Cours DevOps

EMISA Dev Web – Octobre 2024

Plan du cours

1. Introduction au DevOps
2. Automatisation : Bash
3. Versionning : Git
4. Conteneurisation : Docker
5. CI/CD : Jenkins

Contenus disponibles sur ce lien github

1. Introduction au DevOps

- De quoi parle-t-on ?
- Origines
- Intérêts
- Un exemple
- *CALMS*
- DevOps en entreprise
- Les outils

1. Introduction au DevOps
De quoi parle-t-on ?

Le DevOps est une **démarche** à laquelle sont associés plusieurs **outils**.

Les principaux avantages sont :

- L’accélération des déploiements
- La réduction du *Time-to-market* (= temps d'arrivée d'une fonctionnalité sur le marché, c'est-à-dire le temps entre le moment de décision de la création de cette fonctionnalité, et son arrivée sur le produit final en production)

1. Introduction au DevOps
Origines

- Terme apparu pour la première fois en 2008
- Problématique du *Mur de la confusion* : séparation des aspects **dev** (pour développeurs) et **ops** (pour opérations) d'une application entre différentes équipes qui ont des objectifs non concordants :
  - Les dev doivent modifier et améliorer l'application
  - Les ops doivent maintenir une application stable, opérationnelle et de qualité
- Le **DevOps est un ensemble de pratiques** qui se concentre sur l'**automatisation des processus** entre les équipes de développement, et les équipes de production et de maintien de l'application développée

1. Introduction au DevOps
Intérêts

Permet de développer, tester et livrer des applications plus rapidement et avec plus de fiabilité

Avantages du DevOps :

- confiance des équipes entre elles
- accélération des livraisons et des déploiements
- résolution des tickets plus rapide
- gestion plus efficace des tâches non planifiées…

1. Introduction au DevOps
Un exemple

- Etsy = entreprise de vente en ligne créée en 2005
- Entreprise traditionnelle avec séparation des devs et des ops. Quelques chiffres : les déploiements prenaient généralement 4h et se faisaient 2 fois par semaine
- En 2008/2009, remise en question et passage au DevOps : collaboration efficace, implémentation de pipelines d'intégration continue et automatisation
- Conséquences :
  - Etsy déploie maintenant plus de 50 fois par jour, et de manière plus sûre, chaque développement étant testé avant d'être déployé en production.
  - Amélioration du *mean time to recovery* (MTTR = capacité à rendre à nouveau opérationnel un système). maximum 4min vs plusieurs heures avant

1. Introduction au DevOps
CALMS

- **Culture** : culture d'entreprise entre les dev et les ops qui doivent travailler ensemble avec une vision et des objectifs communs
- **Automatisation** : tout ce qui peut être automatisé doit l'être => déploiements programmés et fréquents, environnements créés à la demande et tests automatisés.
- **Lean** : gestion des ressources sans gaspillage par exemple en identifiant quels processus prennent le plus de temps sans pour autant améliorer le produit final
- **Mesure** : pour tout automatiser et tout optimiser, nécessité de tout mesurer
- **Sharing** : dev et ops doivent partager des moments et les responsabilités

1. Introduction au DevOps
DevOps en entreprise

Plusieurs approches possibles d'implémentation d'une approche DevOps en entreprise.

Le site https://web.devopstopologies.com/ répertorie divers *patterns* et *anti-patterns* de mise en place possible (ce qu'il faut et ne faut pas faire)

1. Introduction au DevOps
Les outils

Pour mettre en œuvres les pratiques DevOps, de nombreux outils existent. En voici quelques-uns :

- **Jenkins** : serveur d'intégration continue (CI) et de livraison continue (CD) open-source pour automatiser le processus de build, test et déploiement
- **Docker** : plateforme de conteneurisation pour créer, déployer et exécuter des applications dans des conteneurs légers et portables
- **Kubernetes** : système d'orchestration de conteneurs pour gérer et de déployer à grande échelle des applications conteneurisées (souvent avec Docker)
- **Git** : système de contrôle de version distribué qui permet aux équipes de collaborer sur le code source
- **Ansible** : outil d'automatisation et de gestion de configuration open-sourcepour déployer des configurations sur plusieurs serveurs de manière automatisée
- **Terraform** : outil d'infrastructure as code (IaC) qui permet de définir et de provisionner des infrastructures dans le cloud à l'aide de fichiers de configuration
- **Prometheus** : système de surveillance open-source conçu pour collecter et analyser des métriques système et des données en temps réel (souvent utilisé avec des dashboard Grafana)
- **Grafana** : outil de visualisation de données et de monitoring qui permet de créer des tableaux de bord interactifs à partir de données issues de différentes sources (Prometheus, Elasticsearch, InfluxDB...)
- **ELK Stack (Elasticsearch, Logstash, Kibana)** : ensemble d'outils pour la gestion et l'analyse des logs. Elasticsearch est utilisé pour la recherche, Logstash pour la collecte et le traitement des logs, et Kibana pour la visualisation
- **Vault (HashiCorp)** : Un outil de gestion de secrets qui permet de stocker et de gérer en toute sécurité des informations sensibles, comme des clés API, des mots de passe, et des certificats.

2. Automatisation : Bash

- Le shell Bash
- Intérêts en DevOps et admin sys
- Commandes de base
- Manipulation de textes
- Scripts Bash
- Exercices pratiques

2. Automatisation : Bash
Le shell Bash

- **Shell** : interface système d’exploitation
- **Définition** : Bash (Bourne Again SHell) est un interpréteur de commandes Unix utilisé dans les systèmes Linux et macOS pour interagir avec le système d'exploitation via une interface en ligne de commande
- **Historique** : Bash est une amélioration du shell original Bourne (sh), incorporant des fonctionnalités de shells comme csh et ksh
- **Rôle en DevOps** : Bash est essentiel pour interagir avec le système et donc pour l'automatisation des tâches, la gestion des configurations et le déploiement de logiciels ou d’applications sur des serveurs

2. Automatisation : Bash
Intérêts en DevOps et admin sys

- **Automatisation** : écriture de scripts pour automatiser des tâches répétitives
- **Portabilité** : scripts Bash exécutables sur la plupart des systèmes Unix/Linux
- **Puissance** : large gamme de commandes et d'utilitaires pour manipuler les fichiers, les processus et les réseaux
- **Intégration** : bonne intégration avec d'autres outils DevOps comme Git, Docker, Jenkins, etc...

2. Automatisation : Bash
Commandes de base

Navigation dans le système de fichiers

- `ls` : liste les fichiers et dossiers du répertoire courant. Options `-l` et `-a`
- `cd` : change le répertoire courant
- `pwd` : affiche le répertoire courant

Gestion des fichiers et dossiers

- `cp` : copie des fichiers ou dossiers
- `mv` : déplace ou renomme des fichiers ou dossiers
- `rm` : supprime des fichiers
- `mkdir` : crée un nouveau dossier
- `touch` : crée un fichier vide ou met à jour la date de modification

2. Automatisation : Bash
Manipulation de texte

Affichage du contenu

- `cat file` : affiche le contenu d'un fichier
- `less` et `more` : affiche le contenu page par page pour des fichiers longs

Recherches

- `grep` : recherche une chaîne de caractères dans un fichier

Flux et redirections

- `>` : redirige la sortie standard vers un fichier en écrasant le contenu
- `>>` : redirige la sortie standard vers un fichier en ajoutant à la fin
- `|` : redirige la sortie standard d'une commande vers l'entrée standard d'une autre

2. Automatisation : Bash
Scripts Bash

Création et exécution de scripts

- script Bash = fichier texte contenant une série de commandes commençant généralement par `#!/bin/bash` pour indiquer l'interpréteur
- Pour donner les permissions d'exécution : `chmod +x script.sh`
- Pour lancer le script : `./script.sh`

Utilisation des arguments de ligne de commande

- `$0` : nom du script
- `$1`, `$2` , etc... : arguments passés au script

2. Automatisation : Bash
Scripts Bash

Variables et contrôles de flux

- Variables : assignation avec `=` sans espaces
-
```
if [ condition ]; then
  commandes
fi
```
-
```
for var in liste; do
 commandes
done
```
-
```
while [ condition ]; do
  commandes
done
```

2. Automatisation : Bash
Exercices pratiques

Exercices disponibles sur ce lien github

Exercice 1 : Navigation et manipulation de fichiers

1. Créez un dossier nommé `devops_bash`
2. À l'intérieur de ce dossier, créez trois fichiers : `app.py`, `requirements.txt`, `README.md`
3. Copiez le fichier `app.py` et nommez la copie `app_backup.py`
4. Affichez le contenu du dossier `projet_devops`
5. Supprimez le fichier `app_backup.py`

Exercice 2 : Sauvegarder des fichiers

- Écrivez un script `backup.sh` qui copie tous les fichiers `.py` du dossier courant vers un dossier `backup` (à créer s'il n'existe pas).

Exercice 3 : Archiver un répertoire

- Créer un script Bash qui archive un répertoire donné et affiche la taille de l'archive.

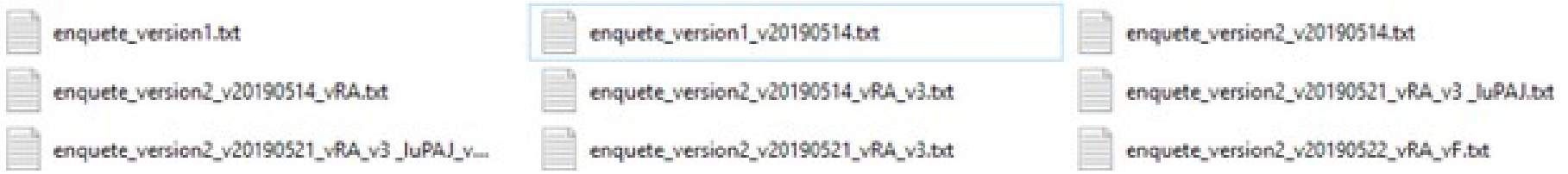
Exercice 4 : Monitoring et logging simple

- Écrire un script qui surveille l'utilisation du CPU et de la mémoire du système toutes les 5 secondes et enregistre dans un fichier si l'utilisation dépasse 80%.

3. Versionning : Git

- Problématique
- Définition et intérêts
- Exemples d’autres systèmes
- Principes
- Commandes de base
- Interactions avec un dépôt distant
- Travailler en équipe
- Exercices pratiques

3. Versionning : Git
Problématique



- **Gestion « artisanale » des versions** :
  - Erreur humaine possible
  - Manque d’information sur les versions et les différences entre versions
  - Reprise du projet compliqué pour une nouvelle personne
  - Partage des fichiers délicat
  - Collaboration à plusieurs en même temps complexe voire impossible

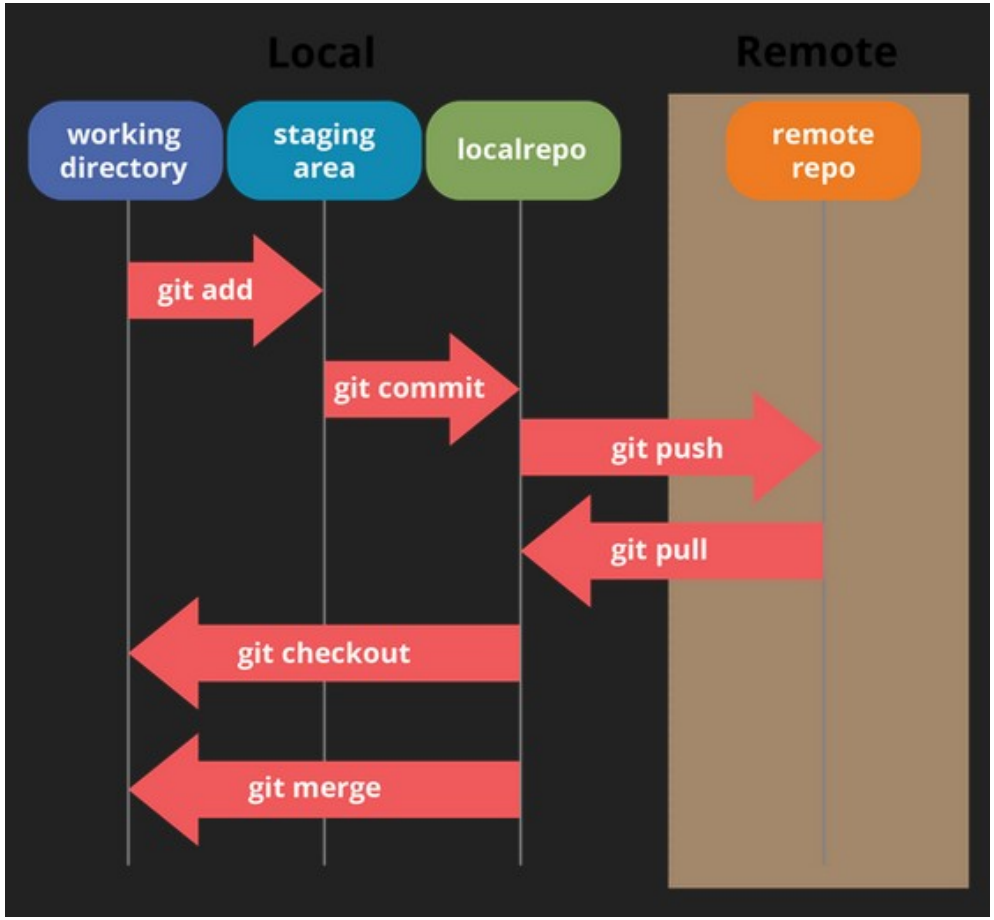
3. Versionning : Git
Définition et intérêts

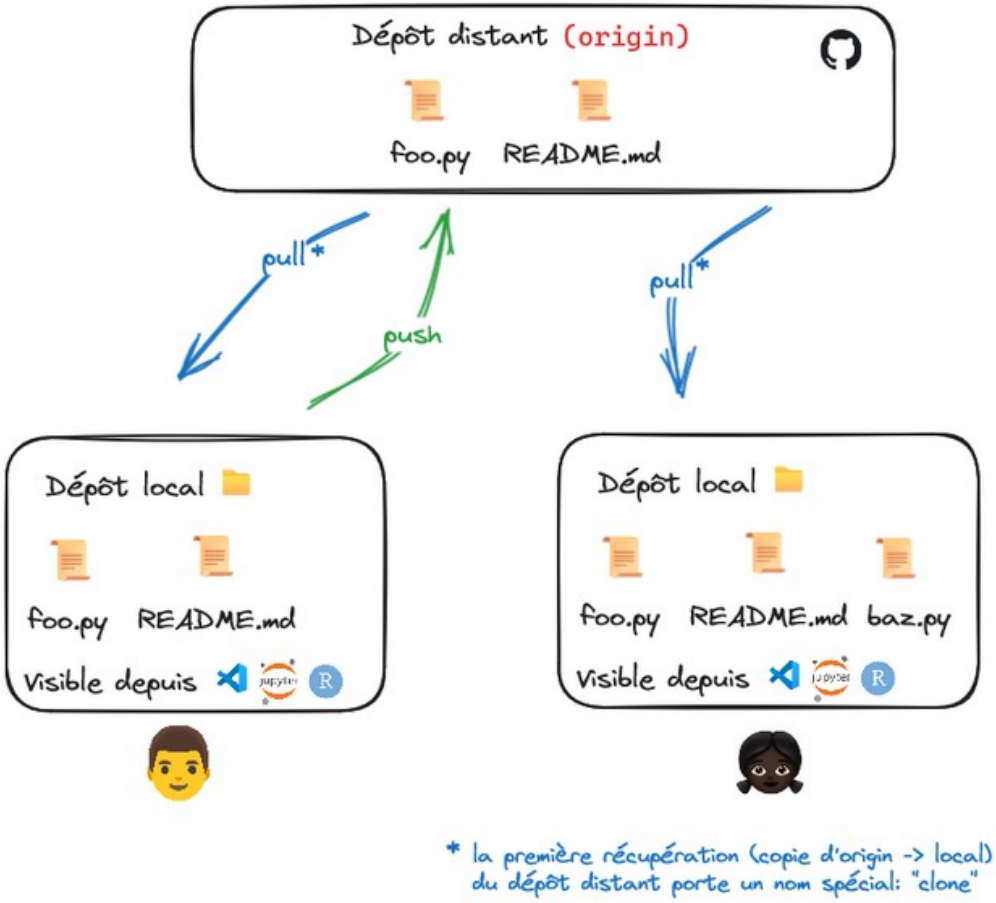
- **Définition** : Git est un système de gestion de versions distribué, créé par Linus Torvalds en 2005 qui permet de suivre les modifications apportées à des fichiers au fil du temps, de revenir à des versions antérieures, et de travailler à plusieurs sur le même projet sans perdre de travail
- **Intérêts** :
  - Historisation des changements : suivre les modifications et revenir à des versions antérieures si nécessaire
  - Collaboration : travailler en équipe sans écraser le travail des autres
  - Branchement : expérimenter de nouvelles fonctionnalités sans affecter la version stable
  - Distribution : chaque développeur possède une copie complète du dépôt

3. Versionning : Git
Exemples d’autres systèmes

- **Autres systèmes** : SVN, Mercurial et Perforce
- **Différence avec Git** :
  - Distribué vs Centralisé : Git est distribué, chaque clone est un dépôt complet, contrairement à SVN par exemple qui est centralisé
  - Performance : Git est généralement plus rapide pour les opérations locales
  - Branches légères : La gestion des branches est plus flexible et légère avec Git

3. Versionning : Git Principes





3. Versionning : Git
Commandes de base

- `git init` : initialisation d'un dépôt = transforme un répertoire local en dépôt Git
- `git add` : ajoute des changements au staging area (zone d'indexation)
- `git commit` : enregistre les changements ajoutés avec un message descriptif
- `git log` : affiche l'historique des commits

3. Versionning : Git
Interactions avec un dépôt distant

Utilisation de plateformes (GitHub, GitLab, BitBucket,...) pour :

- collaborer avec d'autres développeurs
- sauvegarder le code en ligne
- bénéficier des fonctionnalités de gestion de projets (issues, pull requests)

Commandes de base pour les dépôts distants

- `git clone` : cloner un dépôt distant sur votre machine locale
- `git remote` : gérer les dépôts distants
- `git push` : envoyer les commits locaux vers le dépôt distant
- `git pull` : récupérer et fusionner les changements du dépôt distant

3. Versionning : Git
Travailler en équipe

Gestion des branches : une branche = version parallèle du code sur laquelle on peut travailler indépendamment

- `git branch` : liste les branches existantes ou crée une nouvelle branche
- `git checkout` : permet de naviguer entre les branches
- `git merge` : fusionne une branche dans la branche courante

Résolution de conflits : surviennent lorsque des changements incompatibles sont apportés à la même partie du code

1. Identifier les fichiers en conflit (Git les signale)
2. Éditer les fichiers pour résoudre les conflits manuellement
3. Ajouter les fichiers résolus avec git add.
4. Effectuer un commit pour finaliser la fusion.

3. Versionning : Git
Exercices pratiques

Exercice 1 : Initialisation d'un dépôt et 1er commit

1. Créez un dossier pour votre projet et initialisez un dépôt Git dans ce dossier.

2. Créez un fichier `README.md` avec une description de votre projet.

3. Ajoutez ce fichier à Git et validez le changement avec un commit.

Exercice 2 : Travailler avec les branches

1. Créez une nouvelle branche appelée `dev` pour y développer une fonctionnalité.

2. Ajoutez une modification dans cette branche et validez-la.

3. Fusionnez cette branche dans la branche principale main.

Exercice 3 : Résolution de conflits

1. Créez deux branches à partir de `main` : `dev1` et `dev2`.

2. Modifiez le même fichier dans les deux branches de manière conflictuelle (modifiez la même ligne).

3. Fusionnez `dev1` dans `main`, puis tentez de fusionner `dev2` (ce qui générera un conflit).

4. Résolvez manuellement le conflit et complétez la fusion.

Exercice 4 : Gestion des branches avec `git rebase`

1. Créez un dépôt Git avec une branche `main` et une branche `feature`.

2. Faites des commits dans les deux branches.

3. Utilisez `git rebase` pour réappliquer les commits de la branche `feature` sur `main`, réorganisant ainsi l'historique.

Exercice 5 : Sauvegarder un travail avec `git stash`

1. Modifiez un fichier dans votre projet sans le committer.

2. Utilisez `git stash` pour mettre de côté ces modifications.

3. Récupérez les modifications avec `git stash apply`.

Exercice 6 : Révisions et collaboration à plusieurs

1. Allez sur https://github.com/louiskuhn/git_revisions

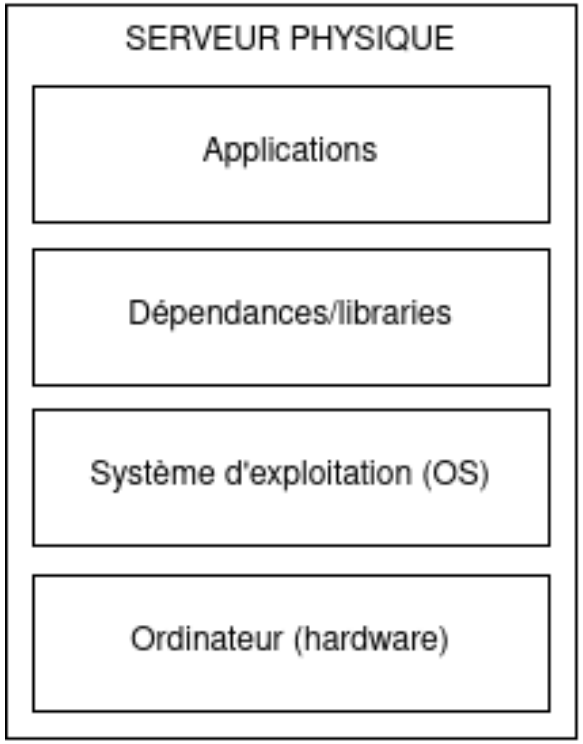
2. Faire les exos...

Exercices disponibles sur
ce lien github

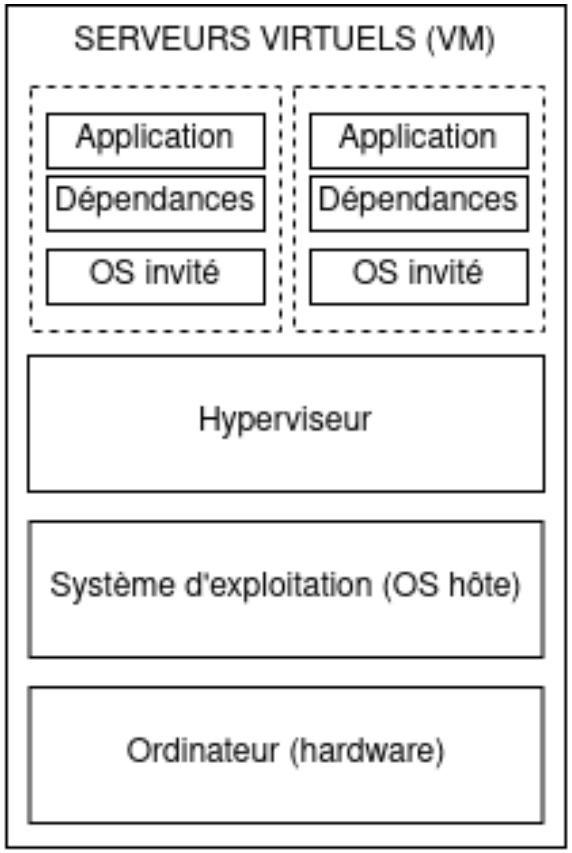
4. Conteneurisation : Docker

- Problématique
- La virtualisation
- Les containers
- Docker
- Concepts
- Les volumes
- Les réseaux
- Exercices pratiques

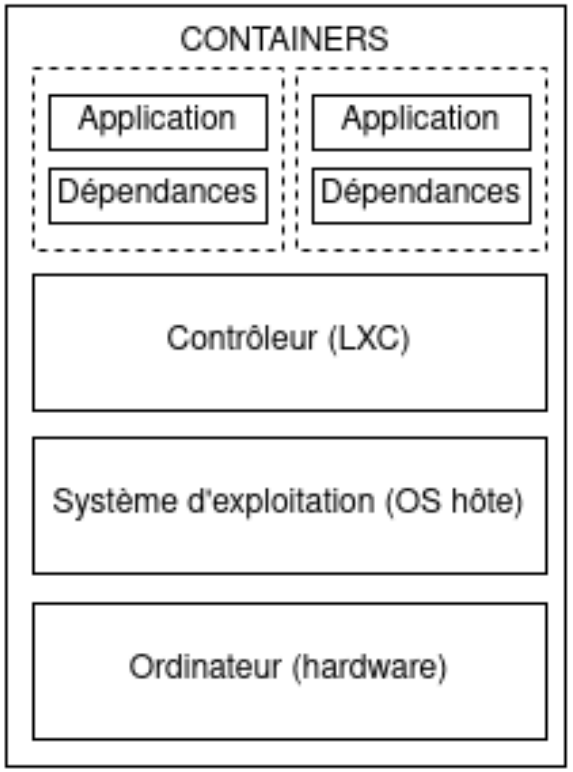
4. Conteneurisation : Docker
La virtualisation



Architecture simple mais **sous-exploitation des capacités** du serveur car non utilisées



Capacités du serveur utilisées mais non optimisées car espace mémoire, CPU et espace disque surchargés par les nombreux OS hôte

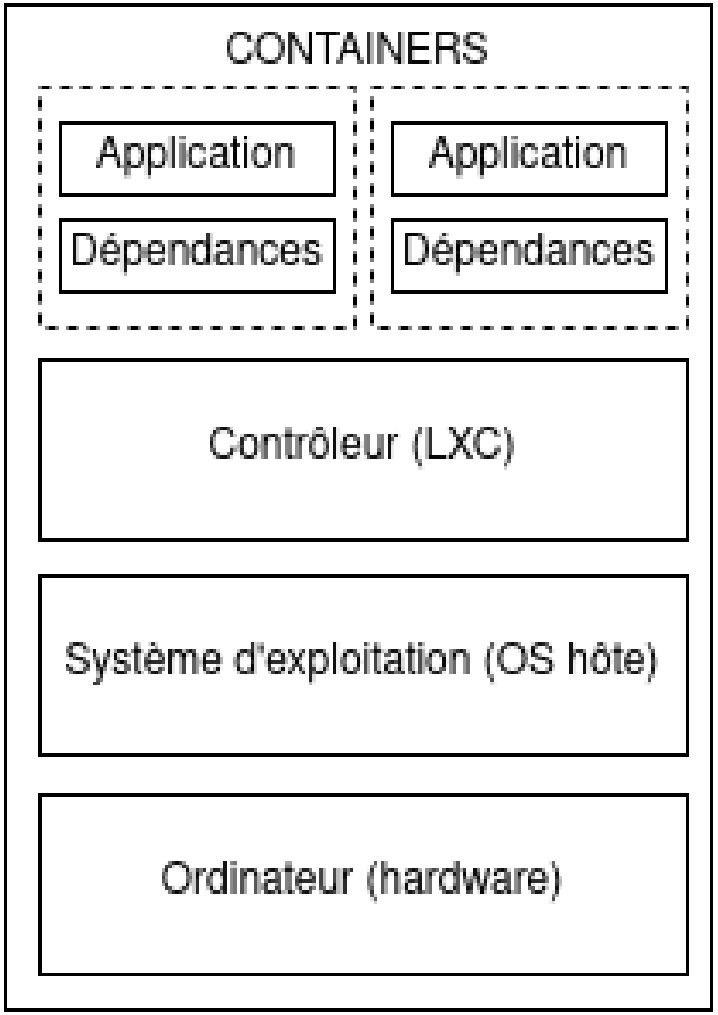


Capacités du serveur utilisées et optimisées car un seul OS

4. Conteneurisation : Docker
La virtualisation

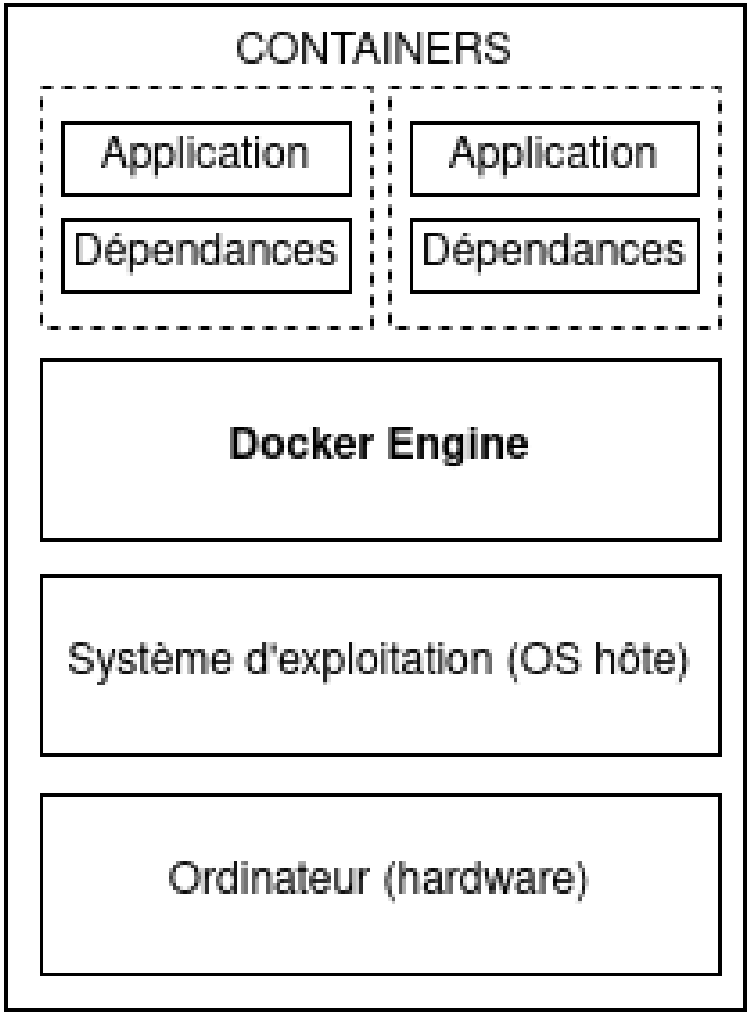
- **Hyperviseurs Type 2 - hyperviseurs hébergés** : VMWare, Fusion, VirtualBox…
- **Hyperviseurs Type 1 - hyperviseurs natifs** : VMWare, ESX, Microsoft Hyper-V…
- **Containers** : Docker, Kubernetes (k8s), OpenVZ…

4. Conteneurisation : Docker
Les containers



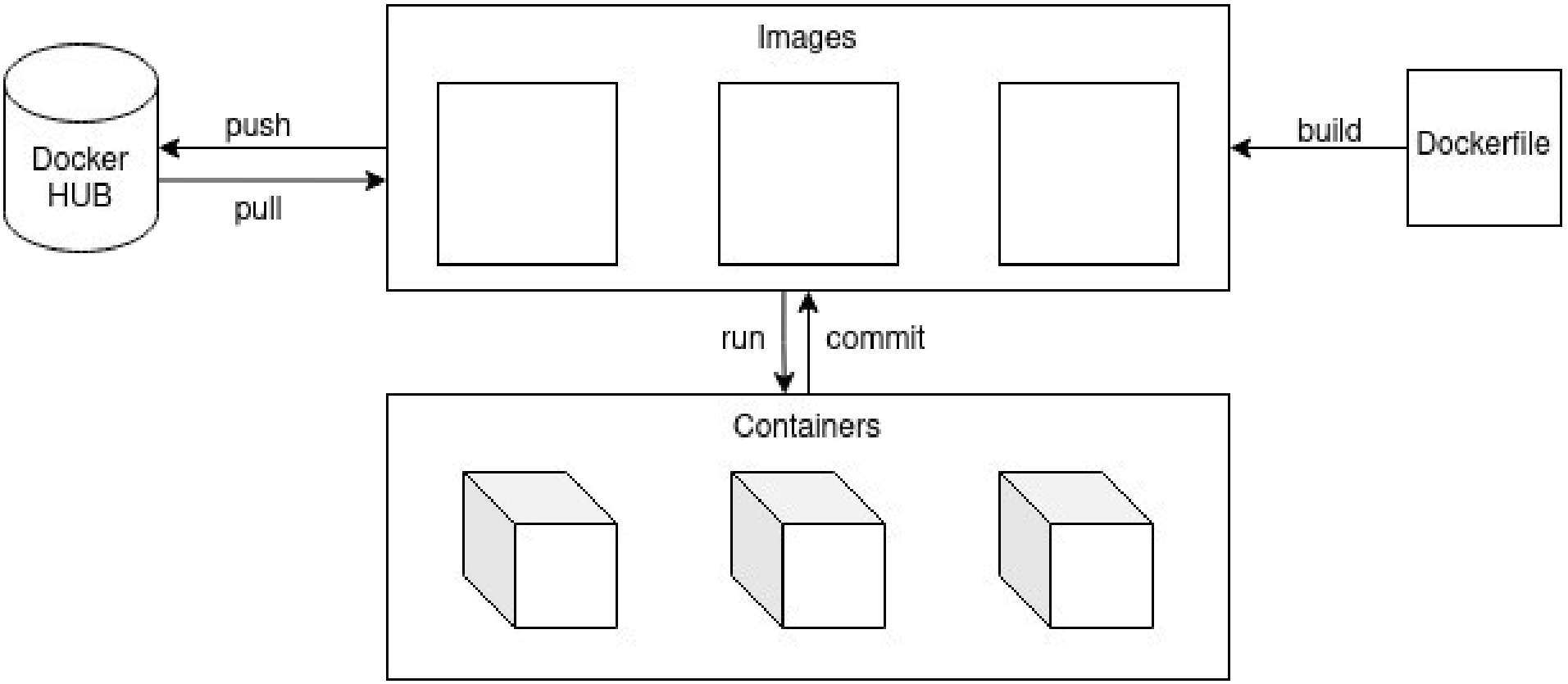
- **LinuX Containers (LXC)** = méthode de cloisonnement au niveau de l'OS basée sur 2 fonctionnalités du noyau Linux :
  - **Cgroups** = Control groups pour limiter et isoler les ressources
  - **Namespace** = méthode de cloisonnement des espaces de nommages pour rendre inaccessibles (même invisibles !) les ressources d'un groupe à l'autre
- Les containers permettent de **packager une application** avec **toutes les dépendances nécessaires** et la **configuration**
- Les containers sont **portables** et peuvent être facilement **partagés/déplacés**

4. Conteneurisation : Docker
Docker



- **Docker** : technologie de **virtualisation** par conteneurs reposant sur le **LXC** qui joue le rôle du **contrôleur**
- **Images** : **templates** prêts à l'emploi avec des instructions pour la création de conteneurs
- **Container** : un **empilement d'images** (en lecture seules) et une **couche accessible en écriture** pour la **configuration personnelle** de votre conteneur

4. Conteneurisation : Docker
Docker



4. Conteneurisation : Docker Concepts

- **HUB** = registre permettant
  - de télécharger des images docker gratuitement
  - d'héberger ses propres images docker publique (gratuit) ou privées (payant)
- **Image =** templates :
  - contenant l'ensemble des fichiers et des paramètres d'exécution par défaut du programme souhaité
  - « versionnée » avec des tags
  - Syntaxe : [registry]/[user/]image_name[:tag] (tag par défaut *latest*)
- **Container** = instance d'une image Docker :
  - Personnalisable avec différents paramètres (réseau, volumes, …)
  - Chaque container est isolé des autres à moins d'être associés explicitement (par exemple via un même réseau ou un volume partagé)

4. Conteneurisation : Docker
Les volumes

Par défaut, toutes les données sont **stockées dans le container** : suppression du container => données supprimées

Concept de « **volume** » pour permettre la **persistance** : les données sont extériorisées :

- Soit sur le système de fichier de l'hôte (comportement par défaut)
- Soit ailleurs, avec des plugins : AWS S3, NFS, ...

- Plusieurs types :
  - **Volumes anonymes** : créés par défaut si définis dans l'image mais non référencés lors de l'exécution
  - **Volumes hôtes** :

```
docker run -v "/home/username/data:/var/data" […] image_name
```

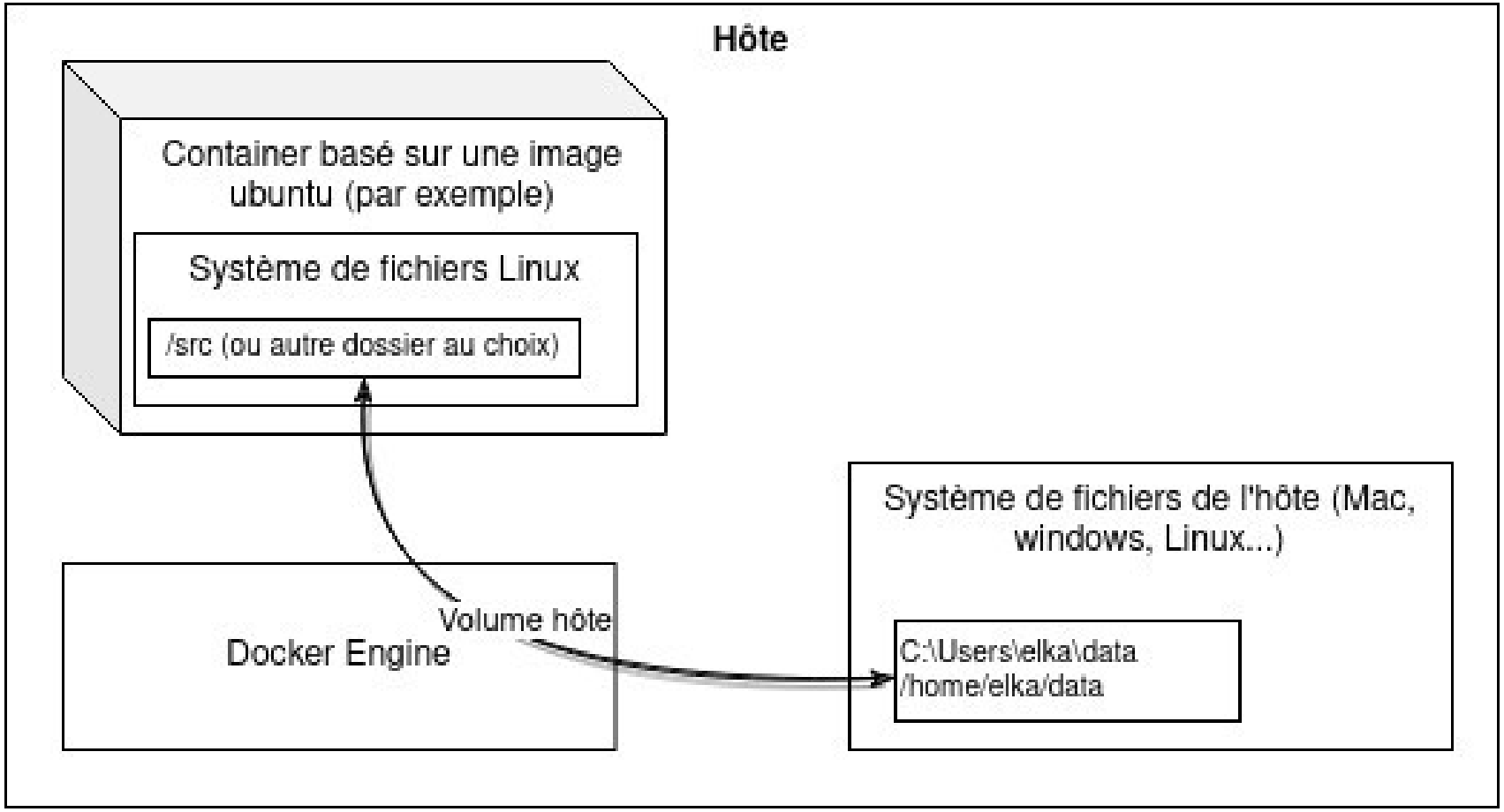
```
docker run -v "C:\Users\Username\data:/var/data" […] image_name
```

  - **Volumes nommés** :

```
docker volume create volume_name
```

```
docker run -v "volume_name:/var/data" […] image_name
```

4. Conteneurisation : Docker
Les volumes




```
docker run -v "/home/elka/data:/src" image_name [commande]
```

4. Conteneurisation : Docker
Les réseaux

- Docker crée des **réseaux virtuels** :
  - Plusieurs containers sur un même réseau peuvent interagir ensemble
  - Condition : avoir le port exporté dans l'image

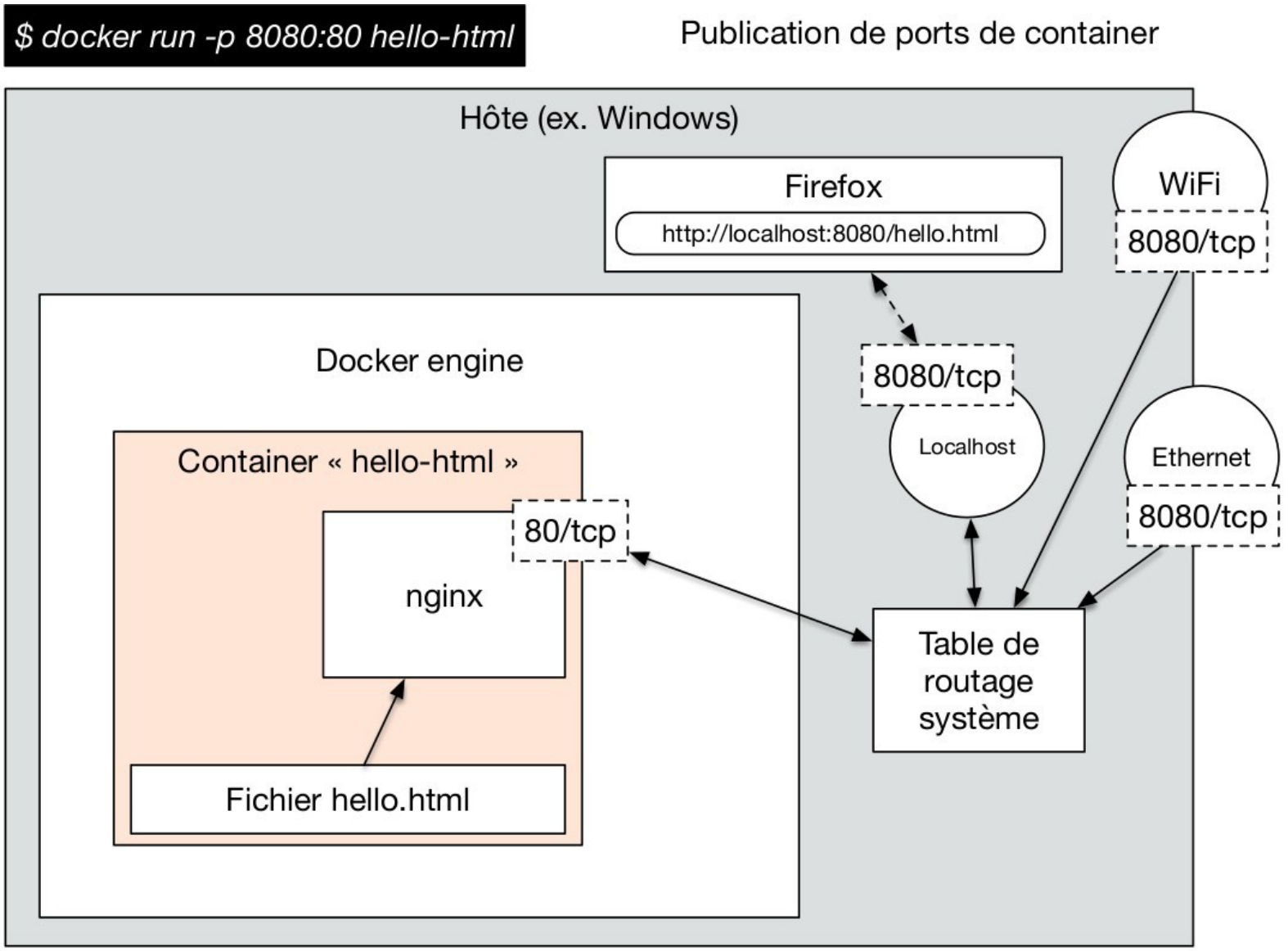
```
docker network create network_name
```

```
docker run --network network_name image_name
```

- Publication de ports : ports « fermés » par défaut depuis l'extérieur du conteneur, il faut « publier » le port :

```
docker run -p "port_externe:port_interne" image_name
```

4. Conteneurisation : Docker
Les réseaux



4. Conteneurisation : Docker
Exercices pratiques

Exercices disponibles sur ce lien github

5. CI/CD : Jenkins

- Concepts CI/CD
- Jenkins
- Jobs & Pipeline
- Exercices pratiques

5. CI/CD : Jenkins
Concepts CI/CD

- **Intégration Continue (CI)** : pratique de développement qui implique de fusionner fréquemment le code de chaque développeur dans un référentiel principal. Chaque intégration déclenche automatiquement des tests, permettant de détecter rapidement les erreurs et de garantir un code stable
- **Déploiement Continu (CD)** : automatise la livraison des versions validées du code dans des environnements de *staging* ou de production. Cela permet de publier rapidement et de façon fiable les changements de code
- **Intérêts dans le cycle de développement** :
  - *Qualité* : tests automatisés détectent les bugs plus tôt
  - *Rythme* : déploiements fréquents => livraison des fonctionnalités plus rapide
  - *Feedback* : problèmes identifiés et résolus plus vite

5. CI/CD : Jenkins
Jenkins

- **Jenkins** : outil logiciel open-source écrit en Java permettant d'automatiser les parties non humaines du développement logiciel. Jenkins utilise une architecture distribuée pour exécuter des builds sur plusieurs machines
- **Intérêts**
  - capacité à automatiser le build, les tests et le déploiement du code
  - communauté large et active.
  - extensibilité : plus de 1500 plugins pour intégrer des outils et services tiers
- **Fonctionnalités principales**
  - *Pipelines* : Définir des flux de travail complexes
  - *Plugins* : nombreux plugins pour intégrer différents outils, notamment :
    - Git : intégration avec des dépôts Git.
    - Pipeline : création de pipelines sous forme de code (Jenkinsfile).
    - Docker : exécution de builds dans des conteneurs Docker.
  - *Interface web* : pour configurer et surveiller les builds

5. CI/CD : Jenkins
Jobs & Pipeline

- ***Job*** : tâche automatisée, comme compiler du code, exécuter des tests, déployer une application
- **2 types** :
  - *Freestyle Job* : configuration simple, utile pour les tâches de base
  - *Pipeline Job* : permet de définir des étapes CI/CD dans un fichier `Jenkinsfile`, idéal pour les pipelines complexes.
- **Étapes typiques d'un pipeline** :
  - Build : compilation du code source
  - Test : exécution des tests unitaires et autres tests
  - Analyse : vérification du code avec des outils d'analyse statique
  - Déploiement : mise en production ou en environnement de test

5. CI/CD : Jenkins
Exercices pratiques

Exercices disponibles sur ce lien github