le tout

```
sudo docker login docker.io
sudo docker push docker.io/<username>/debian-openjdk-18
```

Maintenant l'image a bien été créée à partir de nos instructions et a été publiée sur le registry de Docker Hub. On peut maintenant tester si l'image fonctionne correctement en lançant un nouveau pod.

```
docker run --name debian-openjdk-18-test -it docker.io/<username>/debian-openjdk-18
```

Nous pouvons maintenant par exemple l'utiliser dans un GitLab-CI ou encore un docker-compose par exemple.

```
image: <username>/debian-openjdk-18
```

# ? Créer un oeuf pterodactyl

Pterodactyl utilise des docker pour exécuter les services mais a besoin de quelques configuration supplémentaire pour fonctionner correctement avec les fichiers et la commande STARTUP.

Attention, pterodactyl nécessite que les dockers soient le plus petit possible. Aussi pour faire des fichiers binaires, il est judicieux de les compiler eux même dans un docker comme vu dans la section sur docker.

## ? Creation du Docker

⚠ Note: /home/container est un volume, donc les fichiers hôte sont privilégiés et écraseront ceux placé par le Dockerfile. C'est pour cela qu'il vaut mieux les stocker ailleurs et les redéplacer dans le entrypoint.sh

Pour créer le Docker on peut créer un nouveau fichier `Dockerfile`

```
FROM docker.io/library/debian:bullseye

# CHANGER ICI
LABEL author="nom du developpeur" maintainer="adresse@mail.com"

# Commandes pour setup le bot (RUN pour lancer des commandes et ADD pour transfer des fichiers locaux)
# CHANGER ICI
ADD aurore /aurore
RUN chmod +x /aurore

# Pour déplacer un fichier dans /home/container
ADD nginx.conf /nginx.conf

# Création de l'utilisateur container et des variables obligatoires pour ptero
RUN apt update && apt install -y wget
RUN useradd -d /home/container -m container
USER container
ENV USER=container HOME=/home/container
```

```
WORKDIR /home/container
ADD ./entrypoint.sh /entrypoint.sh
CMD ["/bin/bash", "/entrypoint.sh"]
```

Et pour le `entrypoint.sh` qui va exécuter la commande d'install (rien à modifier)

```
#!/bin/bash
cd /home/container

# Placer ici des fichiers pour les mettre dans /home/container
ls /home/container/nginx.conf || cp /nginx.conf /home/container/nginx.conf # Va déplacer le
fichier nginx.conf si il n'existe pas encore

# Replace Startup Variables
MODIFIED_STARTUP=`eval echo $(echo ${STARTUP} | sed -e 's/{{/${/g' -e 's/}}/}/g')`
echo ":/home/container$ ${MODIFIED_STARTUP}"

# Run the Server
eval ${MODIFIED_STARTUP}
```

Ensuite on peut lancer le docker et publier.

```
sudo docker build . -t <username>/<repo>
sudo docker login
sudo docker push <username>/<login>
```

## ? Création d'un oeuf Pterodactyl

1. Créer une nest dans le panel admin
2. Création d'un oeuf dans une nest dans lequel on va préciser :
  - Le nom, la description, l'auteur·ice
  - L'image (`docker.io/<username>/<repo>`)
  - La commande de démarrage (relative au dossier `/home/container`)
  - La stop command (`^C` à mettre par défaut)
  - Les autre configuration, juste écrire `{}` dans chaque par défaut
3. Dans la section variable, ajouter les variables d'environnement nécessaire
4. L'oeuf est prêt et par conséquent on peut l'exporter en JSON ou créer un serveur utilisant l'oeuf.

# ? Utiliser Docker Compose

```
version: '3'

# On peut définir plusieurs conteneurs d'un seul coup en utilisant docker-compose
# Et si on modifie ce fichier, cela va supprimer les anciens docker pour les mettre à jour
services:
  db:
    # On définit l'image
    # Sinon on peut aussi utiliser build: <chemin> pour demander de build l'image par soi même
    image: mariadb:10.6.4-focal

    # On peut définir les arguments de la commande CMD
    command: '--default-authentication-plugin=mysql_native_password'

    # On peut définir des volumes, soit par bind : <chemin sur l'hôte>:<chemin sur le docker>
    #                               soit par nom : <nom du volume>:<chemin sur le docker> où
    # dans quel cas cela sera géré par Docker
    # Ici on va lier /var/lib/mysql au volume du nom de "db_data"
    volumes:
      - db_data:/var/lib/mysql

    # On peut définir une condition pour que le conteneur redemarre automatiquement (par
    # défaut: no)
    restart: always

    # On peut définir des variable d'environnement
    # Quand il y en a vraiment beaucoup et que l'on ne veut pas les mettre ici on peut les
    # mettre dans un fichier à part et utiliser "env_file" à la place pour les importer (voir la
    # doc pour plus d'info)
    environment:
      - MYSQL_ROOT_PASSWORD=somewordpress
      - MYSQL_DATABASE=wordpress
      - MYSQL_USER=wordpress
      - MYSQL_PASSWORD=wordpress

    # Ou encore des ports
```

```

# Les ports dans docker suivent la syntaxe HOST_PORT:CONTAINER_PORT
ports:
  - "3306:3306"
  - "33060:33060"

# Pareil ici mais avec un autre service du nom de "wordpress"
wordpress:
  image: wordpress:latest
  volumes:
    - wp_data:/var/www/html
  ports:
    - 80:80
  restart: always
  environment:
    - WORDPRESS_DB_HOST=db
    - WORDPRESS_DB_USER=wordpress
    - WORDPRESS_DB_PASSWORD=wordpress
    - WORDPRESS_DB_NAME=wordpress

# Enfin ici on peut définir les noms des volumes (seulement dans le cas par nom, pas
nécessaire dans les cas de bind)
volumes:
  db_data:
  wp_data:

```

Un docker compose comme celui ci, permet d'automatiquement déployer plusieurs conteneurs/services sur *une* machine très simplement. C'est très pratique pour expérimenté (si le fichier est mis à jour puis redémarré, cela va automatiquement remettre les conteneurs à jours)

Il suffit juste d'être dans un dossier où un fichier du nom de `docker-compose.yml` existe et de lancer la commande :

```

# Pour le lancer en mode debug :
sudo docker-compose up

# Pour le lancer pour de bon
sudo docker-compose up -d

# Pour le stopper
sudo docker-compose down

```

Ainsi maintenant on peut faire des configurations plus complexes beaucoup plus simplement qu'avec les lignes de commandes.

Pour en savoir plus sur la syntaxe des fichiers Docker Compose (version 3) vous pouvez cliquer [ici](#)

# ? Réseau, reverse proxy avec Traefik

Maintenant, quand on veut pouvoir avoir plusieurs services sur un même port avec des noms de domaines différents. On doit utiliser un reverse-proxy, on peut par exemple utiliser `nginx` mais ce n'est pas pratique car cela sort de l'écosystème Docker.

Donc à la place on va utiliser `traefik`. Et on va faire ces configurations dans des docker-compose par soucis de simplicité et de lisibilité.

De plus traefik a l'avantage d'automatiquement renouveler les certificats HTTPS.

## ? HTTP (non sécurisé)

```
version: '3'

services:
  # On crée le service traefik
  traefik:
    image: "traefik:latest"

    # On spécifie certains paramètres dans la commande
    command:
      # On définit que c'est docker que l'on utilise
      - "--providers.docker=true"
      # On dit à traefik de ne router des conteneurs que quand on dit qu'il faut le faire
      - "--providers.docker.exposedbydefault=false"
      # Et on définit un entrypoint (là où les utilisateurs vont se connecter) au port 80
      # appelé "web"
      - "--entrypoints.web.address=:80"

    # On expose le port 80
    ports:
      - "80:80"

    # Et on donne accès à traefik à docker en lecture seule
```



volumes:

- "/var/run/docker.sock:/var/run/docker.sock:ro"

# Ici c'est un simple service de test

whoami:

image: "traefik/whoami"

# Les labels permettent de mettre des "étiquettes" sur les conteneurs, c'est ceux ci que traefik recherche

labels:

# On dit à traefik de s'occuper de ce conteneur

- "traefik.enable=true"

# On dit à traefik de router ceci vers "mon-nom-de-domaine.net"

- "traefik.http.routers.whoami.rule=Host(`mon-nom-de-domaine.net`)"

# Et on lui dit aussi d'utiliser l'entrypoint web défini plus tot

- "traefik.http.routers.whoami.entrypoints=web"

Maintenant on peut faire `sudo docker-compose up` et aller sur `http://mon-nom-de-domaine.net` pour voir le site.

## ? HTTPS (sécurisé)

version: '3'

services:

traefik:

image: "traefik:latest"

command:

- "--api.insecure=true"
- "--providers.docker=true"
- "--providers.docker.exposedbydefault=false"
- "--entrypoints.web.address=:80"

# On ajoute un nouveau entrypoint sur le port 443 appelé "websecure"

- "--entrypoints.websecure.address=:443"

# On dit que l'on va ajouter un resolver de certificat en utilisant les challenge http appelé "myresolver"

# Par défaut on va utiliser letsencrypt

- "--certificatesresolvers.myresolver.acme.httpchallenge=true"

# On lui donne comme entrypoint celui qui est sur le port 80

```

- "--certificatesresolvers.myresolver.acme.httpchallenge.entrypoint=web"

# On expose les ports 80 et 443
ports:
  - "80:80"
  - "443:443"
volumes:
  - "/var/run/docker.sock:/var/run/docker.sock"
whoami:
  image: "traefik/whoami"
  labels:
    - "traefik.enable=true"
    - "traefik.http.routers.whoami.rule=Host(`mon-nom-de-domaine.net`)"
  # On change le entrypoint pour être celui de 443 (websecure)
  - "traefik.http.routers.whoami.entrypoints=websecure"
  # Et on défini le resolver comme étant "myresolver"
  - "traefik.http.routers.whoami.tls.certresolver=myresolver"

```

Pareil ici en faisant `sudo docker-compose up` on va maintenant pouvoir aller sur `https://mon-nom-de-domaine.net` et y voir le site.

## ? Comment utiliser traefik pour plusieurs docker-compose différents ?

Quelque chose que je n'ai pas précisé c'est que quand on utilise docker-compose, tous les services de celui ci sont dans le même "réseau" docker, qui est isolé du reste.

Seulement on ne peut avoir qu'une seule fois traefik car les ports ne peuvent être bind qu'une seule fois. Mais traefik a aussi besoin que tous les services au quel il doit accéder soit dans le même réseau que lui.

Donc pour utiliser plusieurs docker-compose avec un seul traefik, on va avoir besoin de définir un réseau.

```

# Ici on crée un réseau appelé "traefik"
sudo docker network create traefik

```

Maintenant on peut définir ce réseau (externe car défini en avance) dans nos docker-compose

Premier (pour traefik) :

```

version: '3'

services:
  traefik:
    image: "traefik:latest"
    command:
      - "--api.insecure=true"
      - "--providers.docker=true"
      - "--providers.docker.exposedbydefault=false"
      - "--entrypoints.web.address=:80"

      - "--entrypoints.websecure.address=:443"
      - "--certificatesresolvers.myresolver.acme.httpchallenge=true"
      - "--certificatesresolvers.myresolver.acme.httpchallenge.entrypoint=web"

    ports:
      - "80:80"
      - "443:443"
    volumes:
      - "/var/run/docker.sock:/var/run/docker.sock"

    # On lie ce service au réseau "traefik"
    networks:
      - traefik

# On précise que le réseau traefik est externe et non pas interne au docker-compose
networks:
  traefik:
    external: true

```

Et le deuxième pour "whoami"

```

version: '3'

services:
  whoami:
    image: "traefik/whoami"
    labels:
      - "traefik.enable=true"
      - "traefik.http.routers.whoami.rule=Host(`mon-nom-de-domaine.net`)"

```

- ```
- "traefik.http.routers.whoami.entrypoints=websecure"
- "traefik.http.routers.whoami.tls.certresolver=myresolver"
```

# On lie ce service au réseau "traefik"

```
networks:
```

- ```
- traefik
```

# On précise que le réseau "traefik" est externe et non pas interne au docker-compose

```
networks:
```

```
traefik:
```

```
external: true
```

Maintenant on peut lancer les deux docker-compose et tout devrait tout de même fonctionner comme avant ☐☐

💡 Note: Par défaut docker-compose crée un réseau pour les services qui y sont, mais quand vous définissez manuellement un réseau comme ici, ce n'est pas le cas. Donc si vous souhaitez que les services communiquent entre eux, mettez les également dans le réseau `traefik`.

## ? Petits ajouts

- Comment faire pour qu'un port 9000 d'un service devienne le port 80 par exemple ? Pour faire de la redirection de ports on peut utiliser ceci

```
- "traefik.http.services.NOMDUSERVICE.loadbalancer.server.port=9000" # On défini le port du service comme étant 9000
- "traefik.http.routers.NOMDUSERVICE.entrypoints=NOMDELENTPOINT" # On le lie à l'entrypoint de :80
```

- Comment lier un service à un hôte tel que `http://mon-nom-de-domaine.net/hello` (avec un préfixe) :

```
- "traefik.http.routers.NOMDUSERVICE.rule=(Host(`mon-nom-de-domaine.net`) && PathPrefix(`/hello`))"
```

- Comment faire une redirection du http vers https (http étant le port 80 donc l'entrypoint "web" et https étant le port 443 et donc l'entrypoint "websecure")

- "--entrypoints.web.http.redirections.entrypoint.to=websecure"
- "--entrypoints.web.http.redirections.entrypoint.scheme=https"

# ? Visualiser tout avec Portainer

Portainer est un super outil pour avoir une interface web pour gérer tout à propos de Docker : images, stacks (docker-compose), conteneurs, volumes, réseaux, etc.

On va utiliser traefik comme reverse proxy avec portainer. Sauf que si on met les deux dans le même fichier docker-file, on ne pourra plus gérer traefik avec portainer car il aura été créé en dehors.

On va d'abord créer un réseau traefik pour pouvoir ajouter traefik par après :

```
sudo docker network create traefik
```

Donc on va mettre traefik après. Tout d'abord on va créer portainer en version non sécurisée :

```
version: '3'
services:
  portainer:
    image: portainer/portainer-ce:latest
    restart: always
    volumes:
      - "/var/run/docker.sock:/var/run/docker.sock"
      - "./portainer_data:/data"

    # Une fois que traefik aura été configuré on pourra supprimer cette partie
    ports:
      # 9000 est le port non sécurisé de portainer (qui va après passer par traefik pour être
      sécurisé)
      - "9000:9000"

    labels:
      - "traefik.enable=true"

    # On va passer les trucs du port 9000 au port 443 pour être sécurisé par traefik avec
    letsencrypt
    # Vous pouvez changer le nom de domaine, le nom de l'entrypoint et le nom du resolver de
    certificat
    - "traefik.http.services.portainer.loadbalancer.server.port=9000"
    - "traefik.http.routers.portainer.rule=Host(`votre-nom-de-domaine-ici.net`)"
```

- "traefik.http.routers.portainer.entrypoints=https"
- "traefik.http.routers.portainer.tls.certresolver=le"

networks:

- traefik

networks:

traefik:

external: true

Après on peut le lancer en faisant :

```
sudo docker-compose up -d
```

Ensuite on peut configurer portainer sur son interface à l'adresse `http://ip-du-serveur:9000`. Ensuite on peut ajouter traefik en cliquant sur l'environnement local puis sur Stack puis sur "Add new stack"

version: '3'

services:

traefik:

image: "traefik:latest"

restart: always

ports:

- "80:80"
- "443:443"

command:

- "--providers.docker=true"
- "--providers.docker.exposedbydefault=false"
- "--entrypoints.http.address=:80"
- "--entrypoints.https.address=:443"
- "--certificatesresolvers.le.acme.httpchallenge=true"
- "--certificatesresolvers.le.acme.httpchallenge.entrypoint=http"

volumes:

- "/var/run/docker.sock:/var/run/docker.sock:ro"

networks:

- traefik

networks:

traefik:

```
external: true
```

Une fois cela fait on peut le deployer et normalement portainer devrait être accessible à l'adresse `https://votre-nom-de-domaine.net` si c'est bien le cas alors on va pouvoir supprimer l'exposition du port non sécurisé :

```
# On peut commenter la section "ports" du docker-file d'origine puis le relancer
❏❏
#ports:
#- "9000:9000"
```

Et enfin relancer portainer :

```
sudo docker-compose up -d
```

Et voilà maintenant portainer est installé et configuré !



# ?? Archiver un site et des images

Voici comment j'ai fait pour faire un backup complet de mon site avec Docker. Ce site était un site Bookstack qui était donc composé de Bookstack, de dépendances PHP ainsi que d'une base de donnée mariadb.

Malheureusement je ne peux pas simplement migrer le site vers une nouvelle instance car la version de Bookstack est trop ancienne pour être compatible.

D'abord je fais un backup des fichiers important sur le serveur

```
# Je vais dans le dossier du site
cd /var/www/BookStack

# Je génère un script sql qui va me permettre de restaurer ma base de donnée "bookstack_db"
mysqldump -u root bookstack_db > bookstack-backup-db.sql

# Ensuite je crée une archive tar qui contient tous mes fichiers importants
tar -czvf bookstack-backup-files.tar.gz .env public/uploads storage/uploads

# Enfin je vais prendre note de la version de bookstack
cat version
```

Ensuite je récupère ces fichiers dans un nouveau dossier sur mon ordi :

```
mkdir bookstack-backup
cd bookstack-backup
rsync -rv debian@snowcode.ovh:/var/www/BookStack/bookstack-backup-* .
```

Ensuite je vais créer un docker-compose dans ce dossier avec le contenu suivant :

```
version: "2"
services:
  bookstack:
    image: ghcr.io/linuxserver/bookstack:version-v0.31.4 # Ici je met la version de BookStack
    container_name: bookstack
    environment:
```

```

- PUID=1000
- PGID=1000
- APP_URL=http://localhost:8080
- DB_HOST=bookstack_db
- DB_PORT=3306
- DB_USER=bookstack
- DB_PASS=secret
- DB_DATABASE=bookstackapp
volumes:
  - ./bookstack_files:/config
ports:
  - 8080:80
depends_on:
  - bookstack_db
bookstack_db:
  image: lscr.io/linuxserver/mariadb:10.6.10
  container_name: bookstack_db
  environment:
    - PUID=1000
    - PGID=1000
    - MYSQL_ROOT_PASSWORD=secret
    - TZ=Europe/Brussels
    - MYSQL_DATABASE=bookstackapp
    - MYSQL_USER=bookstack
    - MYSQL_PASSWORD=secret
  volumes:
    - ./bookstack_db:/config

```

Maintenant on peut lancer ce serveur

```
sudo docker-compose up
```

Et on va placer les fichiers dans les différents volumes créés

```

sudo su # Il faut être root pour faire tout ce qui suis
mv bookstack-backup-files.tar.gz bookstack_files
mv bookstack-backup-db.sql bookstack_db
cd bookstack_files
tar xvzf bookstack-backup-files.tar.gz

```

Enfin on va entrer dans le conteneur de la base de donnée pour y importer notre script

```
sudo docker exec -it bookstack_db bash -c "mysql -u bookstack -p bookstackapp <
/config/bookstack-backup-db.sql"
# Le mot de passe est "secret"
```

Maintenant on peut accéder au site à l'adresse `http://localhost:8080`

# Archiver les images

Maintenant si on veut aller encore plus loin on peut également archiver les images de mariadb et bookstack.

Tout d'abord on va lister les images:

```
sudo docker images
```

Maintenant on va prendre les ID de celles de bookstack et mariadb et on va les exporter dans un fichier .tar

```
sudo docker image save aad0c49aebf3 -o bookstack.tar
sudo docker image save 39a4293c3071 -o mariadb.tar
```

Et voilà !

Maintenant si on souhaite réimporter ces images :

```
sudo docker image import bookstack.tar ghcr.io/linuxserver/bookstack:version-v0.31.4
sudo docker image import mariadb.tar lscr.io/linuxserver/mariadb:10.6.10
```

Les images ont maintenant été réimportées !