Projet de fin de module DevOps - Docker

Table of Contents

1. Introduction 3
1.1. Présentation générale 3
1.2. Modalités 3
1.2.1. Travail en binôme 3
1.2.2. Repository GitLab 3
1.2.3. Choix des technologies 4
1.2.4. Date limite 4
1.3. Objectifs pédagogiques 4
2. Architecture cible 5
2.1. Vue d'ensemble 5
2.2. Composants obligatoires 5
2.2.1. 1. Reverse-Proxy 5
2.2.2. 2. Serveur d'application Java EE 6
2.2.3. 3. Base de données 7
2.2.4. 4. Stack ELK 7
2.3. Déploiement 7
3. Exigences techniques détaillées 8
3.1. 1. Docker Compose 8
3.1.1. Structure du fichier 8
3.1.2. Variables d'environnement 9
3.1.3. Networks 9
3.1.4. Volumes 10
3.2. 2. Images Docker 10
3.2.1. Multi-stage builds 10
3.2.2. Taille des images 11
3.2.3. Fichier `.dockerignore` 12
3.2.4. Pas de secrets dans les images 12
3.3. 3. Health Checks 13

3.3.1. Sur tous les services . . . 13
3.3.2. Restart policies . . . 14
3.3.3. Dépendances avec conditions . . . 14
3.4. 4. Resource Management . . . 15
3.5. 5. Logging . . . 15
3.5.1. Format JSON . . . 16
3.5.2. Intégration Logstash . . . 16
3.5.3. Dashboards Kibana . . . 17
3.6. 6. Sécurité . . . 17
3.6.1. Utilisateur non-root . . . 17
3.6.2. Gestion des secrets . . . 17
3.6.3. Scanning de vulnérabilités avec Trivy . . . 18
3.7. 7. Documentation . . . 19
3.7.1. Sections obligatoires . . . 19
3.7.2. Diagramme d'architecture . . . 21
4. Grille d'évaluation 22
4.1. Critères principaux (10 points) . . . 22
4.2. Compléments valorisés . . . 23
4.2.1. Complément 1 : Observabilité avec Prometheus + Grafana (+1 pt) . . . 23
4.2.2. Complément 2 : Pipeline de validation avec Goss (+1 pt) . . . 25
4.2.3. Complément 3 : Gestion automatique des dépendances (+0.75 pt) . . . 26
4.2.4. Complément 4 : Performance & Load Testing (+1 pt) . . . 27
4.2.5. Complément 5 : Haute disponibilité (+1 pt) . . . 29
4.2.6. Complément 6 : Multi-environnements (+0.75 pt) . . . 31
5. Conseils pratiques 33
5.1. Par où commencer ? . . . 33
5.2. Gestion du temps . . . 33
5.3. Workflow Git recommandé . . . 34
5.4. Communication avec les enseignants . . . 35
5.5. Utilisation de l'IA (ChatGPT, Claude, etc.) . . . 36
5.6. Erreurs courantes à éviter . . . 36
6. Ressources utiles 37
6.1. Documentation officielle . . . 37
6.2. Exemples et tutoriels . . . 37
6.3. Outils . . . 38
7. FAQ 38

8. Conclusion 40

Date limite de rendu : 4 janvier 2026
Seuls les commits antérieurs à cette date seront pris en compte pour la correction.

1. Introduction

1.1. Présentation générale

Ce document décrit le projet de fin de module pour le cours **DevOps - Docker** du Master 2 ILI, année universitaire 2025-2026.
Le projet se décompose en **deux parties distinctes** :

- **Partie 1** : Infrastructure Docker (notée sur 10 points)
- **Partie 2** : Pipeline CI/CD (notée sur 10 points)

Note finale = **Partie 1** + **Partie 2** = **/20**
Ce document détaille uniquement la **Partie 1 - Infrastructure Docker**.

1.2. Modalités

1.2.1. Travail en binôme

- Le projet est **obligatoirement** réalisé en binôme
- Les binômes seront constitués au début du projet
- Les deux membres du binôme recevront la même note

1.2.2. Repository GitLab

Vous devez créer un repository **privé** sur l'instance GitLab de l'université avec les caractéristiques suivantes :

- Repository **privé** (pas public)
- Les enseignants ajoutés en tant que **Maintainer** :
  - Bruno Verachten (gounthar)
  - Daniel Le Berre
  - Farid Ait-Kara
- Documentation complète dans le `README.md`

- Respect des bonnes pratiques Git (commits atomiques, branches, merge requests)

Communication importante :
Pour que nous soyons notifiés de vos demandes :

- Créez des **Merge Requests** (MR) et ajoutez-nous en reviewers
- Créez des **Issues** pour les questions techniques
- **Ne commitez PAS directement sur** `main` sans MR

Les commits directs sur `main` ne génèrent pas de notification !

1.2.3. Choix des technologies

Le choix des technologies (reverse-proxy, serveur d'application) sera effectué via un **vote sur Moodle** en début de projet.

1.2.4. Date limite

4 janvier 2026 - 23h59
Seuls les commits effectués avant cette date seront corrigés. Aucune exception ne sera accordée. Les commits ne mentent pas.

1.3. Objectifs pédagogiques

Ce projet a pour objectif de vous préparer aux pratiques professionnelles DevOps modernes en vous permettant de :

- Concevoir et déployer une **architecture multi-tiers** complète avec Docker
- Appliquer les **bonnes pratiques de sécurité** et d'optimisation
- Documenter et justifier vos **choix techniques**
- Mettre en œuvre des **outils modernes** d'observabilité et de qualité
- Communiquer efficacement via Git et les outils collaboratifs

Notre philosophie :
L'objectif est de vous **préparer** à l'entreprise, pas de vous **piéger** !
N'hésitez pas à :

- Poser des questions (via Issues GitLab ou MR, ou encore Mattermost ou par email)
- Demander des clarifications
- Éventuellement, solliciter une review intermédiaire

2. Architecture cible

2.1. Vue d'ensemble

Vous allez créer une architecture **production-ready** conteneurisée qui déploie une application Java EE complète.
L'architecture comporte **4 composants principaux** :

```
@startuml
!theme plain

rectangle "Client Web" as client #LightBlue

rectangle "Reverse Proxy\n(Nginx/Apache/Traefik/...)" as proxy #Orange

rectangle "Serveur d'Application\n(Tomcat/Wildfly/Jetty)" as app #Green

database "Base de Données\n(PostgreSQL/MySQL)" as db #Yellow

rectangle "Stack ELK" as elk #Purple {
  component "Elasticsearch" as es
  component "Logstash" as ls
  component "Kibana" as kb
}

client --> proxy : "HTTP/HTTPS"
proxy --> app : "Proxy pass"
app --> db : "JDBC"
app --> ls : "Logs"
proxy --> ls : "Logs"
ls --> es : "Indexation"
es --> kb : "Visualisation"

@enduml
```

2.2. Composants obligatoires

2.2.1. 1. Reverse-Proxy

Choix à faire parmi :

- Apache HTTPd
- Nginx
- Nginx Unit
- HAProxy
- Traefik

- Caddy

Rôle :

- Point d'entrée unique de l'architecture
- Terminaison SSL/TLS (si HTTPS implémenté)
- Load-balancing si plusieurs instances du serveur d'application
- Routage des requêtes vers le serveur d'application

Critères de choix :

Justifiez votre choix dans le README en fonction de critères comme :

- Performance
- Facilité de configuration
- Support natif de certaines fonctionnalités (ex: ACME pour Caddy, service mesh pour Traefik)
- Expérience personnelle ou curiosité technique

2.2.2. 2. Serveur d'application Java EE

Choix à faire parmi :

- Apache Tomcat
- Wildfly (anciennement JBoss)
- Jetty

Rôle :

- Héberger et exécuter l'application web (fichier WAR)
- Gérer les sessions utilisateurs
- Communiquer avec la base de données via JDBC
- Exposer l'application sur un port interne (non exposé directement)

Application :

Vous devez déployer un fichier WAR fonctionnel. Vous pouvez :

- Utiliser un exemple fourni dans ce module de cours, dans sa seconde partie, ou dans le cours de M. Leberre.
- Créer votre propre application simple (ex: CRUD basique)
- Utiliser une application open-source (ex: Jenkins, Nexus, etc.)

L'application doit :

- Se connecter à la base de données
- Être fonctionnelle et accessible via le reverse-proxy
- Avoir au minimum une page d'accueil et une action métier (CRUD,

> calcul, etc.)

2.2.3. 3. Base de données

Choix à faire parmi :

- PostgreSQL (recommandé)
- MySQL

Rôle :

- Stockage persistant des données applicatives
- Isolation réseau (non exposée directement à l'extérieur)
- Données persistées via volumes Docker

Exigences :

- Volumes **nommés** (pas de volumes anonymes)
- Health checks configurés
- Credentials gérés via variables d'environnement (pas en dur)
- Script d'initialisation optionnel (schema SQL)

2.2.4. 4. Stack ELK

Composants :

- **Elasticsearch** : Moteur de recherche et d'indexation des logs
- **Logstash** : Collecteur et transformateur de logs
- **Kibana** : Interface de visualisation et d'analyse

Rôle :

- Centraliser les logs de tous les services
- Fournir des dashboards de monitoring
- Permettre la recherche et l'analyse des logs

Exigences minimales :

- Les logs du reverse-proxy et du serveur d'application doivent être envoyés à Logstash
- Elasticsearch doit indexer ces logs
- Kibana doit être accessible et afficher au moins un dashboard basique
- Les logs doivent être au format JSON quand c'est possible

2.3. Déploiement

L'ensemble de l'architecture doit pouvoir se démarrer avec une seule commande :

```
docker compose up -d
```

Temps de démarrage attendu : < 2 minutes
Tous les services doivent être "healthy" après ce délai.

3. Exigences techniques détaillées

3.1. 1. Docker Compose

3.1.1. Structure du fichier

Votre fichier `docker-compose.yml` doit être :

- **Bien structuré** : utilisation de l'indentation YAML correcte
- **Lisible** : commentaires pour expliquer les choix non-évidents
- **Modulaire** : utilisation de `extends` ou `x-` anchors pour éviter la duplication

Exemple de bonne structure :

```yaml
# Ancres YAML pour réutiliser des configurations
x-common-healthcheck: &common-healthcheck
  interval: 30s
  timeout: 10s
  retries: 3
  start_period: 40s

x-logging: &default-logging
  driver: json-file
  options:
    max-size: "10m"
    max-file: "3"

services:
  reverse-proxy:
    image: nginx:alpine
    ports:
      - "${PROXY_PORT:-80}:80"
    networks:
      - frontend
    volumes:
      - ./nginx/nginx.conf:/etc/nginx/nginx.conf:ro
    depends_on:
      app-server:
        condition: service_healthy
    healthcheck:
      <<: *common-healthcheck
      test: ["CMD", "nginx", "-t"]
    logging: *default-logging
    deploy:
```

```
      resources:
        limits:
          cpus: '0.5'
          memory: 256M
        reservations:
          cpus: '0.1'
          memory: 128M

  # ... autres services
```

3.1.2. Variables d'environnement

Fichier `.env` :

- Contient TOUTES les variables sensibles (mots de passe, secrets, etc.)
- **NE DOIT PAS** être commité dans Git
- Ajouté au `.gitignore`

Fichier `.env.example` :

- Contient un template avec toutes les variables nécessaires
- Les valeurs sont des exemples ou des placeholders
- **DOIT** être commité dans Git pour guider les utilisateurs

Exemple de `.env.example` :

```
# Port exposé du reverse-proxy
PROXY_PORT=80

# Configuration base de données
POSTGRES_DB=myapp
POSTGRES_USER=appuser
POSTGRES_PASSWORD=CHANGEME

# Configuration application
APP_SECRET_KEY=CHANGEME_RANDOM_STRING
APP_ENV=production

# Configuration ELK
ELASTIC_PASSWORD=CHANGEME
KIBANA_PASSWORD=CHANGEME
```

Sécurité :

- Les mots de passe par défaut doivent être changés
- Les secrets ne doivent JAMAIS apparaître en clair dans le `compose.yml`
- Utilisez `${VARIABLE_NAME}` pour référencer les variables

3.1.3. Networks

Vous devez créer **au minimum 2 réseaux** :

- **frontend** : Réseau exposé contenant le reverse-proxy
- **backend** : Réseau privé pour la communication base de données / application

Exemple :

```
networks:
  frontend:
    driver: bridge
  backend:
    driver: bridge
    internal: true  # Pas d'accès externe (optionnel mais recommandé)
```

Isolation :

- Le reverse-proxy est connecté au réseau `frontend`
- Le serveur d'application est connecté aux réseaux `frontend` et `backend`
- La base de données est connectée uniquement au réseau `backend`

Cette isolation améliore la sécurité en limitant les communications possibles.

3.1.4. Volumes

Tous les volumes doivent être **nommés** (pas de volumes anonymes).

Exemple :

```
volumes:
  postgres_data:
    driver: local
  elasticsearch_data:
    driver: local
```

Pourquoi des volumes nommés ?

- Évite la perte de données lors de `docker compose down`
- Permet de facilement identifier et gérer les volumes
- Facilite les sauvegardes et restaurations

3.2. 2. Images Docker

3.2.1. Multi-stage builds

Pour les services construits (pas ceux utilisant des images officielles), vous devez utiliser des **multi-stage builds**.

Exemple pour une application Java :

```
# Stage 1: Build
FROM maven:3.9-eclipse-temurin-17 AS builder

WORKDIR /app
COPY pom.xml .
# Télécharger les dépendances (cache Docker layer)
RUN mvn dependency:go-offline

COPY src ./src
RUN mvn package -DskipTests

# Stage 2: Runtime
FROM tomcat:10-jre17-temurin-alpine

# Supprimer les apps par défaut de Tomcat
RUN rm -rf /usr/local/tomcat/webapps/*

# Copier le WAR depuis le stage de build
COPY --from=builder /app/target/myapp.war /usr/local/tomcat/webapps/ROOT.war

# Utilisateur non-root (sécurité)
RUN addgroup -g 1001 -S appgroup && \
    adduser -u 1001 -S appuser -G appgroup && \
    chown -R appuser:appgroup /usr/local/tomcat

USER appuser

EXPOSE 8080

CMD ["catalina.sh", "run"]
```

Avantages :

- Image finale plus légère (pas de Maven, pas de code source)
- Séparation build / runtime
- Amélioration de la sécurité

3.2.2. Taille des images

Critères d'évaluation :

- Images applicatives (serveur d'app) : < **500 MB**
- Images de services (proxy, base de données) : < **200 MB**

Conseils pour réduire la taille :

- Utiliser des images de base Alpine quand possible (`alpine`, `slim`)
- Multi-stage builds
- Nettoyer les caches (`apt-get clean`, `yum clean all`, etc.)
- Utiliser `.dockerignore`

3.2.3. Fichier `.dockerignore`

Chaque service avec un Dockerfile doit avoir un `.dockerignore` configuré.

Exemple :

```
# Git
.git
.gitignore
.gitattributes

# CI/CD
.github
.gitlab-ci.yml

# Documentation
README.md
docs/
*.md

# IDE
.vscode/
.idea/
*.iml

# Build artifacts (pour éviter de les copier dans l'image)
target/
build/
dist/
node_modules/

# Tests
tests/
test/
*.test.js

# Environnement
.env
.env.*
!.env.example
```

3.2.4. Pas de secrets dans les images

Interdictions absolues :

- Mots de passe en dur dans les Dockerfiles
- Clés privées (SSH, GPG, etc.) copiées dans l'image
- Tokens d'API stockés dans le code source
- Fichiers `.env` copiés dans l'image

Bonne pratique :

- Utiliser des variables d'environnement passées au runtime

- Utiliser Docker Secrets (pour Swarm) ou des outils externes (Vault)
- Vérifier avec `docker history <image>` qu'aucun secret n'est visible

3.3. 3. Health Checks

3.3.1. Sur tous les services

Chaque service doit avoir un health check configuré.

Exemples par type de service :

Base de données PostgreSQL :

```
healthcheck:
  test: ["CMD-SHELL", "pg_isready -U ${POSTGRES_USER} -d ${POSTGRES_DB}"]
  interval: 10s
  timeout: 5s
  retries: 5
  start_period: 10s
```

Base de données MySQL :

```
healthcheck:
  test: ["CMD", "mysqladmin", "ping", "-h", "localhost", "-u", "root", "-p${MYSQL_ROOT_PASSWORD}"]
  interval: 10s
  timeout: 5s
  retries: 5
  start_period: 10s
```

Serveur d'application (Tomcat) :

```
healthcheck:
  test: ["CMD", "curl", "-f", "http://localhost:8080/health"]
  interval: 30s
  timeout: 10s
  retries: 3
  start_period: 40s
```

Pour Tomcat, vous devez créer un endpoint `/health` simple qui retourne 200 OK. Si votre application n'a pas de endpoint dédié, vous pouvez tester la page d'accueil.

Reverse-proxy (Nginx) :

```
healthcheck:
  test: ["CMD", "nginx", "-t"]
  interval: 30s
  timeout: 5s
```

```
  retries: 3
  start_period: 10s
```

Elasticsearch :

```
healthcheck:
  test: ["CMD-SHELL", "curl -f http://localhost:9200/_cluster/health || exit 1"]
  interval: 30s
  timeout: 10s
  retries: 5
  start_period: 60s
```

3.3.2. Restart policies

Chaque service doit avoir une politique de redémarrage appropriée.

Choix recommandés :

- `unless-stopped` : Pour les services critiques qui doivent toujours tourner
- `on-failure:5` : Pour les services applicatifs (redémarre max 5 fois en cas d'erreur)

Exemple :

```
services:
  database:
    # ...
    restart: unless-stopped

  app-server:
    # ...
    restart: on-failure:5
```

3.3.3. Dépendances avec conditions

Utilisez `depends_on` avec la condition `service_healthy` pour orchestrer le démarrage.

Exemple :

```
services:
  app-server:
    # ...
    depends_on:
      database:
        condition: service_healthy

  reverse-proxy:
    # ...
    depends_on:
      app-server:
        condition: service_healthy
```

Effet :

- Le serveur d'application ne démarre que quand la base de données est "healthy"
- Le reverse-proxy ne démarre que quand le serveur d'application est "healthy"
- Évite les erreurs de connexion au démarrage

3.4. 4. Resource Management

Chaque service doit avoir des limites de ressources CPU et mémoire.

Exemple :

```
services:
  database:
    # ...
    deploy:
      resources:
        limits:
          cpus: '1.0'      # Maximum 1 CPU
          memory: 1024M    # Maximum 1 Go de RAM
        reservations:
          cpus: '0.25'     # Réservation minimum
          memory: 512M     # Réservation minimum
```

Recommandations par service :

Service	CPU Limit	Memory Limit
Reverse-proxy	0.5	256M
Serveur d'application	1.0	1024M
Base de données	1.0	1024M
Elasticsearch	2.0	2048M
Logstash	1.0	1024M
Kibana	1.0	512M

Ces valeurs sont des recommandations de départ. Vous pouvez les ajuster selon :

- Les ressources de votre machine
- Les tests de charge
- Les métriques observées

3.5. 5. Logging

3.5.1. Format JSON

Les logs doivent être au format JSON quand c'est possible.

Configuration Docker Compose :

```
services:
  app-server:
    # ...
    logging:
      driver: json-file
      options:
        max-size: "10m"
        max-file: "3"
        labels: "service=app,env=production"
```

Rotation des logs :

- `max-size` : Taille maximale d'un fichier de log (ex: 10m = 10 mégaoctets)
- `max-file` : Nombre de fichiers conservés (ex: 3 = logs des 3 dernières rotations)

3.5.2. Intégration Logstash

Les logs doivent être envoyés à Logstash pour centralisation.

Configuration Logstash (exemple) :

```
input {
  tcp {
    port => 5000
    codec => json
  }
}

filter {
  # Ajout de métadonnées
  mutate {
    add_field => { "[@metadata][environment]" => "production" }
  }
}

output {
  elasticsearch {
    hosts => ["elasticsearch:9200"]
    index => "app-logs-%{+YYYY.MM.dd}"
  }
}
```

Configuration application (Logback pour Java) :

```
<configuration>
  <appender name="LOGSTASH" class="net.logstash.logback.appender.LogstashTcpSocketAppender">
    <destination>logstash:5000</destination>
    <encoder class="net.logstash.logback.encoder.LogstashEncoder" />
```

```
  </appender>

  <root level="INFO">
    <appender-ref ref="LOGSTASH" />
  </root>
</configuration>
```

3.5.3. Dashboards Kibana

Vous devez créer au moins **un dashboard Kibana basique** avec :

- Graphique de volume de logs par service
- Tableau des derniers logs
- Filtre par niveau (INFO, WARN, ERROR)

3.6. 6. Sécurité

3.6.1. Utilisateur non-root

Tous les conteneurs doivent tourner avec un utilisateur **non-root**.

Dans le Dockerfile :

```
# Créer un utilisateur
RUN addgroup -g 1001 -S appgroup && \
    adduser -u 1001 -S appuser -G appgroup

# Changer les permissions
RUN chown -R appuser:appgroup /app

# Passer à l'utilisateur
USER appuser
```

Ou dans le docker-compose.yml :

```
services:
  app-server:
    # ...
    user: "1001:1001"
```

Vérification :

```
# Doit afficher un utilisateur autre que root (uid != 0)
docker compose exec app-server whoami
```

3.6.2. Gestion des secrets

Options recommandées :

1. **Variables d'environnement** (solution simple pour l'exercice)
2. **Docker Secrets** (si vous utilisez Swarm mode)
3. **HashiCorp Vault** (complément valorisé avancé)

Exemple avec Docker Secrets :

```
secrets:
  db_password:
    file: ./secrets/db_password.txt

services:
  database:
    # ...
    secrets:
      - db_password
    environment:
      POSTGRES_PASSWORD_FILE: /run/secrets/db_password
```

3.6.3. Scanning de vulnérabilités avec Trivy

Vous devez scanner vos images avec **Trivy**.

Installation :

```
# Linux
curl -sfL https://raw.githubusercontent.com/aquasecurity/trivy/main/contrib/install.sh | sh -s -- -b
/usr/local/bin

# macOS
brew install trivy

# Docker
docker run --rm -v /var/run/docker.sock:/var/run/docker.sock \
  aquasec/trivy image myimage:latest
```

Utilisation :

```
# Scanner une image locale
trivy image myapp:latest

# Scanner avec un niveau de sévérité minimum
trivy image --severity HIGH,CRITICAL myapp:latest

# Générer un rapport JSON
trivy image -f json -o report.json myapp:latest
```

Exigence :

- Aucune vulnérabilité **CRITICAL** non justifiée
- Les vulnérabilités **HIGH** doivent être documentées dans le README

3.7. 7. Documentation

Le fichier `README.md` doit être **complet et professionnel**.

3.7.1. Sections obligatoires

````
# Nom du Projet

Description brève du projet (1-2 paragraphes).

## Architecture

### Vue d'ensemble

Diagramme de l'architecture (PlantUML, draw.io, ou autre).

### Composants

Description de chaque composant :
- Rôle
- Technologies utilisées
- Ports exposés
- Dépendances

## Prérequis

- Docker >= 24.0
- Docker Compose >= 2.20
- Minimum 8 Go de RAM disponible
- etc.

## Installation

### 1. Cloner le repository

```bash
git clone <url>
cd <project>
```

### 2. Configuration

Copier le fichier d'environnement :

```bash
cp .env.example .env
```

Éditer `.env` et modifier les valeurs :
- `POSTGRES_PASSWORD` : mot de passe de la base de données
- etc.

### 3. Démarrage
````

```bash
docker compose up -d
```

4. Vérification

Vérifier que tous les services sont "healthy" :

```bash
docker compose ps
```

Utilisation

Accès à l'application

- Application web : http://localhost:8080
- Kibana (logs) : http://localhost:5601

Comptes par défaut

⚠️ À changer en production !

- Kibana : elastic / changeme
- Base de données : appuser / changeme

Configuration avancée

(Si applicable : multi-environnements, HA, etc.)

Tests

(Comment tester que l'application fonctionne)

```bash
# Test du endpoint health
curl http://localhost:8080/health

# Test d'une fonctionnalité métier
curl -X POST http://localhost:8080/api/items -d '{"name":"test"}'
```

Troubleshooting

Problème : Service ne démarre pas

```bash
# Voir les logs du service
docker compose logs -f <service-name>

# Redémarrer un service spécifique
docker compose restart <service-name>
```

Problème : Base de données inaccessible

```
- Vérifier le health check : `docker compose ps`
- Vérifier les credentials dans `.env`
- Vérifier les logs : `docker compose logs database`

## Choix techniques

### Choix du reverse-proxy : Nginx

Justification :
- Performance excellente
- Configuration simple
- Large communauté
- etc.

### Choix du serveur d'application : Tomcat

Justification :
- Léger et rapide
- Support natif des WAR
- Bonne intégration avec l'écosystème Java
- etc.

## Améliorations futures

(Optionnel : ce qui pourrait être ajouté)

## Contributeurs

- Prénom Nom (@username)
- Prénom Nom (@username)

## Licence

(Si applicable)
```

3.7.2. Diagramme d'architecture

Vous devez inclure un **diagramme visuel** de l'architecture.

Outils recommandés :

- PlantUML (génération automatique depuis le code)
- draw.io / diagrams.net
- Mermaid (intégré dans GitLab/GitHub)
- Excalidraw

Exemple PlantUML :

```
@startuml
!theme plain

actor User

rectangle "Docker Compose" {
```

```
  component "Nginx" as proxy
  component "Tomcat" as app
  database "PostgreSQL" as db
  component "Elasticsearch" as es
  component "Logstash" as ls
  component "Kibana" as kb
}

User --> proxy : HTTP
proxy --> app
app --> db : JDBC
app --> ls : Logs
proxy --> ls : Logs
ls --> es
es --> kb
User --> kb : Monitoring

@enduml
```

4. Grille d'évaluation

4.1. Critères principaux (10 points)

La Partie 1 est notée sur **10 points** répartis selon les critères suivants :

Critère	Points	Détail
Architecture fonctionnelle	**3 pts**	• 1 pt : Stack complète déployable avec `docker compose up -d` • 1 pt : Tous les services sont accessibles et fonctionnels • 1 pt : Communication entre les services opérationnelle
Qualité technique	**3 pts**	• 0.75 pt : Multi-stage builds et images optimisées • 0.75 pt : Health checks et depends_on corrects • 0.75 pt : Resource limits configurés • 0.75 pt : Sécurité (utilisateur non-root, secrets, Trivy)

Critère	Points	Détail
Documentation	**2 pts**	• 1 pt : README complet avec architecture, prérequis, installation • 0.5 pt : Diagramme d'architecture clair • 0.5 pt : Justifications des choix techniques
Bonnes pratiques	**2 pts**	• 0.5 pt : Images < 500 MB (app) et < 200 MB (services) • 0.5 pt : Logs structurés (JSON) vers Logstash/Kibana • 0.5 pt : Variables d'environnement (.env), pas de secrets en clair • 0.5 pt : Networks isolés, volumes nommés
TOTAL	**10 pts**	

4.2. Compléments valorisés

Les compléments valorisés permettent de **compenser les points manquants** et d'atteindre la note maximale de 10/10.

Formule de calcul :

```
Note Partie 1 = min(10, Critères principaux + Compléments valorisés)
```

Exemple :

- 7/10 sur les critères principaux + 2 points de compléments = **9/10**
- 8/10 sur les critères principaux + 3 points de compléments = **10/10** (plafonné)

Stratégie recommandée : Qualité > Quantité

Il vaut mieux :

- **1-2 compléments de qualité** (bien implémentés, documentés, fonctionnels)
- Que **4 compléments bâclés** (partiellement fonctionnels, non documentés)

Choisissez les compléments qui **vous intéressent** et que vous voulez **approfondir** !

4.2.1. Complément 1 : Observabilité avec Prometheus + Grafana (+1 pt)

Objectif : Mettre en place une stack de monitoring moderne.

Composants :

- **Prometheus** : Collecte de métriques temps réel
- **Grafana** : Visualisation et dashboards
- **Exporters** : cAdvisor (métriques Docker), Node Exporter (métriques système)
- **Micrometer** : Bibliothèque Java pour exposer des métriques applicatives

Critères d'évaluation :

- 0.5 pt : Prometheus configuré et collecte les métriques
- 0.25 pt : Dashboard Grafana "système" (CPU, RAM, réseau des conteneurs)
- 0.25 pt : Dashboard Grafana "applicatif" (métriques métier, latence, throughput)
- Bonus : Au moins 1 alerte Prometheus configurée

Configuration Prometheus (exemple) :

```yaml
# prometheus.yml
global:
  scrape_interval: 15s

scrape_configs:
  # Métriques Docker via cAdvisor
  - job_name: 'cadvisor'
    static_configs:
      - targets: ['cadvisor:8080']

  # Métriques système
  - job_name: 'node-exporter'
    static_configs:
      - targets: ['node-exporter:9100']

  # Métriques applicatives (si Micrometer configuré)
  - job_name: 'spring-app'
    metrics_path: '/actuator/prometheus'
    static_configs:
      - targets: ['app-server:8080']
```

Intégration Micrometer dans Spring Boot :

```xml
<!-- pom.xml -->
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```

```properties
# application.properties
management.endpoints.web.exposure.include=health,info,prometheus
management.metrics.export.prometheus.enabled=true
```

Ressources utiles :

- https://prometheus.io/docs/introduction/overview/
- https://grafana.com/docs/grafana/latest/getting-started/
- https://github.com/google/cadvisor
- https://micrometer.io/docs

4.2.2. Complément 2 : Pipeline de validation avec Goss (+1 pt)

Objectif : Automatiser les tests de l'infrastructure Docker.
Outil : Goss - Outil de tests d'infrastructure
Critères d'évaluation :

- 0.5 pt : Fichier `goss.yaml` avec tests complets
- 0.25 pt : Pipeline CI/CD qui exécute les tests Goss
- 0.25 pt : Rapport de tests généré (JUnit XML ou JSON)

Exemple de tests Goss :

```
# goss.yaml
port:
  tcp:80:
    listening: true
    ip:
      - 0.0.0.0

  tcp:5432:
    listening: true

http:
  http://localhost:80:
    status: 200
    timeout: 5000

  http://localhost:80/health:
    status: 200
    body:
      - '{"status":"UP"}'

docker-container:
  reverse-proxy:
    running: true
    status: "running"

  app-server:
    running: true
    status: "running"
    health-status: "healthy"

  database:
    running: true
    status: "running"
```

```
    health-status: "healthy"
```

Pipeline GitLab CI (exemple) :

```
# .gitlab-ci.yml
test-infrastructure:
  stage: test
  image: docker:24
  services:
    - docker:24-dind
  script:
    - apk add --no-cache curl
    # Installer Goss
    - curl -fsSL https://goss.rocks/install | sh
    # Démarrer les services
    - docker compose up -d
    # Attendre que les services soient healthy
    - sleep 30
    # Exécuter les tests
    - goss validate --format junit > goss-report.xml
  artifacts:
    reports:
      junit: goss-report.xml
```

Ressources utiles :

- https://github.com/goss-org/goss
- https://github.com/aelsabbahy/goss/blob/master/docs/manual.md

4.2.3. Complément 3 : Gestion automatique des dépendances (+0.75 pt)

Objectif : Automatiser la mise à jour des dépendances (images Docker, dépendances Maven, etc.).

Outils (choisir un) :

- **Updatecli** (recommandé pour Docker)
- **Renovate**
- **Dependabot**

Critères d'évaluation :

- 0.5 pt : Configuration fonctionnelle avec au moins 3 dépendances surveillées
- 0.25 pt : Au moins 1 Pull Request automatique créée par l'outil

Configuration Updatecli (exemple) :

```
# updatecli/updatecli.d/docker-images.yaml
name: "Update Docker images"

sources:
  nginx:
    kind: dockerimage
```

```
    spec:
      image: nginx
      tagfilter: "alpine$"

  postgres:
    kind: dockerimage
    spec:
      image: postgres
      tagfilter: "^15-alpine$"

targets:
  docker-compose-nginx:
    name: "Update nginx image in docker-compose.yml"
    kind: yaml
    spec:
      file: docker-compose.yml
      key: services.reverse-proxy.image
    sourceid: nginx

  docker-compose-postgres:
    name: "Update postgres image in docker-compose.yml"
    kind: yaml
    spec:
      file: docker-compose.yml
      key: services.database.image
    sourceid: postgres
```

Pipeline GitLab CI (exemple) :

```
# .gitlab-ci.yml
updatecli:
  stage: update-dependencies
  image: updatecli/updatecli:latest
  script:
    - updatecli diff --config updatecli/updatecli.d/
    - updatecli apply --config updatecli/updatecli.d/
  only:
    - schedules
```

Ressources utiles :

- https://www.updatecli.io/
- https://docs.renovatebot.com/
- https://docs.github.com/en/code-security/dependabot

4.2.4. Complément 4 : Performance & Load Testing (+1 pt)

Objectif : Tester les performances de l'application et prouver l'impact des optimisations.
Outils (choisir un) :

- **k6** (recommandé, simple et moderne)

- **Gatling** (plus complet, orienté Java)

Critères d'évaluation :

- 0.5 pt : Scénarios de tests de charge réalistes (connexion, CRUD, etc.)
- 0.25 pt : Rapport de performance avec métriques clés (throughput, latence, erreurs)
- 0.25 pt : Preuve d'optimisations (comparaison avant/après)

Exemple de test k6 :

```
// load-test.js
import http from 'k6/http';
import { check, sleep } from 'k6';

export const options = {
  stages: [
    { duration: '30s', target: 10 },  // Montée à 10 utilisateurs
    { duration: '1m', target: 50 },   // Montée à 50 utilisateurs
    { duration: '30s', target: 0 },   // Descente à 0
  ],
  thresholds: {
    http_req_duration: ['p(95)<500'], // 95% des requêtes < 500ms
    http_req_failed: ['rate<0.01'],   // Taux d'erreur < 1%
  },
};

export default function () {
  // Test page d'accueil
  const res = http.get('http://localhost:8080');
  check(res, {
    'status is 200': (r) => r.status === 200,
    'response time < 500ms': (r) => r.timings.duration < 500,
  });

  sleep(1);

  // Test API
  const payload = JSON.stringify({
    name: 'Test Item',
    value: Math.random() * 100,
  });
  const params = {
    headers: { 'Content-Type': 'application/json' },
  };
  const apiRes = http.post('http://localhost:8080/api/items', payload, params);
  check(apiRes, {
    'API status is 201': (r) => r.status === 201,
  });

  sleep(1);
}
```

Exécution :

```
# Installer k6
```

```
brew install k6  # macOS
# ou
docker pull grafana/k6

# Exécuter le test
k6 run load-test.js

# Générer un rapport HTML
k6 run --out json=results.json load-test.js
```

Métriques attendues dans le rapport :

- **Throughput** : requêtes/seconde (RPS)
- **Latence** : p50, p95, p99 (percentiles)
- **Taux d'erreur** : % de requêtes échouées
- **Comparaison avant/après** : prouver l'impact des optimisations (ex: cache Redis, tuning JVM, etc.)

Ressources utiles :

- https://k6.io/docs/
- https://gatling.io/docs/

4.2.5. Complément 5 : Haute disponibilité (+1 pt)

Objectif : Déployer plusieurs instances du serveur d'application avec load-balancing.

Critères d'évaluation :

- 0.5 pt : Au moins 2 instances du serveur d'application
- 0.25 pt : Load balancing configuré sur le reverse-proxy (round-robin ou least-conn)
- 0.25 pt : Démonstration de zero-downtime deployment (script de rolling update)

Configuration Docker Compose (scale) :

```
services:
  app-server:
    # ... configuration habituelle
    deploy:
      replicas: 3  # 3 instances

  reverse-proxy:
    # ... configuration habituelle
    depends_on:
      - app-server
```

Configuration Nginx (load balancing) :

```
# nginx.conf
upstream app_servers {
    least_conn;  # Ou : round_robin, ip_hash
    server app-server-1:8080;
```

```
    server app-server-2:8080;
    server app-server-3:8080;
}

server {
    listen 80;

    location / {
        proxy_pass http://app_servers;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;

        # Health check passif
        proxy_next_upstream error timeout http_502 http_503 http_504;
    }
}
```

Script de rolling update :

```
#!/bin/bash
# rolling-update.sh

# Mettre à jour les instances une par une
for instance in app-server-1 app-server-2 app-server-3; do
  echo "Updating $instance..."

  # Arrêter l'instance
  docker compose stop $instance

  # Mettre à jour l'image
  docker compose pull $instance

  # Redémarrer l'instance
  docker compose up -d $instance

  # Attendre que l'instance soit healthy
  until [ "$(docker inspect --format='{{.State.Health.Status}}' $instance)" == "healthy" ]; do
    echo "Waiting for $instance to be healthy..."
    sleep 5
  done

  echo "$instance updated successfully"
  sleep 10  # Pause entre les mises à jour
done

echo "Rolling update completed!"
```

Démonstration :

- Lancer le script de rolling update
- Pendant l'exécution, faire des requêtes continues à l'application (`while true; do curl …; sleep 1; done`)

- Prouver qu'aucune requête n'échoue (taux d'erreur = 0%)

Ressources utiles :

- https://docs.docker.com/compose/compose-file/deploy/
- https://nginx.org/en/docs/http/load_balancing.html

4.2.6. Complément 6 : Multi-environnements (+0.75 pt)

Objectif : Gérer plusieurs environnements (dev, staging, prod) avec Docker Compose.

Critères d'évaluation :

- 0.5 pt : Au moins 3 environnements avec configurations différentes
- 0.25 pt : Utilisation de Compose overrides (`compose.override.yml`, `compose.prod.yml`)

Structure des fichiers :

```
project/
├──── compose.yml              # Configuration de base (commune)
├──── compose.override.yml     # Dev (chargé automatiquement)
├──── compose.staging.yml      # Staging
├──── compose.prod.yml         # Production
├──── .env.example
├──── .env.dev
├──── .env.staging
└──── .env.prod
```

`compose.yml` **(base commune) :**

```yaml
services:
  app-server:
    build: ./app
    environment:
      - APP_ENV=${APP_ENV}
    networks:
      - backend
```

`compose.override.yml` **(dev - chargé par défaut) :**

```yaml
services:
  app-server:
    build:
      context: ./app
      target: dev  # Stage de développement
    ports:
      - "8080:8080"  # Exposer pour debug
    volumes:
      - ./app/src:/app/src  # Hot-reload
    environment:
      - DEBUG=true
      - LOG_LEVEL=DEBUG
```

`compose.prod.yml` **(production) :**

```yaml
services:
  app-server:
    build:
      context: ./app
      target: production  # Stage optimisé
    environment:
      - DEBUG=false
      - LOG_LEVEL=WARN
    deploy:
      replicas: 3
      resources:
        limits:
          cpus: '1.0'
          memory: 1024M
    restart: unless-stopped
```

Utilisation :

```
# Environnement dev (par défaut)
docker compose up -d

# Environnement staging
docker compose -f compose.yml -f compose.staging.yml --env-file .env.staging up -d

# Environnement production
docker compose -f compose.yml -f compose.prod.yml --env-file .env.prod up -d
```

Scripts utiles :

```
#!/bin/bash
# deploy.sh

ENV=${1:-dev}

case $ENV in
  dev)
    docker compose up -d
    ;;
  staging)
    docker compose -f compose.yml -f compose.staging.yml --env-file .env.staging up -d
    ;;
  prod)
    docker compose -f compose.yml -f compose.prod.yml --env-file .env.prod up -d
    ;;
  *)
    echo "Usage: $0 {dev|staging|prod}"
    exit 1
    ;;
esac
```

Ressources utiles :

- https://docs.docker.com/compose/extends/

5. Conseils pratiques

5.1. Par où commencer ?

1. **Choisir les technologies** (vote Moodle)
2. **Créer le repository GitLab** et ajouter les enseignants
3. **Créer un** `docker-compose.yml` **basique** avec les 4 services
4. **Faire fonctionner l'architecture** (priorité absolue)
5. **Ajouter progressivement** : health checks, resource limits, logging
6. **Optimiser les images** : multi-stage builds, `.dockerignore`
7. **Sécuriser** : utilisateur non-root, Trivy, secrets
8. **Documenter au fur et à mesure** (pas à la fin !)
9. **Compléments valorisés** : uniquement si le temps le permet

Approche itérative recommandée :
Ne cherchez pas la perfection du premier coup. Procédez par étapes :

1. Faire fonctionner (même si imparfait)
2. Améliorer progressivement
3. Documenter chaque étape
4. Merger régulièrement (petites MR)

C'est exactement comme en entreprise !

5.2. Gestion du temps

Estimation de temps par tâche :

Tâche	Durée estimée
Configuration de base (compose.yml, services)	2-3h
Déploiement de l'application WAR	1-2h
Health checks et depends_on	1h
Resource limits et logging	1h

Tâche	Durée estimée
Sécurité (non-root, Trivy)	1-2h
Stack ELK (configuration complète)	2-3h
Documentation (README complet)	2-3h
Total critères principaux	**10-15h**
Complément 1 (Observabilité)	3-4h
Complément 2 (Goss)	2-3h
Complément 3 (Updatecli)	2h
Complément 4 (Load Testing)	3-4h
Complément 5 (HA)	3-4h
Complément 6 (Multi-env)	2h

Priorisez les critères principaux !
Assurez-vous d'avoir une architecture **fonctionnelle et bien documentée** avant de vous lancer dans les compléments valorisés.
Mieux vaut :

- 10/10 sur les critères principaux
- Que 6/10 sur les critères + 4 compléments partiels

5.3. Workflow Git recommandé

Branches :

- `main` : Code stable et fonctionnel uniquement
- `develop` : Branche de développement principale
- `feature/<nom>` : Branches pour les nouvelles fonctionnalités
- `fix/<nom>` : Branches pour les corrections de bugs

Commits :

Utilisez le format **Conventional Commits** :

```
feat(compose): add PostgreSQL service with health check
fix(nginx): correct proxy_pass configuration
docs(readme): add architecture diagram
chore(deps): update nginx to 1.25-alpine
```

Merge Requests :

- Créez des MR **petites et focalisées**
- Ajoutez les enseignants en reviewers
- Décrivez clairement ce qui change et pourquoi
- Attendez les retours avant de merger

Exemple de description de MR :

```
## Description

Ajout du service PostgreSQL avec :
- Volume nommé pour la persistance
- Health check basé sur `pg_isready`
- Credentials via variables d'environnement

## Checklist

- [x] Service démarre correctement
- [x] Health check fonctionnel
- [x] Documentation mise à jour
- [ ] Tests Goss ajoutés (prévu dans prochaine MR)

## Captures d'écran

(Optionnel mais apprécié)
```

5.4. Communication avec les enseignants

Privilégiez GitLab pour la communication :

- **Issues** : Pour les questions techniques, blocages, demandes de clarification
- **Merge Requests** : Pour les reviews de code, validation d'approches
- **Comments** : Pour des retours précis sur du code

Format d'une bonne issue :

```
## Problème

Le health check de Tomcat échoue systématiquement après 40 secondes.

## Ce que j'ai essayé

1. Augmenté `start_period` à 60s
2. Testé manuellement avec `curl http://localhost:8080` → fonctionne
3. Vérifié les logs : aucune erreur visible

## Question

Est-ce que le health check doit pointer vers un endpoint spécifique ?
Ou est-ce que la page d'accueil Tomcat par défaut suffit ?
```

```
## Contexte

- Serveur : Tomcat 10
- OS : Ubuntu 22.04
- Docker : 24.0.6
- Branch : feature/tomcat-healthcheck
```

5.5. Utilisation de l'IA (ChatGPT, Claude, etc.)

L'IA est autorisée pour :

- Comprendre des concepts
- Débloquer sur des erreurs
- Générer des configurations de base
- Apprendre de nouvelles syntaxes

MAIS ATTENTION AU PLAGIAT !

Nous reconnaissons facilement :

- Le code généré par IA (style, commentaires, structure)
- Les explications copiées-collées sans compréhension
- Les configurations "par défaut" non adaptées à votre cas

L'IA doit vous aider à APPRENDRE, pas à ÉVITER DE RÉFLÉCHIR.

Bonne utilisation de l'IA :

```
Prompt : "Explique-moi comment fonctionne un health check Docker Compose"
→ Lire la réponse, comprendre, adapter à votre cas
→ Tester, documenter ce que vous avez compris
```

Mauvaise utilisation de l'IA :

```
Prompt : "Génère-moi un docker-compose.yml complet pour l'examen"
→ Copier-coller sans comprendre
→ Ça ne marche pas, vous ne savez pas pourquoi
→ Vous ne pouvez pas expliquer vos choix
```

5.6. Erreurs courantes à éviter

Erreur	Solution
□ Secrets dans Git (`.env` committé)	□ Ajouter `.env` au `.gitignore`, committer `.env.example`
□ Images trop grosses (> 1 Go)	□ Multi-stage builds, images Alpine, `.dockerignore`

Erreur	Solution
❌ Pas de health checks	✅ Ajouter `healthcheck` sur tous les services
❌ Volumes anonymes	✅ Utiliser des volumes **nommés** (section `volumes:`)
❌ Tout en `root`	✅ Créer un utilisateur non-root dans les Dockerfiles
❌ Documentation absente	✅ Documenter au fur et à mesure, pas à la fin
❌ Commits directs sur `main`	✅ Utiliser des branches et des Merge Requests
❌ Configurations en dur (localhost, ports)	✅ Utiliser des variables d'environnement
❌ Logs perdus (sortie standard uniquement)	✅ Intégration avec Logstash/Kibana
❌ Pas de tests (Goss, load testing)	✅ Automatiser les tests dans la CI/CD

6. Ressources utiles

6.1. Documentation officielle

- **Docker** : https://docs.docker.com/
- **Docker Compose** : https://docs.docker.com/compose/
- **Docker Hub** (images) : https://hub.docker.com/
- **Nginx** : https://nginx.org/en/docs/
- **Apache HTTPd** : https://httpd.apache.org/docs/
- **Tomcat** : https://tomcat.apache.org/
- **PostgreSQL** : https://www.postgresql.org/docs/
- **MySQL** : https://dev.mysql.com/doc/
- **Elasticsearch** : https://www.elastic.co/guide/en/elasticsearch/reference/current/index.html
- **Logstash** : https://www.elastic.co/guide/en/logstash/current/index.html
- **Kibana** : https://www.elastic.co/guide/en/kibana/current/index.html

6.2. Exemples et tutoriels

- **Awesome Compose** : https://github.com/docker/awesome-compose (exemples officiels)
- **Votre cours Docker** : Tout est dedans ! Relisez les chapitres :
  - Docker Compose
  - Images Docker
  - Volumes
  - Réseaux
  - Sécurité et Production

6.3. Outils

- **Trivy** (scanning) : https://github.com/aquasecurity/trivy
- **Goss** (testing) : https://github.com/goss-org/goss
- **Updatecli** (dépendances) : https://www.updatecli.io/
- **k6** (load testing) : https://k6.io/
- **Prometheus** : https://prometheus.io/
- **Grafana** : https://grafana.com/

7. FAQ

Q: Peut-on utiliser Docker Swarm au lieu de Docker Compose ?
R: Non, l'objectif est d'utiliser Docker Compose. Docker Swarm est hors périmètre pour ce projet.

Q: Doit-on obligatoirement utiliser la stack ELK ? Peut-on utiliser Loki/Grafana à la place ?
R: La stack ELK est **obligatoire** pour la partie principale. Vous pouvez ajouter Loki/Grafana en complément valorisé si vous le souhaitez.

Q: Peut-on utiliser une application existante (Jenkins, Nexus, etc.) au lieu de créer notre propre WAR ?
R: Oui, tant que l'application :

- Est déployée comme un WAR sur votre serveur d'application
- Se connecte à la base de données

- Est fonctionnelle et accessible

Q: Les compléments valorisés sont-ils obligatoires pour avoir 10/10 ?
R: Non. Si vous obtenez 10/10 sur les critères principaux, vous n'avez pas besoin de compléments.
Les compléments servent à **compenser** les points manquants et à démontrer des compétences avancées.

Q: Peut-on faire plus de 10/10 sur la Partie 1 ?
R: Non, la note est plafonnée à 10/10.

Q: Combien de compléments faut-il faire ?
R: Il n'y a pas de nombre imposé. Privilégiez la **qualité** :

- 1-2 compléments **bien faits** > 4 compléments **bâclés**

Q: Peut-on proposer un complément non listé ?
R: Oui, mais **demandez validation** via une issue GitLab avant de vous lancer. Proposez :

- Description du complément
- Valeur pédagogique
- Estimation du temps nécessaire

Q: Les images Docker doivent-elles être publiées sur Docker Hub ?
R: Ce n'est pas obligatoire mais c'est un plus (bonne pratique professionnelle).
Si vous publiez vos images :

- Utilisez un registry public (Docker Hub, GHCR, etc.)
- Documentez les tags dans le README
- Ajoutez les URLs dans le `compose.yml`

Q: Doit-on implémenter HTTPS avec certificat SSL/TLS ?
R: Ce n'est pas obligatoire mais valorisé (peut compenser des points manquants).
Si vous implémentez HTTPS :

- Utilisez Let's Encrypt avec Caddy (le plus simple)

- Ou configurez Nginx avec des certificats auto-signés
- Documentez la configuration dans le README

Q: Peut-on travailler sur plusieurs branches en parallèle ?
R: Oui, c'est même recommandé ! Utilisez des branches pour :
- Chaque fonctionnalité (`feature/elk-stack`)
- Chaque complément (`feature/prometheus-grafana`)
- Chaque correction (`fix/nginx-config`)

Q: Que faire si on est bloqué ?
R: **Demandez de l'aide !** Via :
- Issue GitLab (pour les questions techniques)
- Merge Request (pour une review intermédiaire)
- Email (en dernier recours)

Ne restez pas bloqué 2 jours sur un problème, demandez après 1-2 heures de recherche.

Q: La date limite (4 janvier 2026) inclut-elle les heures ?
R: Oui, la deadline est **4 janvier 2026 à 23h59** (minuit).
Tous les commits après cette heure seront ignorés.

Astuce : Ne vous y prenez pas au dernier moment ! Visez le **31 décembre** pour terminer, cela vous laisse une marge.

Q: Peut-on modifier le projet après la deadline pour corriger des bugs ?
R: Non. Seuls les commits **avant** le 4 janvier 2026 à 23h59 seront évalués.

8. Conclusion

Ce projet est une opportunité de **mettre en pratique** tout ce que vous avez appris dans ce cours et de vous **préparer** aux pratiques DevOps professionnelles.
Nos attentes :
- Une architecture **fonctionnelle** et **bien conçue**

- Une documentation **complète** et **professionnelle**
- Des choix techniques **justifiés**
- Une communication **active** (MR, Issues)
- Du **travail d'équipe** (binôme)

Rappel de notre philosophie :
L'objectif est de vous **préparer** à l'entreprise, pas de vous **piéger** !

- Posez des questions
- Demandez des clarifications
- Sollicitez des reviews intermédiaires
- Communiquez via GitLab

Nous sommes là pour vous aider à réussir !

**Bon courage et amusez-vous bien ! **

Contact :

- Bruno Verachten : gounthar@gmail.com
- GitLab : @gounthar

Ressources :

- Repository du cours : https://github.com/gounthar/cours-devops-docker
- Slides du cours : https://gounthar.github.io/cours-devops-docker/