DevOps & MLOps

Filière INDIA (3ème année)

Pr. Abderrahim EL QADI

Département Mathématique appliquée et Génie Informatique

ENSAM-Université Mohammed V de Rabat

A.U. 2025-2026

Plan

Partie 1 : DevOps
1. Introduction à DevOps
2. DevOps et le cycle de vie des applications
3. Outils de DevOps
   - 3.1. Gestion du code source avec Git et GitHub
   - 3.2. Conteneurisation avec Docker
   - 3.3. Orchestration avec Kubernetes (introduction)
   - 3.4. Infrastructure as Code (IaC)

Partie 2 : MLOps
1. Introduction à MLOps
2. Cycle de vie d'un projet ML
3. Intégration continue / Livraison continue pour le ML (CI/CD for ML)
4. Déploiement de modèles

Partie 1 : DevOps

1. Introduction à DevOps
2. DevOps et le cycle de vie des applications
3. Outils de DevOps
   - 3.5. Gestion du code source avec Git et GitHub
   - 3.6. Conteneurisation avec Docker
   - 3.7. Orchestration avec Kubernetes (introduction)
   - 3.8. Infrastructure as Code (IaC)

Références

1. DevOps Tools for Java Developers Best Practices from Source Code to Production Containers. Stephen Chin, Melissa McKay, Ixchel Ruiz & Baruch Sadogursky
2. Docker on Windows From 101 to production with Docker on Windows. Elton Stoneman
3. DevOps, Intégrez et déployez en continu, Edition Eni
4. DevOps 4 Null. Une introduction à devops
5. DevOps Introduction. Thomas Ropars (https://tropars.github.io/)
6. https://azure.microsoft.com/fr-fr/overview/what-is-devops/#culture
7. https://aws.amazon.com/fr/docker/
8. https://www.jenkins.io

1. Introduction à DevOps

– DevOps est la contraction des mots « **développement** » et « **opérations** ».
– Il s'agit d'une méthode de travail où les **équipes de développement, d'exploitation** informatique et de sécurité s'associent pour créer, tester une application et fournir un retour d'information régulier tout au long du cycle de vie du développement logiciel.

– **C'est quoi un ingénieur DevOps ?**

C'est un professionnel de l'informatique qui travaille avec des développeurs de logiciels, des opérateurs et des administrateurs de systèmes, du personnel d'exploitation informatique et d'autres personnes pour superviser et/ou gérer les versions ou les déploiements de code sur la base de l'intégration continue / du développement continu.

– La **culture DevOps** consiste essentiellement à être plus agile, à livrer des produits plus rapidement et de meilleure qualité.

– **Les 3 piliers : Culture, Automatisation, Mesure**

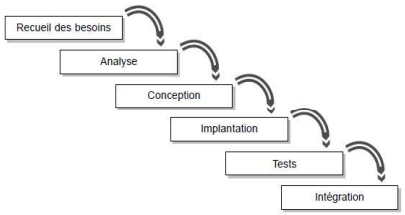
1. **Culture**	- Collaboration étroite entre les équipes de développement (Dev) et d'exploitation (Ops) - Communication transparente, feedback continu, apprentissage collectif - Responsabilité partagée
2. **Automatisation**	- Réduction des tâches manuelles, répétitives et sujettes à l'erreur - Chaîne CI/CD (Intégration Continue / Livraison Continue) - Infrastructure as Code (IaC) - Tests automatisés (unitaires, d'intégration, de sécurité) - Déploiements automatisés, rollback, provisioning
3. **Mesure**	- Suivi des performances du système et des équipes - Indicateurs clés (KPI) : Fréquence de déploiement, Délai de livraison (lead time)

– **Quels sont les principaux avantages du DevOps ?**

- DevOps accroît la coopération et la compréhension entre les départements des opérations et du développement.
- DevOps augmente la cadence de production et de déploiement des développements logiciels par le biais de systèmes de production appelés livraison et déploiement continus, ce qui se traduit par une plus grande agilité de développement.
- DevOps offre une grande agilité, ce qui se traduit par une amélioration de la qualité grâce à une vérification quasi-continue des produits dès les premiers stades du développement et à la réduction des erreurs potentielles.
- DevOps assure une plus grande sécurité des applications développées, grâce notamment à l'automatisation des procédures de contrôle des menaces et des vulnérabilités.
- DevOps permet une livraison plus flexible et une meilleure adaptation à l'environnement que les systèmes de développement traditionnels dans lesquels le temps de déploiement est plus long

2. DevOps et le cycle de vie des applications

– **Cycle de vie classique (cascade) des applications**



- Le coût lié à la correction d'un défaut augmente à chaque étape.
- En détectant les problèmes plus tôt, on crée des logiciels plus efficaces, plus rapidement et avec moins d'efforts.

Vision classique

- 60% de programmation
- 20% de tests
- 10% d'intégration
- 10% de documentation

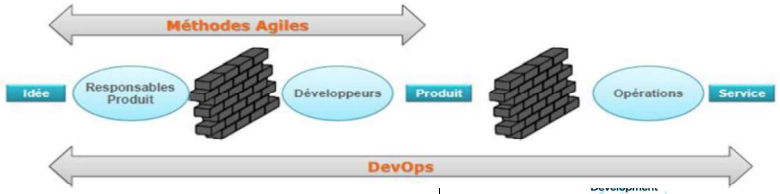
DevOps

25% de programmation
10% de tests
10% d'intégration
15% de documentation
10% automatisation des tests
10% automatisation du déploiement
20% qualité/validation

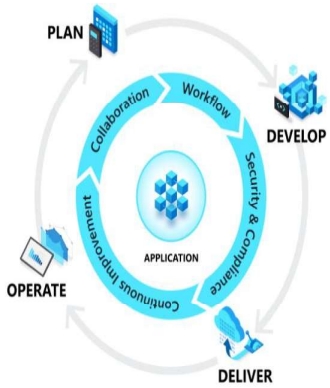
– **DevOps est une démarche de collaboration agile** entre développeurs, équipes opérationnelles et métiers (Business) sur l'ensemble du cycle de vie du produit.

Elle étend Agile en intégrant les opérations dans les itérations

Elle influence le cycle de vie des applications tout au long de leurs phases de planification, de développement, de livraison et d'exploitation



En adoptant les pratiques DevOps, les équipes veillent à assurer la fiabilité, la haute disponibilité du système, et visent à éliminer les temps d'arrêt tout en renforçant la sécurité et la gouvernance.



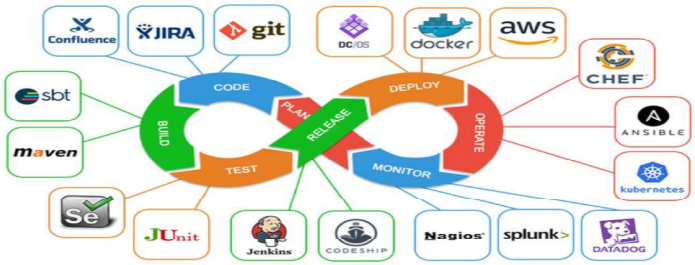
Planifier (Plan): les équipes DevOps imaginent, définissent et décrivent les fonctionnalités des applications et des systèmes qu'elles créent:
Création de backlogs, suivi des bogues, gestion du développement logiciel agile avec Scrum, l'utilisation de tableaux Kanban et la visualisation des progrès réalisés grâce aux tableaux de bord.

Développement (Develop) : comprend tous les aspects du codage (écriture, test, révision et intégration du code par les membres de l'équipe), ainsi que la génération de ce code dans des artefacts de build pouvant être déployés dans divers environnements.

Fournir (Deliver): la livraison consiste à déployer des applications dans des environnements de production de manière cohérente et fiable.
La phase de livraison englobe également le déploiement et la configuration de l'infrastructure de base entièrement régie qui constitue ces environnements.

Opérer (Operate): la phase d'exploitation implique la maintenance, la supervision et le dépannage des applications dans les environnements de production.

3. Outils de DevOps



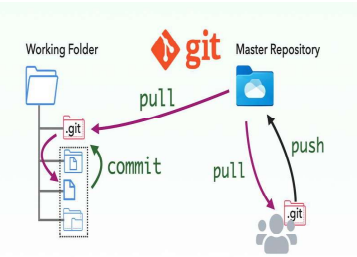
Git, Github, Gitlab	Gestion du code source
Jenkins, GitlabCI, Bamboo, TeamCity	Intégration continue et de déploiement continu (CI-CD)
Ansible, Terraform	Automatisation
Prometheus et Grafana, Elasticsearch, Kibana	Monitoring et Alerte
Jira et Trello	Gestion de projet
Docker, Kubernetes	Conteneurs
Google Cloud Platform, AWS, et Azure	Cloud providers
JUnit, Postman	Automatisation des Tests

3.1. Git / GitHub

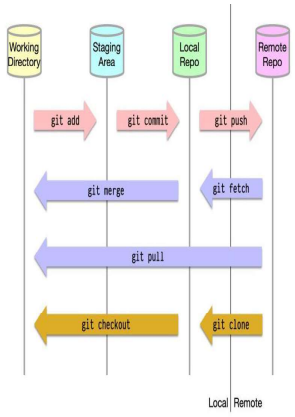
– **Git** est un projet open-source qui a été lancé en 2005.
- Il s'agit d'un système de contrôle de version distribué.
- Tout développeur de l'équipe ayant un accès autorisé peut gérer le code source et l'historique des modifications à l'aide des outils de ligne de commande Git.

– **Github** est une plateforme de gestion et d'organisation de projets basée sur le cloud qui intègre les fonctions de contrôle de version de Git.

– Le stockage du code dans un « référentiel » sur GitHub permet de :
• Exposer ou partager le travail.
• Observer et gérer les évolutions du code au fil du temps.
• Offrir à d'autres individus la possibilité de réviser le code et de proposer des améliorations.
• Travailler ensemble sur un projet commun sans avoir peur que les modifications affectent le travail de vos collaborateurs avant que vous ne soyez prêt à les intégrer.



– Principales Fonctionnalités de GitHub



git **init**	permet d'initialiser un dépôt local vide dans le répertoire courant
git **clone** <url-du-dépôt>	Cloner un dépôt
git **add** <fichier>	Ajouter des changements pour ajouter tous les fichiers git **add .**
git **commit -m** "Message de commit"	permet de valider les modifications sur le dépôt local
git **push origin** <nom-de-la-branche>	permet d'envoyer les modifications indexées au dépôt distant
git **pull origin** <nom-de-la-branche>	Tirer les changements du dépôt distant
git **checkout -b** <nom-de-la-branche>	Créer une nouvelle branche
git **checkout** <nom-de-la-branche>	Changer de branche
git **merge** <nom-de-la-branche>	Fusionner une branche dans la branche actuelle
git **remote add** <nom> <url>	permet de lier le dépôt local (initie par init) `a un dépôt distant (url sur GitHub) ;

Exemple 1 : Contribuer à un Dépôt existant

1. Télécharger un référentiel depuis GitHub	git **clone** https://github.com/owner/repo.git owner/repo.git est le nom utilisé pour cloner
2. Accéder au répertoire de dépôt local	cd my_repertoire
3. Créer une nouvelle Branche	git **branch** my-branch
4. Basculer vers la nouvelle Branche	git **checkout** my-branch
5. Apporter des Modifications	Modifier les fichiers nécessaires, par exemple, file1.md et file2.md, à l'aide d'un éditeur de texte.
6. Ajouter ce fichier à l'index de Git	git **add** file1.md file2.md
7. Effectuer un commit pour enregistrer les modifications avec un message descriptif	git **commit -m** "my snapshot"
8. Pousser les changements vers la branche correspondante sur GitHub. Utiliser l'option --set-upstream pour lier la branche locale à la branche distante.	git **push --set-upstream** origin my-branch

Exemple 2 : Démarrer un nouveau dépôt et le publier sur GitHub

1. Créer un nouveau répertoire et l'Initialiser avec Git	git **init** my-repo
2. Accéder au répertoire créer	cd my_repertoire
3. Créer un fichier README.md qui décrit le projet	
4. Ajouter le fichier README.md	git **add** README.md
5. Effectuer un commit pour enregistrer les modifications avec un message descriptif	git **commit -m** "add README to initial commit"
6. Fournir le chemin du dépôt créé sur GitHub	git **remote** add origin https://github.com/YOUR-USERNAME/YOUR-REPOSITORY-NAME.git
7. Pousser les modifications vers GitHub vers la branche principale (main) du dépôt distant	git **push --set-upstream** origin main

Exemple 3 : Contribuer à une Branche existante sur GitHub

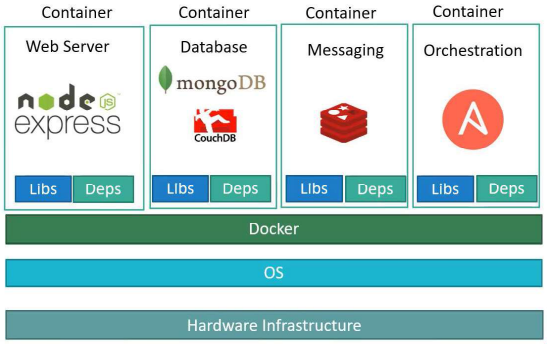
1. Accéder au répertoire de dépôt local	cd my_repertoire
1. Mettre à jour toutes les branches de suivi à distance	git **pull**
2. Basculer vers la branche existante sur laquelle vous souhaitez travailler	git **checkout** feature-a
3. Modifier le fichier file1.md ou tout autre fichier	
4. Mettre en Scène le Fichier Modifié	git **add** file1.md
5. Effectuer un commit pour enregistrer les modifications avec un message descriptif	git **commit -m** "edit file1"
6. Pousser les changements vers la branche correspondante sur GitHub	git **push**

3.2. Conteneur Docker

– **Docker** est la société qui a justement lancé **la technologie des conteneurs** en 2013, et qui domine le marché aujourd'hui.

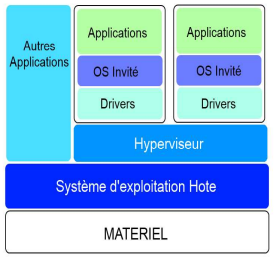
- **Pour les Humains :** les conteneurs sont comme des boîtes standardisées utilisées pour emballer plusieurs articles ensemble en vue d'une expédition.
  Tout comme un colis contient différents articles prêts à être expédiés
- **Pour les Dev / Ops :** les conteneurs sont des outils standardisés pour encapsuler une application entière avec toutes ses configurations et dépendances dans une seule entité logique.
  - Les conteneurs sont des unités exécutables de logiciels qui regroupent le code de l'application ainsi que ses bibliothèques et dépendances.
  - Ces conteneurs sont légers, portables et peuvent être exécutés sur n'importe quel serveur qui prend en charge Docker.
- **Pour les Machines :** est un processus isolé extrait d'archives tar et ancré dans des namespaces, qui définissent son environnement d'exécution, incluant les ressources et les processus visibles.

– **Docker** initialement créé pour fonctionner avec la plateforme Linux, fonctionne désormais avec d'autres OS tels que Windows ou Apple macOS.
– Chaque container exécuté partage les services du système d'exploitation.



3.2.1. Les machines virtuelles (MV) et les containers

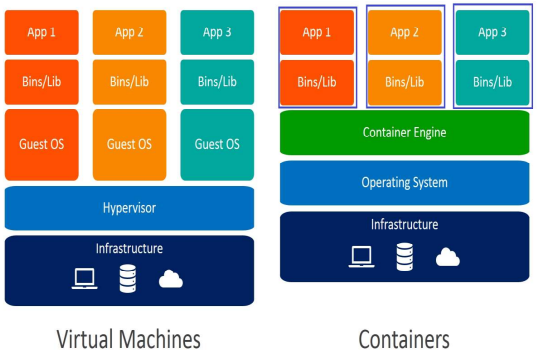
– La **virtualisation hardware** consiste à faire fonctionner sur une même machine physique, plusieurs systèmes comme s'ils fonctionnaient sur des machines physiques distinctes.



  - Une **machine virtuelle (MV)** est une représentation virtuelle ou une émulation d'un ordinateur physique.
  - Une **MV** va donc exécuter son propre OS qui va pouvoir accéder aux ressources de la machine physique qui l'héberge (CPU, mémoire RAM, etc).
  - Un **hyperviseur**, une petite couche logicielle, alloue des ressources informatiques physiques (par ex. des processeurs, de la mémoire, du stockage) à chaque VM.
  - L'hyperviseur sépare chaque machine virtuelle des autres pour éviter toute interférence entre elles.
  - **VMware** a été l'un des premiers à développer et à commercialiser une technologie de virtualisation basée sur des hyperviseurs.
– Les hyperviseurs de MV reposent sur une émulation du hardware, et **requièrent donc beaucoup de puissance de calcul**.
  Pour remédier à ce problème, de nombreuses architectures se tournent vers les **containers**, et par extension vers **Docker**.

– **Les containers** sont proches des MV, mais présentent un avantage important.

- **Les containers se partagent le même noyau de système d'exploitation** et isolent les processus de l'application du reste du système.
- Le conteneur fournit une virtualisation au niveau du système d'exploitation (**virtualisation software**) tandis que la machine virtuelle, quant à elle, fournit une virtualisation au niveau du matériel (**virtualisation hardware**).



– Il est donc nettement **plus efficient** qu'un hyperviseur **en termes de consommation des ressources système**.
– Il est possible d'exécuter près de 4 à 6 fois plus d'instances d'applications avec un container qu'avec des machines virtuelles sur le même hardware.

- Une **MV** s'apparente à un système d'exploitation complet, d'une taille de plusieurs gigaoctets, permettant le partitionnement des ressources d'une infrastructure.
- Un **conteneur** délivre uniquement les ressources nécessaires à une application.
- Le conteneur **partage le kernel de son OS** avec d'autres conteneurs. C'est une différence avec une MV, utilisant un hyperviseur pour distribuer les ressources hardware.
- Cette méthode permet de réduire l'empreinte des applications sur l'infrastructure.
- Le conteneur regroupe tous les composants système nécessaires à l'exécution du code, sans pour autant peser aussi lourd d'un OS complet.

- Un **conteneur**, c'est finalement un système d'exploitation minimaliste et un package logiciel qui contient :
  - Les services (applications, BDD, serveur web)
  - Les dépendances de ces services (librairie, code source, fichiers de configuration - ensemble des ressources externes nécessaires à l'exécution du code.)

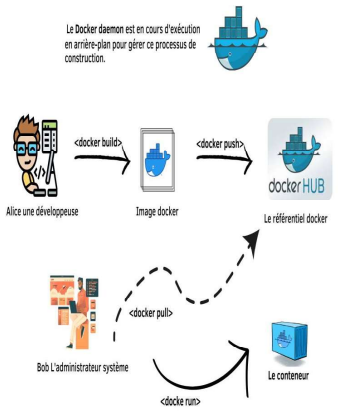
  Tout en les isolant les uns des autres sur un même serveur physique.
  On nomme cet ensemble une **image**.

3.2.2. Docker : quelles sont les composants ?

- **Docker** est une technologie de **conteneurisation** qui facilite la gestion de dépendances au sein d'un projet et ce, à tous les niveaux (développement et déploiement).
  - Disponible sur Linux, Windows et Mac OS,
  - Le mécanisme de Docker se centre autour des **conteneurs** et de leur orchestration, et c'est en cela que la **conteneurisation** se différencie de la **virtualisation**.

- Docker est un système d'exploitation pour conteneurs.
- Docker est installé sur chaque serveur et offre des commandes simples pour concevoir, démarrer ou arrêter des conteneurs.
- Les conteneurs démarrent rapidement (en quelques secondes) car ils n'ont pas besoin de charger un système d'exploitation complet. Ils utilisent moins de ressources, car ils partagent le noyau de l'hôte.
- Les VMs démarrent plus lentement (en minutes) en raison du chargement d'un système d'exploitation complet. Elles consomment plus de ressources, car chaque VM nécessite sa propre allocation de RAM, CPU et stockage.

La plateforme Docker repose sur **plusieurs technologies et composants** :

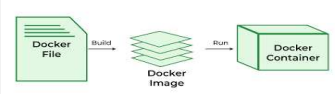


Docker Client	L'interface principale, permettant de communiquer avec le système Docker. Il reçoit les commandes par le biais de l'interface de ligne de commande et les transmet au Docker Daemon.
Docker Hub	Registre officiel de Docker Il s'agit d'un répertoire SaaS permettant de gérer et de partager les conteneurs.
Docker Engine	L'application à installer sur la machine hôte pour créer, exécuter et gérer des conteneurs Docker.
Docker Daemon	Traite les requêtes API afin de gérer les différents aspects de l'installation tels que les images, les conteneurs ou les volumes de stockage
Containers	Un conteneur Docker **est une instance d'image Docker** exécutée sur un microservice individuel ou un stack d'application complet.
Docker Registry	système de catalogage permettant l'hébergement et le "push and pull" des images Docker.

1. Dockerfile

- Chaque conteneur Docker débute avec un " **Dockerfile** ".
- Il s'agit d'un **fichier texte** rédigé dans une syntaxe compréhensible, comportant les instructions de création d'une image Docker.
- Un Dockerfile **précise le système d'exploitation** sur lequel sera basé le conteneur, et les langages, variables environnementales, emplacements de fichiers, ports réseaux et autres composants requis.
- Le **Dockerfile** permet de créer une image. Cette **image** contient la liste des instructions qu'un **conteneur** devra exécuter lorsqu'il sera créé à partir de cette même image.
- Dans le Dockerfile, chaque instruction correspond à une couche :

```
# syntax=docker/dockerfile1
FROM ubuntu :18.04
COPY . /app
RUN make /app
CMD python /app/app.py
```



FROM	crée une couche à partir de l'image Docker ubuntu:18.04.
COPY	ajoute des fichiers à partir du répertoire courant de votre client Docker
RUN	crée l'application avec make
CMD	indique la commande à exécuter dans le conteneur

Les commandes de Dockerfile :

FROM	la commande qui sert à définir l'image de base de l'application
LABEL	permet d'ajouter des métadonnées à l'image
RUN	sert à exécuter des instructions dans le conteneur
ADD	pour ajouter des fichiers dans l'image
WORKDIR	afin de spécifier le répertoire dans lequel sera exécuté tout l'ensemble des instructions
EXPOSE (facultatif)	sert à spécifier le port que l'on souhaite utiliser
VOLUME (facultatif)	pour spécifier le répertoire à partager avec l'hôte
CMD	sert à indiquer au conteneur la commande à exécuter lors de son lancement

```
FROM node:6.10.3

# Create app directory
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app

# Install app dependencies
COPY package.json /usr/src/app/
RUN npm install

# Bundle app source
COPY . /usr/src/app

EXPOSE 9000
CMD [ "npm", "start" ]
```


2. Docker Compose

- Outil puissant qui facilite la gestion des applications complexes composées de plusieurs conteneurs, en utilisant un fichier de configuration **YAML**, pour définir les services, leurs configurations et leurs dépendances de manière claire et structurée.
- Généralement inclus avec l'installation de Docker Desktop

- **Concepts Clés** :

docker-compose.yml	fichier de configuration, tous les services, leurs images, les volumes, les réseaux et d'autres options requises pour l'application
Services	Chaque service représente un conteneur dans l'application. Par exemple, une application web, une base de données, etc
Volumes	servent à stocker les informations entre les redémarrages des conteneurs
Réseaux	Permettent la communication entre les services

– Commandes de Docker Compose

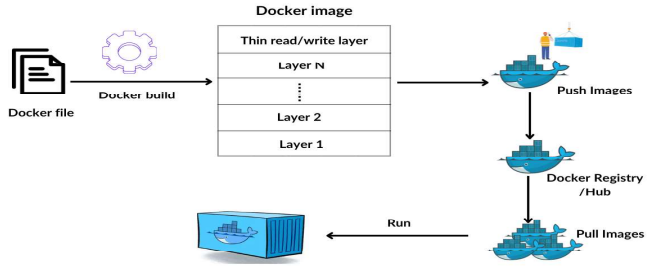
docker compose up	Crée et démarre les services définis dans le fichier compose.yml l'option -d pour exécuter en mode détaché
docker compose down	Arrêter les services
docker compose ps	Afficher l'état des services
docker compose build	Construire ou reconstruire les images
docker compose exec web bash	Exécuter une commande dans un service
docker compose ps service_name	Detail du service
docker init	Initialise le docker dans l'application

- Avantages de Docker Compose
  - **Simplicité** : Facilité de définition d'applications multi-conteneurs dans un seul fichier.
  - **Isolation** : Chaque service fonctionne dans son propre conteneur, ce qui facilite la gestion des dépendances.
  - **Facilité de déploiement** : Commandes simples pour démarrer et arrêter l'application.

3. Docker Image

- Un modèle en lecture seule, utiliser pour créer des conteneurs Docker.
- L'image est composée de plusieurs couches empaquetant toutes les installations, dépendances, bibliothèques, processus et codes d'application nécessaires pour un environnement de conteneur pleinement opérationnel.

- Après avoir écrit le Dockerfile, on invoque **l'utilitaire " build "** pour créer une image basée sur ce fichier.
  - Cette image se présente comme un fichier portable indiquant quels composants logiciels le conteneur exécutera et de quelle façon.



– **Commandes de docker :**

Action	Commande
Version de Docker installé	docker --version
Création d'une image à partir du Dockerfile	docker build -t image_name path_to_dockerfile #Ex. docker build -t myapp .
Liste toutes les images présentes sur le système	docker images
Télécharge une image à partir du Docker Hub	docker pull image_name:tag #Ex. docker pull nginx:latest
Pour supprimer une image	docker rmi ID_de_l'image
Afficher les logs d'un conteneur avec leurs détails	docker logs --details
Pour etiqueter une image	docker tag myapp:latest myapp:v1
Pour pousser une image vers Docker Hub	docker push myapp:v1
Pour inspecter le détail de l'image	docker image inspect myapp:v1
Pour sauvegarder une image en tar archive	docker save -o myapp.tar myapp:v1
Charge une image à partir de tar archive	docker load -i image_name.tar
Élagage d'une image inutilisée Élagage des images inutilisées	docker image prune docker image prune -a

4. Docker Conteneur

- Un conteneur Docker est une instance exécutable d'une image Docker.
- Il encapsule une application, ses dépendances, ses bibliothèques et son environnement d'exécution dans un espace isolé, léger et portable.
- Cycle de vie d'un conteneur
  1. Création : à partir d'une image (docker run ou docker create)
  2. Démarrage : docker start
  3. Exécution : l'application tourne
  4. Arrêt : docker stop
  5. Suppression : docker rm

– **Commandes du docker conteneur**

Action	Commande
Exécuter un conteneur à partir d'une image	docker run container name image name Ex. docker run myapp
Exécuter un conteneur nommé à partir d'une image	docker run –name container_name image_name:tag Ex. docker run –name my_container –d myapp:v1 - d : démarre le conteneur en mode détaché (en arrière-plan).
Exécuter un conteneur en mode interactif	docker run –it my_container
Exécuter un conteneur en mode shell interactif	docker run –it my_container sh
Lister les conteneurs actifs	docker ps
Lister tous les conteneurs	docker ps -a
Arrêter un conteneur	docker stop my_container
Redémarrer	docker restart my_container
Voir les logs	docker logs my_container
Exécuter une commande à l'intérieur	docker exec -it my_container /bin/bash
Supprimer un conteneur	docker rm my_container
Supprimer un conteneur en execution	docker rm -f my_container

Exemple 1 :

```
# 1. Construire une image (voir Dockerfile)
docker build -t mon-app .

# 2. Lancer un conteneur à partir de cette image
docker run -d -p 5000:5000 --name app-instance mon-app

# 3. Vérifier qu'il tourne
docker ps

# 4. Accéder à l'application → http://localhost:5000

# 5. Nettoyer
docker stop app-instance
docker rm app-instance
```

Exemple 2: Création d'un conteneur pour une application web simple en utilisant Flask, un Framework Python.

1. Création d'un répertoire pour le projet **: python-app**
   ```
   cd python-app
   ```
2. Création d'un fichier nommé **app.py** avec le contenu suivant :
   ```
   from flask import Flask

   app = Flask(__name__)

   @app.route('/')
   def hello():
       return 'Hello, Docker!'

   if __name__ == '__main__':
       app.run(host='0.0.0.0', port=5000)
   ```
3. Création du fichier **requirements.txt** avec le contenu suivant :
   ```
   Flask
   ```


4. Création du fichier nommé **Dockerfile** dans le répertoire **python-app :**
   ```
   # Étape 1 : Utiliser une image de base
   FROM python:3.9-slim

   # Étape 2 : Définir le répertoire de travail
   WORKDIR /app

   # Étape 3 : Copier le fichier requirements.txt et installer les dépendances
   COPY requirements.txt ./
   RUN pip install --no-cache-dir -r requirements.txt

   # Étape 4 : Copier le reste des fichiers de l'application
   COPY app.py ./

   # Étape 5 : Exposer le port 5000
   EXPOSE 5000

   # Étape 6 : Commande pour démarrer l'application
   CMD ["python", "app.py"]
   ```
5. Construction de l'image à partir du Dockerfile :
   docker build -t python-app .
   - **-t** permet de spécifier le nom que l'on donne à l'image
   - "**.**" à la fin de la commande désigne l'emplacement de la source de l'image. Cet emplacement vient de l'instruction que vous avez écrite sur la commande COPY du dockerfile "."

   Cela crée une image Docker nommée python-app.

6. Exécution du Conteneur
   docker run -d -p 5000:5000 python-app
   Cela exécute le conteneur en arrière-plan et mappe le port 5000 du conteneur au port 5000 de votre machine.

7. Accès à l'application : http://localhost:5000.

5. Docker et Volume

- Volumes : Mécanisme de stockage persistant dans Docker.
- Contrairement au système de fichiers du conteneur, les volumes survivent au redémarrage ou à la suppression du conteneur.
  Exemple : docker run -v mon_volume:/chemin/dans/conteneur ...

Action	Commande
Création un volume nommé	docker volume create volume_name Ex. docker volume create my_volume
Lister tous les volumes	docker volume ls
Détail du volume	docker volume inspect volume_name
Supprimer un volume	docker volume rm volume_name
Lancer un conteneur à partir du volume	docker run –name container_name –v volume_name:/path/in/container image_name:tag
Lancer un conteneur avec un port	docker run –name container_name –p host_port:container_port image_name ex. docker run --name my_container -p 8080:80 myapp
Copier des fichiers entre container et volume	docker cp local_file container_name:/path/in/container ex. docker cp data.txt my_container:/app/data
Lister tous les reseaux	docker network ls