

## Cours DevOps - Partie 1 : Architecture & Fondations

---

### Accueil et Présentation

---

#### Faire connaissance

- Qu'avez-vous déjà fait avec Docker ?
- Qui suis-je ?
- Qu'avez-vous fait en termes de mises en production ? Quels ont été vos pires soucis/bugs en production ?

#### Objectifs de la formation

Voilà une vue d'ensemble des notions que l'on va apprendre ensemble.

1. **Maîtriser l'architecture DevOps** : comprendre comment les pièces s'assemblent
2. **Savoir superviser une architecture DevOps** : monitoring, logs, métriques
3. **Être opérationnel** : repartir avec des compétences directement applicables

#### Instant Notions

- Paradigmes de programmation
- SOLID : SRP (Single Responsibility Principle), DRY (Don't Repeat Yourself)

#### Évaluations

Lors de notre cours vous aurez 3 notes :

- Un QCM pour évaluer la compréhension des notions de bases du cours
- Un TD de projet individuel
- Un TP de projet de groupe

#### Méthode pédagogique

- **30% théorie, 70% pratique** : on code ensemble !
- **Participation active** : "Coupez"-moi la parole 🙋, pas de monologue d'enseignant, le but est que l'on discute et construise ensemble !
- **Projets concrets** :
  - Pour commencer soft, une API de Todo App qui applique l'essentiel du devops
  - Nous continuerons avec la création d'une API Jurassic Park, approfondissant le devops
- **Découpages des chapitres** : chaque partie est découpée en deux : les "Core Notions", et les "Allons plus loin"

# Quelles bonnes pratiques appliquez-vous ?

En voilà quelques unes essentielles :

Il n'y a pas de mauvaise question (PAS DU TOUT)

- Les moteurs de recherches sont nos amis
  - L'anglais 🇬🇧
  - L'IA, quelle est votre usage ?
  - La pratique est cruciale !
  - Collaborer donne des ailes
  - Faire de la veille technologique, tout change tout le temps (et YouTube c'est sympa 📺)
- Dès ce premier cours nous allons mettre en pratique ces réflexes d'auto-apprentissage (nécessaires tout au long de la carrière).

---

## Planning de la Journée

- Étape 1 Introduction DevOps et Architecture
- Étape 2 Installation des outils
- Étape 3 Agile/Scrum et besoins DevOps
- Étape 4 Docker Avancé et Orchestration
- Étape 5 TP Solo - API Todo App + Dockerisation
- Étape 6 Git Workflows et Gestion de Projet

---

## 1 - Introduction DevOps et Architecture

### CORE NOTIONS

#### Qu'est-ce que le DevOps ?

Le DevOps, c'est faire collaborer les développeurs (Dev) et les opérationnels (Ops) pour livrer du code plus vite et plus sûrement.

Quel est le problème quand Dev et Ops ne se parlent pas ?

##### 1. Définir et configurer une architecture DevOps

Une architecture DevOps est un ensemble d'outils et de pratiques qui permettent aux équipes de développement et d'opérations de collaborer efficacement.

Avec Docker, on peut créer des conteneurs pour nos applications, ce qui facilite leur déploiement et leur gestion. Nous creuserons cela un peu plus tard.

##### 2. Automatisation du cycle de vie d'une application

Grâce au CI / CD (Intégration Continue / Déploiement Continu) sur GitHub, vous pouvez automatiser le processus de test et de déploiement de votre application.

Cela signifie que chaque fois que vous apportez des modifications, des tests unitaires sont exécutés automatiquement pour s'assurer que tout fonctionne avant de déployer les changements !

### 3. Outils, techniques

Nous les observerons au fil de ce cours.

### 4. Supervision et Performance des Applications

Il est essentiel de surveiller nos applications et notre infrastructure pour garantir leur disponibilité, leurs performances, éviter les imprévus.

Des outils de monitoring peuvent nous alerter en cas de problème.

Sur le dernier projet nous pourrons expérimenter cela.

## Les 3 piliers du DevOps

### 1. Culture

- Collaboration entre équipes
- Responsabilité partagée
- Feedback continu

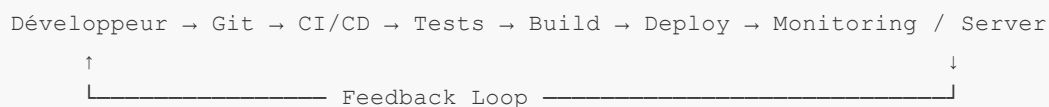
### 2. Automation

- Tests automatisés
- Déploiement automatique
- Infrastructure as Code

### 3. Measurement

- Métriques de performance
- Monitoring applicatif
- Alerting intelligent

## Architecture DevOps type



## Le cycle DevOps

1. **Plan** : Définir les features
2. **Code** : Développer
3. **Build** : Compiler/packager
4. **Test** : Valider automatiquement
5. **Release** : Préparer la mise en production
6. **Deploy** : Mettre en production
7. **Operate** : Maintenir en conditions opérationnelles
8. **Monitor** : Observer et mesurer

## 2 - Partie 1 : Installation des Outils

---

### CORE NOTIONS

#### Les prérequis

**Sondage rapide :** Qui a déjà installé Docker ? Node.js ? Git ? Quelle est votre stack principale / vos stacks acquises ?

#### Introduction à Docker

On ne le nomme plus : [Fireship](#) ! Si vous voulez démystifier un sujet rapidement allez sur sa chaîne. Pour le cours du jour : [Docker in 100 Seconds](#)

De nos jours, le développement d'applications ne se limite pas à l'écriture de code. La multiplicité des langages, des cadres, des architectures et des interfaces discontinues entre les outils pour chaque étape du cycle de vie crée une énorme complexité.

👉 Docker simplifie et accélère votre flux de travail, tout en donnant aux développeurs la liberté d'innover en choisissant leurs outils, leurs piles d'applications et leurs environnements de déploiement pour chaque projet.

Les conteneurs sont une unité logicielle standardisée qui permet aux développeurs d'isoler leur application de son environnement, ce qui résout le problème du "ça marche sur ma machine".

Pourquoi docker ? -> [Why Docker](#) | [Docker](#)

Très bonne ressource à suivre : [Docker Simplified: A Hands-On Guide for Absolute Beginners](#)

#### Pourquoi utiliser Docker en tant que développeur ?

Avantages de Docker :

- Rapidité: Démarrage et arrêt rapides de l'application.
- Multiplateforme: Fonctionne sur n'importe quel système.
- Construction et destruction rapides des conteneurs.
- Configuration facile: Évite les problèmes d'installation manuelle des dépendances.
- Environnements isolés: Maintient la propreté de l'espace de travail.
- Déploiement facilité: Simplifie la mise en ligne du projet sur le serveur

Le but principal est de pouvoir reproduire les environnements : faire un Dockerfile personnalisé, pour que n'importe qui le réutilise pour build son environnement grâce à cet immuable snapshot. On met cette image en ligne pour que n'importe qui puisse la récupérer et générer son propre container.

Ensuite, si le projet doit être une infrastructure composée de plusieurs applications, on pourra toutes les conteneuriser et les orchestrer pour qu'elles cohabitent dans un même environnement, malgré leurs différences.

C'est faisable en utilisant Docker Compose, mais on a également des outils tels que Kubernetes pour faire ce travail. Kubernetes simplifie la gestion des applications, améliore la disponibilité et facilite la mise à l'échelle, le tout grâce à une automatisation intelligente.

Imaginez **Kubernetes** comme un chef d'orchestre pour vos applications. Vous avez plusieurs applications emballées dans des conteneurs, et Kubernetes s'occupe de les déployer, de les mettre à l'échelle (ajuster automatiquement le nombre de copies en fonction de la demande), et de s'assurer qu'elles fonctionnent correctement. Il gère également la répartition du trafic entre les différentes parties de votre application, garantissant une disponibilité continue.

**Docker Swarm** est une autre technologie liée à la gestion de conteneurs, mais elle se concentre spécifiquement sur l'orchestration de conteneurs Docker. Contrairement à Kubernetes, qui est plus complexe et offre une gamme plus étendue de fonctionnalités, Docker Swarm est une solution d'orchestration plus légère et plus simple, développée directement par Docker.

Nous verrons cela plus tard.

Pour l'instant, comprenons-mieux comment s'articule un environnement Docker seul.

## Docker : Les trois termes les plus importants !

### Dockerfile :

Script qui décrit les étapes de construction d'une image Docker.

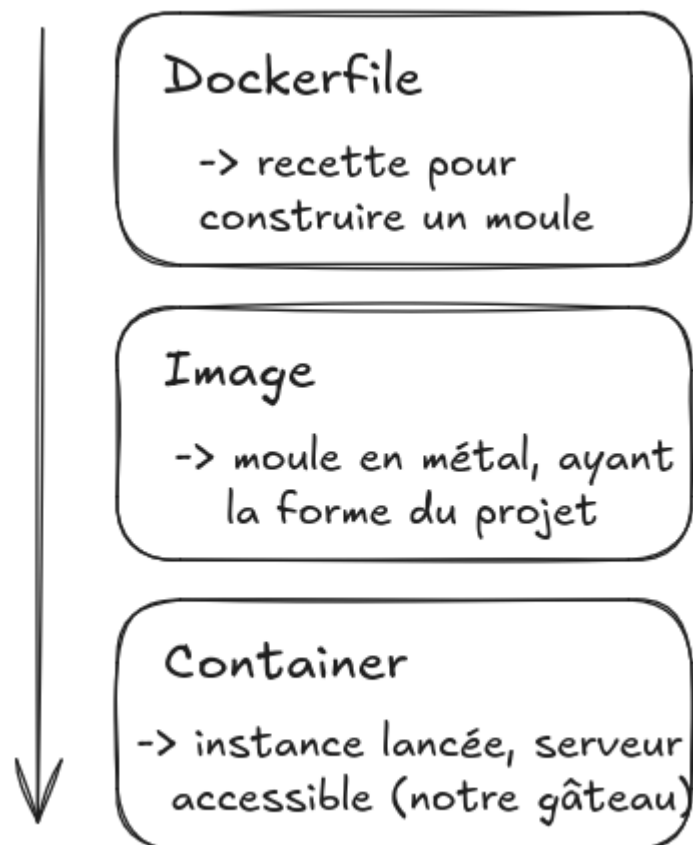
### Image :

Snapshot (ou instantané de stockage, permet de réaliser une copie de données stockées sur un système de stockage, ou une copie des modifications apportées à ces données) immuable et exécutable d'une application et de son environnement. (**template for running containers**)

### Container :

Instance en cours d'exécution d'une image Docker, isolée et portable. (running process)

## Visuellement : Dockerfile, Image, Container





Mais j'ai pas compris, c'est quoi une image ?

C'est comme un modèle ou un "plan" qui contient tout ce qu'il faut pour faire fonctionner une application : le code, les bibliothèques, les dépendances et les configurations nécessaires

**Une image est en lecture seule et ne change pas** : elle sert simplement de base.

À partir de cette image, Docker peut créer un conteneur (une instance en cours d'exécution de l'image), c'est-à-dire un environnement isolé où l'application tourne vraiment

Cela permet de garantir que l'application marchera toujours pareil, peu importe où on la lance  
On crée souvent une image à partir d'un fichier spécial appelé Dockerfile, qui décrit étape par étape comment construire l'image

## Installation guidée

### 1. Docker Desktop (+ Docker terminal)

#### Vérification de la présence de Docker

```
# Vérification
docker --version
docker compose version

# Test rapide
docker run hello-world
```

#### Pour installer :

Aller sur le site officiel de Docker : <https://www.docker.com/get-started>

Télécharger selon votre OS.

Suivre les instructions d'installations spécifique à l'OS.

Docker sera présent en version Desktop et dans le terminal.

**"Hello world ne marche pas chez moi"**

Si certaines personnes ont un message contenant 401 unauthorized, c'est que la requête fonctionne, mais que l'accès est non autorisé. Il faut se connecter :

Sur terminal (Git Bash ou Powershell) :

```
docker login -u <mon_username>
```

### 1. Installez Docker sur VSCode

Cela permet d'avoir un support langage, pendant qu'on rédige nos fichiers docker. Pratique pour avoir les bonnes couleurs et de l'aide.

Nous pouvons également faire un lien vers des remote registries.

### 2. Node.js (v18+) (normalement vous l'avez !)

```
# Via nvm (recommandé)
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.0/install.sh |
bash
nvm install 18
nvm use 18

# Vérification
node --version
npm --version
```

### 3. Git & GitLab / Github

```
# Configuration
git config --global user.name "Votre Nom"
git config --global user.email "email@example.com"
```

### 4. VS Code + Extensions

- Docker
- GitLens
- ESLint
- REST Client

## Environnement de secours

S'il y a trop de problèmes pour l'installation locale, voici une alternative :

**GitHub Codespaces** comme backup :

1. Créer un compte GitHub si nécessaire
2. Utiliser un repo test, fait par vos soins
3. Lancer un Codespace

# ALLONS PLUS LOIN

## WSL2 pour Windows

Pour les personnes sur Windows je vous invite à faire une recherche sur WSL2.  
Pourquoi est-ce mieux pour Docker ?

Notons la configuration, et l'intégration VS Code

## Alternatives aux outils

Instant Veille outils :

Connaissez-vous ces outils ?

- [Podman](#) vs Docker
- Yarn/pnpm vs npm
- GitLab vs GitHub vs Bitbucket

---

## 2 - Partie 2 : API Node.JS Dockerisée

Pour s'échauffer, et intégrer progressivement les notions, l'objectif premier est de faire un projet backend Node.js.

Puis nous pourons le conteneuriser.

Grâce à docker le projet fonctionnera sur notre machine, mais sera aussi reproductible dans un autre environnement (autre ordinateur, un serveur, etc.) à l'identique.

=> Suivez [les étapes indiquées dans mon repository](#) pour savoir comment Dockeriser une app basique en Node.js

Nous nous chargerons **après** du projet Todo App.

## Instant Cheat Sheet ! => Terminal cheat sheet

Pour les personnes qui aiment bien creuser le terminal

Voilà quelques commandes utiles

Comment naviguer dans le système de fichiers ?

- `pwd` : savoir où on se trouve
- `cd chemin_du_dossier` : aller dans un dossier (et `cd ..` pour revenir au dossier parent)
- `ls`, ou `ls -l`, ou `ls -la` : lister les éléments

Commandes de manipulation de fichiers :

- `mkdir nom_de_dossier` : créer un dossier
- `touch nom_du_fichier.txt` : créer un fichier
- `cat nom_du_fichier` : afficher le contenu du fichier

Commandes très pratiques :

- ajouter `--help` à la suite de la commande sur laquelle on se pose des questions



- écrire `man le_nom_de_ma_commande` pour en savoir plus sur la commande (man veut dire manual)

On peut éditer nos fichiers directement dans le terminal ?

-> Oui, carrément : des outils d'éditeurs de texte comme [nano](#) et [vim](#) permettent de manipuler nos fichiers. Je conseille vim, que j'utilise beaucoup ! (mais c'est une question d'habitude)

## Définition. C'est quoi un Dockerfile ?

👉 le Dockerfile est un script texte qui contient une série d'instructions permettant de construire une image Docker.

Il spécifie

- l'environnement de travail,
- les dépendances,
- les étapes nécessaires à la configuration d'une application à l'intérieur d'un conteneur Docker.
- ex: sélection d'une image de base, définition du répertoire de travail, copie des fichiers du projet, l'installation des dépendances, etc.

Mise en pratique

À la racine du projet Node.js, dans un fichier nommé Dockerfile, écrire [le code suivant](#).

## Les Images, plus en détail

Définition : l'image est un package léger et autonome qui contient tout le nécessaire pour exécuter une application, y compris :

le code,

- les bibliothèques,
- les dépendances,
- les variables d'environnement,
- les fichiers de configuration

Les images sont construites à partir d'un ensemble d'instructions définies dans un Dockerfile. Elles sont immuables, ce qui signifie qu'une fois créées, elles ne sont pas modifiées. Les conteneurs Docker sont ensuite créés à partir de ces images et représentent l'instance en cours d'exécution de l'application.

Pour créer une image :

```
docker build -t NomDeMonImage source/de/fichiers

# Exemple:
docker build -t NomDeMonImage .

# Bonnes pratiques:
docker build -t NomDeMonImage mon_pseudo/nom-de-mon-projet:1.0 .
```

Erreur courante : l'oubli du `.` à la fin de la commande

- l'usage de `-t` permet de donner un nametag à l'image

- un bon réflexe de donner mon pseudo suivi du nom de l'app
- on remarque qu'il suit les étapes de mon Dockerfile

## Comment observer ce qui se passe ?!

### Côté Docker Desktop

L'application est relativement explicite : sur notre gauche on constate les onglets pour voir les Conteneurs, les Images, les Volumes (que nous verrons plus tard), etc.

### Côté Terminal

Pour obtenir l'image du docker qu'on vient de créer, faire la commande `docker images`

On voit une liste d'images, dont celle qu'on vient tout juste de créer

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
onyj/projet-cocker	1.0	d4ef76a0eac7	X minutes ago	918MB

Maintenant il suffit de copier le code présent en colonne "IMAGE ID".

Si on fait `docker run id_de_mon_image_docker` (pour moi, d4ef76a0eac7) on voit qu'il lance localhost:8080 . Le problème : quand on le lance depuis le navigateur, ce n'est pas accessible ! On expose le 8080 depuis le docker mais c'est pas accessible depuis le monde extérieur.

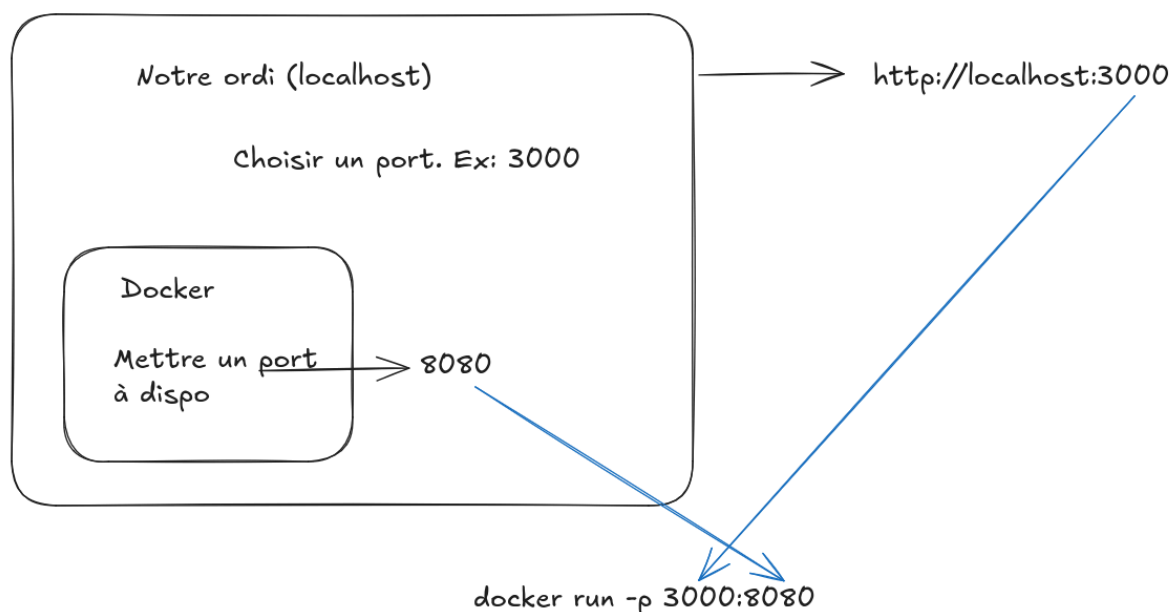
Pour obtenir les instances docker lancées, exécuter la commande `docker ps`

## Exposer les bons ports

### Attends, je dois exposer 2 ports différents ?

Docker nous permet de lancer un environnement virtuel, isolé au sein de notre ordi.

Pour qu'il soit accessible au port local de notre ordi, on va exposer ce fameux environnement virtuel sur un port pour docker



**Donc la bonne méthode, c'est de lancer :**

```
docker run -p 5000:8080 d4ef76a0eac7
```

- 5000 : c'est le port sur notre machine
- 8080 : c'est le port du docker
- le code juste après : l'id de notre image, trouvée grâce à la commande `docker images` lancée dans le terminal
- maintenant ouvrir localhost:5000 sur navigateur, et cela devrait fonctionner !

### Comment désactiver les ports ?

Vous l'aurez peut-être constaté, si on ferme la page, le projet continue de tourner.

Il faut donc :

- soit aller dans Docker Desktop pour désactiver les ports
- soit stopper le processus depuis le terminal

```
# Lister l'activité Docker
docker ps

# Stopper
docker stop NOM_DU_CONTENEUR
```

---

## 3 - Agile/Scrum et Besoins DevOps

Parfait, nous avons de bonnes bases concernant Docker.

Avant d'apprendre davantage de notions DevOps, voyons comment s'organise la gestion d'un projet.

En effet, la méthodologie Agile/Scrum sera intimement liée à la partie DevOps d'un projet ou d'une entreprise.

## CORE NOTIONS

### Pourquoi Agile nécessite le DevOps

#### Les cérémonies Agile et leur impact DevOps

##### 1. Sprint Planning

- Définir les features testables
- Prévoir l'automatisation

Séance où l'équipe définit les fonctionnalités à réaliser pendant le sprint, en privilégiant des tâches testables et automatisables. Cela favorise la planification des tests automatiques et améliore la prédictibilité des livraisons, essentiels pour DevOps.

[Découvrir les cérémonies Agile sur Asana.](#)

##### 2. Daily Standup

- Blocages techniques
- État du pipeline CI/CD

Réunion quotidienne de 15 min pour partager l'avancement, soulever les blocages techniques, et échanger sur l'état du pipeline CI/CD. Cela garantit que les problèmes sont détectés tôt, accélérant ainsi le flux de changement et le feedback, un pilier DevOps.

Le daily standup s'articule en 3 points principaux : chacun doit dire :

- "Ce que j'ai fait la veille"
- "Ce que je vais faire aujourd'hui"
- "Mes points de blocages"

### 3. **Sprint Review**

- Demo automatisée
- Métriques de performance

Présentation du travail réalisé (parfois via des démos automatisées) devant les parties prenantes, avec discussion des métriques de performance. Ce moment permet de valider la qualité logicielle et l'intégration continue dans la chaîne de livraison.

### 4. **Retrospective**

- Amélioration continue du pipeline
- Réduction du lead time

Moment clé d'amélioration continue où l'équipe identifie des axes pour perfectionner ses pratiques, réduire le lead time, et fiabiliser le pipeline. C'est central dans la philosophie DevOps d'évolution itérative des processus.

## **Les bases de la gestion agile**

### **Fondements de la gestion agile :**

- L'approche agile s'appuie sur 4 piliers du Manifeste Agile :
  - Privilégier les individus et leurs interactions plutôt que les processus et outils
  - Produire un logiciel/produit fonctionnel plutôt qu'une documentation exhaustive
  - Collaborer avec le client plutôt que contracter strictement
  - Savoir s'adapter au changement plutôt que suivre un plan figé
- 12 principes détaillent ces valeurs : livraison fréquente de fonctionnalités utiles, adaptation rapide au changement, dialogue constant entre métier et technique, simplicité, équipes auto-organisées, amélioration continue, etc.

### **Les principales pratiques agiles :**

- Développement itératif par « sprints » (courts cycles de travail avec objectifs précis)
- Livraisons fréquentes de versions utilisables (logiciel opérationnel, prototypes...)
- Auto-organisation des équipes : prise d'initiative, responsabilité et transparence du flux de travail
- Feedback constant avec les parties prenantes (démonstrations, rétrospectives...)
- Collaboration renforcée intra/inter-équipes et avec les clients/utilisateurs
- Adaptation continue du plan selon les retours terrain

### **Les frameworks agiles courants :**

- Scrum (sprints, cérémonies, backlog...)
- Kanban (tableau visuel, gestion de flux tiré, limites de travail en cours)
- [Extreme Programming](#) (XP), Lean, Scrumban...

### Kanban dans la gestion technique des projets :

- Kanban utilise un tableau (physique ou numérique) organisé en colonnes représentant les états de traitement des tâches (exemples typiques, adaptés pour la tech) :
  - **Backlog** : stock d'idées/fonctionnalités à développer ou corriger
  - **Todo** : tâches programmées pour le sprint/itération actuelle
  - **Doing** : éléments en cours de développement
  - **Reviewing** : code ou tâches en validation technique (par les pairs, le PO, QA...)
  - **Done** : tâches finalisées
- Les outils Kanban techniques souvent directement liés au code source : par exemple, *GitHub Projects*, Jira Software, etc.
- Chaque carte représente une User Story, une tâche, un bug, etc., et peut être liée à des Pull Requests, à des commits, ou à l'automatisation du pipeline CI/CD.

### Contenu d'un ticket / issue

Et à quoi ressemble l'organisation d'un ticket ?

#### User Stories et Definition of Done

```
## User Story
En tant qu'utilisateur,
Je veux pouvoir créer une tâche,
Afin de ne pas oublier mes actions à faire.

## Definition of Done
- [ ] Code reviewé
- [ ] Tests unitaires passants
- [ ] Tests d'intégration passants
- [ ] Documentation à jour
- [ ] Déployé en staging
- [ ] Monitoring en place
```

### Les Tickets Tech

Il y a plusieurs manières d'organiser la rédaction d'un ticket tech, pour qu'il soit prêt à être exécuté par un profil tech.

```
## Why
Pourquoi faire ce ticket ? / Qu'est-ce qui constitue un problème ?

## How
Quelles étapes ?
- [ ] Faire ceci
- [ ] Faire cela

## Must Have
```

Qu'est-ce que le résultat attendu comporte ?

- [ ] Cliquer ici fait cela
- [ ] Telle chose est visible
- [ ] Les tests unitaires fonctionnent à 90%
- [ ] ...

## ## Reading List

- documentation de telle librairie
- lien vers un guide très proche de notre objectif
- etc...

Faire des tickets les plus petits possibles en termes de fonctionnalités !

Ex : au lieu de "faire un système d'articles blog", -> 1 ticket pour CRUD API, 1 ticket pour page create, etc

Plusieurs compositions de tickets possibles ! Ceci est un exemple. Vous en connaissez d'autres ?

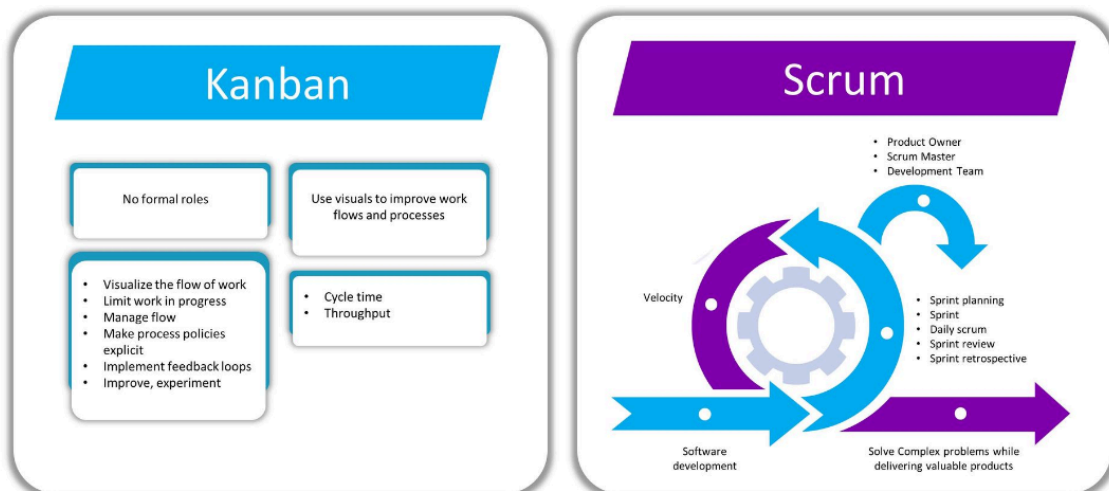
## ALLONS PLUS LOIN

### Kanban vs Scrum

- Flow continu vs sprints
- WIP limits
- Métriques différentes

**Kanban** et **Scrum** sont deux approches Agiles majeures, chacune adaptée à des contextes et besoins différents :

## Scrum and Kanban



Critère	Kanban	Scrum
Cadence du travail	<b>Flux continu</b> : tâches traitées au fil de l'eau, sans cycle fixe	<b>Sprints</b> : cycles définis (souvent 2-4 semaines), chaque sprint ayant un objectif précis

Critère	Kanban	Scrum
Limites WIP	<b>WIP limits</b> imposées explicitement sur chaque colonne pour éviter les surcharges et lisser le flux	WIP par sprint : nombre de tâches fixé au début du sprint, mais pas de limites imposées pour les colonnes
Rôles	Pas de rôles obligatoires	Rôles prescrits : Product Owner, Scrum Master, Équipe de développement
Visualisation du travail	Tableau Kanban (états flexibles : à faire, en cours, etc.), focus sur la visualisation du flux	Tableau Scrum et outils comme le burndown chart pour suivre le sprint
Gestion du changement	Hautement flexible, adaptation continue, tâches ajoutées/supprimées à tout moment	Changement déconseillé pendant un sprint ; ajustement seulement entre deux sprints
Métriques clés	<b>Lead time, cycle time, débit (throughput)</b> : mesure l'efficacité du flux (combien de temps entre début et fin d'une tâche, combien de tâches livrées dans un temps donné)	<b>Vélocité</b> : nombre de points/éléments terminés par sprint, suivi de la planification

## Résumé visuel

### En pratique :

- *Kanban* convient aux équipes avec priorités changeantes ou flux de demandes constant, notamment pour la maintenance ou l'amélioration incrémentale.
- *Scrum* structure l'effort pour des développements itératifs avec objectifs définis, mieux adapté aux projets nécessitant planification stricte et livrables réguliers.

Ressource: [Kanban vs Scrum chez Atlassian](#)

### À retenir :

- **Flow continu** = Kanban : travail tiré, limites WIP, ajustement à la demande, focus sur le flux.
- **Sprints** = Scrum : batch planifié, engagement sur le sprint, cadencement régulier du travail.

### Métriques différentes :

- *Kanban* : lead time, cycle time, throughput.
- *Scrum* : vélocité par sprint, burndown chart

## Autres mots clés intéressants à chercher

- **Schémas** :
  - Cycle Scrum avec points DevOps
  - Value Stream Mapping
  - Métriques DORA

- Outils :
  - Jira automation
  - Azure DevOps boards
  - GitLab issues

---

## 4 - Docker Avancé et Orchestration

---

### CORE NOTIONS

#### Rappel Docker basics

```
# Dockerfile basique
FROM node:18-alpine
WORKDIR /app
COPY package*.json ./
RUN npm ci --only=production
COPY . .
EXPOSE 3000
CMD ["node", "server.js"]
```

#### Docker Compose pour multi-containers

“C’est bien drôle de passer ma vie à taper des commandes dans le terminal, mais il n’y a pas plus rapide ?”

Eh bien si, au lieu de retenir plusieurs commandes complexes par projet, on a un moyen de standardiser les règles dans un fichier nommé `docker-compose.yml`.

Voici [comment créer et configurer `docker-compose.yml`](#).

(Et [voici une autre ressource similaire](#))

Pour un récapitulatif complet sur les notions Docker vues jusqu’à présent, [voilà un tutoriel complet](#).

Un `docker-compose.yml` peut se structurer de la sorte :

```
version: 'X'

services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - ./code
  redis:
    image: redis
```

#### [Synthèse concernant docker-compose](#)

Ci-dessous, un exemple plus complet Node + Postgres + Redis :

---



```

version: '3.8'
services:
  api:
    build: .
    ports:
      - "3000:3000"
    environment:
      - NODE_ENV=production
      - DB_HOST=postgres
    depends_on:
      - postgres
      - redis

  postgres:
    image: postgres:15
    environment:
      POSTGRES_DB: myapp
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
    volumes:
      - postgres_data:/var/lib/postgresql/data

  redis:
    image: redis:7-alpine
    ports:
      - "6379:6379"

volumes:
  postgres_data:

```

L'avantage ultime c'est qu'on n'a plus que 3 commandes principales à retenir :

```
docker compose build
```

Est l'équivalent de la (longue) commande build que l'on faisait dans le terminal

```
docker compose up
```

Équivalent de la commande `docker run...` que l'on faisait

```
docker compose down
```

Permet de stopper le container qui était lancé avec `docker-compose up`

## Best practices Docker

### 1. Multi-stage builds

```

# Build stage
FROM node:18 AS builder
WORKDIR /app

```

```
COPY package*.json ./
RUN npm ci
COPY . .
RUN npm run build

# Production stage
FROM node:18-alpine
WORKDIR /app
COPY --from=builder /app/dist ./dist
COPY --from=builder /app/node_modules ./node_modules
CMD ["node", "dist/server.js"]
```

## 2. Security scanning

```
docker scan myimage:latest
```

## 3. Optimisation des layers

- Regrouper les RUN
- Utiliser .dockerignore
- Mettre le moins changeant en premier

# Les Volumes

Pour les personnes les plus attentives, vous aurez remarqué le mot "volumes" dans le docker-compose.

L'inconvénient des conteneurs docker, c'est qu'ils sont tous isolés les uns des autres, et que si on les ferme, on perd toutes les données dynamiquement calculées.

## Comment rendre les données persistantes et partageables, indépendamment de la vie des conteneurs ?

Un volume est utile pour :

- Garder tout ce qui doit survivre à un redémarrage ou à la suppression d'un conteneur
- Partager des fichiers entre plusieurs conteneurs

## À quoi sert un volume Docker ?

- Persistance des données

Les conteneurs Docker sont éphémères : si on supprime ou recrée un conteneur, tout ce qui est stocké à l'intérieur (dans le système de fichiers du conteneur) est perdu.

👉 Un volume permet de stocker des données de façon persistante, indépendamment du cycle de vie du conteneur.

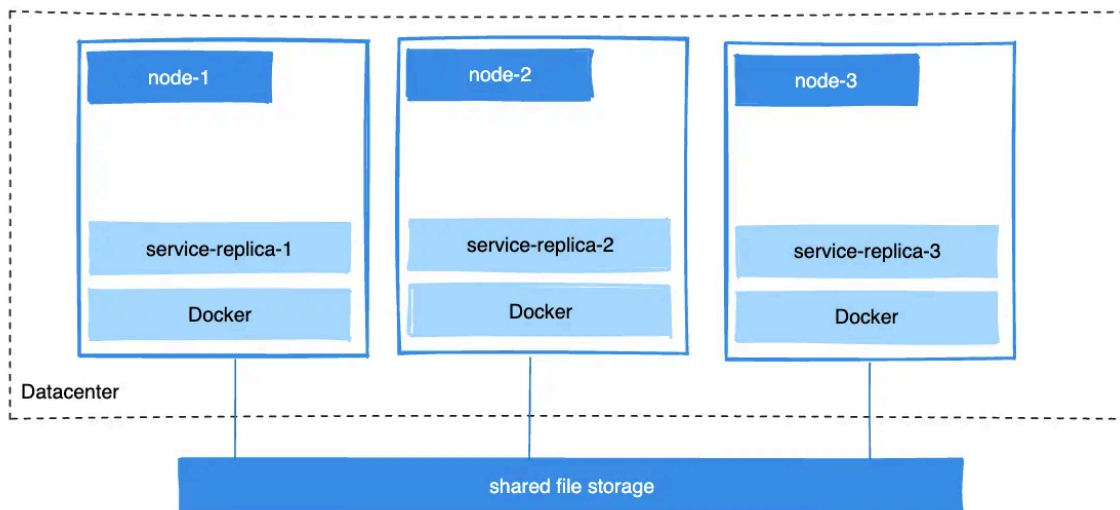
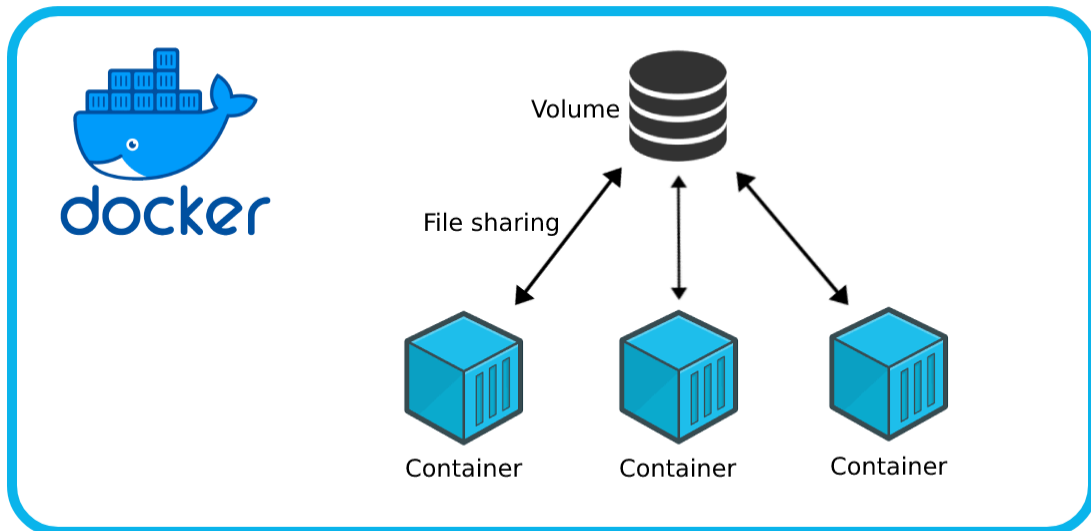
Par exemple, si ton application écrit des fichiers dans /stuff, ces fichiers resteront disponibles même si tu détruis et recrées le conteneur

- Partage de données

Un volume peut être monté dans plusieurs conteneurs en même temps. Cela permet de partager des fichiers ou des données entre plusieurs services ou applications qui tournent dans des conteneurs différents

- Séparation des données et de l'application  
Le volume est stocké sur l'hôte Docker, dans un emplacement géré par Docker (généralement sous `/var/lib/docker/volumes`). Cela sépare clairement les données de l'application, ce qui facilite la sauvegarde, la migration ou la gestion des données sans toucher à l'application elle-même
- Sécurité et portabilité  
Les volumes sont isolés du système de fichiers principal de l'hôte, ce qui réduit les risques de conflits ou de suppression accidentelle. Ils sont aussi plus portables que les montages de type "bind" (qui lient un dossier précis de l'hôte)

## Voilà ce qu'il se passe, visuellement



## Partageons les données grâce aux volumes

Partager les données entre plusieurs containers (persist files).

Créons un volume Docker nommé shared-stuff :

```
docker volume create shared-stuff
```

Lorsque qu'on lance un conteneur avec docker run, utiliser l'option --mount pour lier le volume créé au conteneur. Par exemple :

```
docker run --mount source=shared-stuff, target=/stuff mon_conteneur mon_port
```

[Plus de détails ici](#)

### Que faire de mon volume ensuite ?

- Stocker des données d'application  
Utiliser le volume pour y stocker tout ce que notre application doit conserver : bases de données, fichiers de configuration, fichiers utilisateurs, logs, etc.
- Sauvegarder ou restaurer  
On peut sauvegarder le contenu du volume (par exemple pour des backups réguliers) ou le restaurer sur un autre hôte ou environnement.
- Partager entre plusieurs conteneurs  
Si on a besoin que plusieurs conteneurs accèdent aux mêmes données, on peut monter le même volume dans chacun d'eux.
- Nettoyer ou supprimer  
Si on n'as plus besoin du volume, on peut le supprimer avec docker volume rm shared-stuff (après avoir arrêté et supprimé les conteneurs qui l'utilisent).

## Instant Cheat Sheet! Manipuler le Volume

Concrètement, voilà ce qu'on peut faire sur notre volume (dans un terminal) :

- Voir le contenu d'un volume :

```
docker exec -it mon_conteneur sh
```

ou bash si disponible :

```
docker exec -it mon_conteneur bash
```

afficher le fameux stuff : `ls /stuff`

- Copier un fichier depuis le volume vers notre machine hôte

```
docker cp mon_conteneur:/stuff/nom_du_fichier ./local_destination/
```

- Copier un fichier de notre machine hôte vers le volume

```
docker cp ./mon_fichier mon_conteneur:/stuff
```

- Lister tous les volumes Docker

```
docker volume ls
```

- Inspecter un volume pour voir où il est stocké sur l'hôte

```
docker volume inspect shared-stuff
```

- Supprimer le volume (après avoir supprimé tous les conteneurs qui l'utilisent)

```
docker rm mon_conteneur  
docker volume rm shared-stuff
```

- Sauvegarder le contenu du volume

```
docker run --rm \  
-v shared-stuff:/stuff \  
-v $(pwd) :/backup \  
alpine \  
tar czf /backup/backup-stuff.tar.gz -C /stuff .
```

- Restaurer une sauvegarde dans le volume

```
docker run --rm \  
-v shared-stuff:/stuff \  
-v $(pwd) :/backup \  
alpine \  
tar xzf /backup/backup-stuff.tar.gz -C /stuff
```

---

## 5 - TP Solo - API Tâches + Dockerisation

---

Let's go, réalisons notre **Todo App** (API) Dockerisée !

**Actuellement, nous avons :**

- Une app Node.js très basique
- Une configuration Dockerfile + docker-compose.yml
- Potentiellement un Volume

**Les objectifs sont les suivants :**

### 1. Agile

Appliquer les méthodologies de gestion Agile/Scrum (simplement Kanban, pour aller à l'essentiel) :

- Créer un GitHub Project (kanban)  
Voici [un exemple de Github Project \(Kanban\)](#).
- Rédiger les tickets minimaux pour encadrer les fonctionnalités Todo App à développer  
Deux tickets minimum à présenter dans votre rendu solo, pour les fonctionnalités de la Todo API.

### 2. Développer les fonctionnalités spécifiées dans les tickets

- Les éléments API (Node.js)
- Les éléments Base de données (MySQL ou NoSQL)

Remarques :

- rester simple, l'objet du cours n'est pas le code pur, mais le DevOps.
- Pourquoi faire des fonctionnalités dans une app qui est déjà dockerisée ? On a déjà fini le travail en soit ?  
Eh bien aujourd'hui cela n'est pas utile, mais demain cela prendra tout son sens quand on devra ajouter des automatisations de développement ! La CI/CD.
- Rappel : Le travail produit aujourd'hui et demain fera l'objet de notes individuelles !

### Rendu :

Le projet à démarrer aujourd'hui => Ce repository (**public**) contient :

- Readme,
  - Renseignant toutes les informations relatives au projet : guidelines, comment lancer le projet, etc
  - Référant les éléments de gestion Agile/Scrum / Kanban relatifs au projet Solo
- Kanban : contenant les tickets qui ont permis de faire le projet
- Code source de l'API Node.js : un CRUD permettant la manipulation des `Tasks`
- Base de données : une `Task` comprend `id`, `state`, `description`
- Dockerisation complète du projet : `Dockerfile`, `docker-compose.yml`, bonnes pratiques

Le rendu de demain est un lien vers ce Repository, sur lequel on aura ajouté d'autres fonctionnalités

## CORE NOTIONS

### Énoncé du projet

**Objectif :** Créer une API REST de gestion de tâches avec Node.js, la dockeriser et préparer pour la CI/CD.

### Structure du projet

```

todo-api/
├── src/
│   ├── routes/
│   │   └── tasks.js
│   ├── models/
│   │   └── task.js
│   ├── middleware/
│   │   └── errorHandler.js
│   └── app.js
├── tests/                                # à développer pendant COURS PARTIE 2
│   ├── unit/
│   │   └── task.test.js
│   └── integration/
│       └── api.test.js
├── Dockerfile
├── .dockerignore
├── .gitignore
├── docker-compose.yml
└── package.json
  
```

## Fonctionnalités à implémenter

### 1. CRUD Tâches

- `POST /tasks` - Créer une tâche
- `GET /tasks` - Lister toutes les tâches
- `GET /tasks/:id` - Voir une tâche
- `PUT /tasks/:id` - Modifier une tâche
- `DELETE /tasks/:id` - Supprimer une tâche

### 2. Modèle de données

```
{
  id: "uuid",
  title: "string", (optionnel)
  description: "string",
  status: "string",
  createdAt: "timestamp", (optionnel)
  updatedAt: "timestamp" (optionnel)
}
```

## Code de démarrage

Voilà un morceau de code pouvant servir de base.

**app.js :**

```
const express = require('express');
const cors = require('cors');
const helmet = require('helmet');
const taskRoutes = require('./routes/tasks');
const errorHandler = require('./middleware/errorHandler');

const app = express();

// Middleware
app.use(helmet());
app.use(cors());
app.use(express.json());

// Health check
app.get('/health', (req, res) => {
  res.json({ status: 'ok', timestamp: new Date() });
});

// Routes
app.use('/api/tasks', taskRoutes);

// Error handling
app.use(errorHandler);
```

```
module.exports = app;
```

## Grille d'évaluation

Critère	Points	Description
API fonctionnelle	40%	CRUD complet, gestion erreurs
Tests	30%	Unit + integration, coverage >80%
Docker	20%	Dockerfile optimisé, docker-compose
Documentation	10%	README complet, API doc

Note :

- les tests seront à réaliser demain.
- d'autres critères s'ajouteront

## ALLONS PLUS LOIN

### Intéressant à approfondir

Patterns avancés

- Repository pattern
- Dependency injection
- Clean architecture

Optimisations

- Pagination des résultats
- Filtrage et tri
- Rate limiting

Sécurité API

- Input validation (Joi)
- SQL injection prevention
- CORS configuration

---

## 6 - Git Workflows et Gestion de Projet

### CORE NOTIONS

#### Git Flow vs GitHub Flow vs GitLab Flow

Qui utilise des branches dans ses projets ? Comment vous organisez ?

##### 1. Git Flow



```
main
├─ develop
│   ├── feature/add-authentication
│   ├── feature/improve-performance
│   └─ release/1.2.0
```

## 2. GitHub Flow

```
main
├─ feature/new-api-endpoint
├─ fix/memory-leak
└─ docs/update-readme
```

## 3. GitLab Flow

```
main → staging → production
├─ feature/task-categories
```

## Conventional Commits

```
# Format
<type>(<scope>): <subject>

# Examples
feat(api): add task filtering endpoint
fix(docker): resolve build cache issue
docs(readme): update installation steps
test(tasks): add integration tests for CRUD
chore(deps): update dependencies
```

## Protection des branches

```
# GitLab settings
- Protect main branch
- Require merge requests
- Require pipeline success
- Require code review
```

## Gestion de projet avec GitLab

### 1. Issues

- Templates d'issues
- Labels (bug, feature, documentation)
- Milestones

### 2. Merge Requests

- Template de MR
- Assignee et reviewers
- Linked issues

### 3. Boards

- Kanban automatique
- Swimlanes
- Burndown charts

## ALLONS PLUS LOIN

### Semantic Versioning

- MAJOR.MINOR.PATCH
- Breaking changes
- Automated with semantic-release

### Git hooks

- pre-commit (linting)
- commit-msg (conventional commits)
- pre-push (tests)

### Monorepo strategies

- Lerna
- Nx
- Turborepo

---

## Récapitulatif Jour 1

---

### Ce qu'on a appris aujourd'hui

1. **Architecture DevOps** : Les 3 piliers, le cycle DevOps
2. **Docker avancé** : Multi-stage, compose, best practices
3. **Développement d'API** : Structure, tests, documentation
4. **Git workflows** : Branches, commits, gestion de projet

### Points d'attention pour demain

- **Finir le TP** si nécessaire (API + Docker)
- **Réviser** les commandes Docker
- Nous ferons de l'automatisation CI/CD !