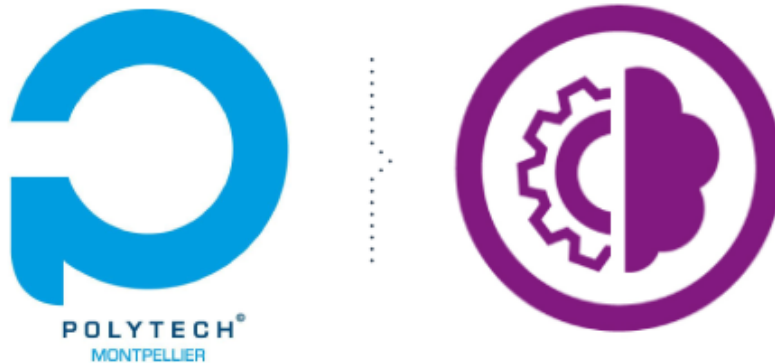


Année Universitaire 2024-2025

INDUSTRIALISATION CI/CD

Rapport - Projet CI/CD



Réalisé par
FRACHE Axel
DESPAUX Noa
SOULET Liam
D03

Sommaire

Sommaire	1
Résumé	1
I. Développement de l'API Web	3
II. Tests automatisés	4
III. Intégration continue (CI)	5
1. Analyse des triggers	6
2. Analyse du job verify	7
2.1. Environnement et configuration de base	7
2.2. Téléchargement et exécution de ktlint	7
2.3. Compilation et tests avec Maven	8
2.4. Gestion du cache Maven	8
2.5. Archivage des résultats de tests	8
IV. Déploiement et Orchestration (CD)	9
1. Workflow GitHub Actions	9
2. Construction de l'image Docker	9
3. Déploiement avec Helm et K3s	10
V. Monitoring	11
1. Endpoints exposés avec Spring Boot Actuator	11
2. Intégration de Prometheus	12
1.1. Configuration des cibles	12
1.2. Annotations Kubernetes	12
3. Visualisation avec Grafana	13
2.1. Déploiement	13
2.2. Ajout de la source de données	13

Résumé

Ce rapport documente la mise en place d'un pipeline CI/CD complet pour une API Web développée avec Kotlin et Spring Boot. L'objectif principal de ce projet était d'automatiser les processus de test, d'intégration, de déploiement et de monitoring afin d'améliorer la qualité et la rapidité de mise en production de l'application.

Le rapport détaille les différentes étapes du pipeline, incluant le développement de l'API, les tests automatisés, l'intégration continue avec GitHub Actions, le déploiement continu avec K3s et Helm, ainsi que le monitoring avec Spring Boot Actuator, Prometheus et Grafana.

Le code source de ce projet est disponible sur GitHub : [Dépôt GitHub](#)

I. Développement de l'API Web

Dans le cadre de ce projet de mise en place d'un pipeline CI/CD, la première étape a été le développement d'une API Web. Notre équipe a choisi Kotlin pour implémenter cette API, en raison de sa compatibilité avec la JVM. Bien que nous possédions une expertise en Java, l'adoption de Kotlin nous a permis d'explorer un langage moderne présentant des similarités avec Java. Le framework Spring Boot a été utilisé pour faciliter le développement de l'API, grâce à son intégration avec Kotlin.

L'API communique avec une base de données PostgreSQL, qui assure le stockage des données relatives aux villes. Le déploiement de cette base de données est automatisé via Docker Compose, en utilisant la dernière version disponible de PostgreSQL. La configuration de la base de données est gérée à l'aide de variables d'environnement, ce qui offre une flexibilité accrue pour les différents paramètres.

L'API permet de gérer un catalogue de villes avec des endpoints simples :

- `POST /city`, qui permet d'ajouter une ville à la base de données.
- `GET /city`, qui permet de récupérer la liste des villes.
- `GET /_health`, qui permet de vérifier l'état de l'API.

II. Tests automatisés

Afin de garantir la qualité de notre code, l'intégration de tests automatisés est essentielle. Nous avons choisi de mettre en place des tests de non-régression. Ces tests permettent de vérifier que les fonctionnalités fondamentales de l'API restent opérationnelles après toute modification du code.

Nous avons donc développé trois tests, chacun couvrant un endpoint spécifique de l'API. Pour réaliser ces tests, nous avons utilisé la bibliothèque MockMvc. Cet outil nous a permis de simuler les requêtes HTTP vers l'API et de valider les réponses obtenues. Par exemple, le test de l'endpoint /health consiste à simuler un appel à cet endpoint et à vérifier que le code de statut de la réponse correspond à la valeur attendue.

Il est important de noter que les tests d'ajout de ville simulent uniquement l'appel à l'API. L'ajout de la ville dans la base de données n'est pas effectué lors du test. Nous nous concentrons sur la validation de la réponse de l'API pour nous assurer de son bon fonctionnement.

III. Intégration continue (CI)

L'intégration continue (CI) joue un rôle crucial dans ce projet. Elle permet non seulement de détecter rapidement les erreurs et les failles potentielles introduites par les modifications du code, mais aussi d'automatiser la construction et la publication de l'image Docker de l'application.

Il est important de noter que la construction et la publication de l'image Docker peuvent être considérées comme une étape à la frontière entre CI et CD. Dans le contexte de ce projet, nous les incluons dans la CI car elle est étroitement liée au processus de validation du code.

Notre pipeline CI est définie dans un fichier workflow ci.yml et se déclenche à chaque push vers le dépôt. Les étapes principales de ce workflow sont les suivantes :

- Définition des fichiers à ignorer (README.md, .gitignore, les fichiers de documentation, etc.) et des trigger actions (push, pull request) pour les autres étapes de la CI
- Checks du code (set up du JDK, tests, lint)
- Build and push l'image Docker du projet et scan de vulnérabilités

Bien que la troisième étape (construction et publication de l'image Docker) puisse être considérée comme faisant partie du CD, nous la traiterons dans la section CI de ce rapport, car elle est intégrée au processus de validation du code. La section CD se concentrera principalement sur le déploiement de cette image dans un environnement d'exécution.

1. Analyse des triggers

Comme on peut l'observer, la pipeline CI se déclenche lorsqu'un collaborateur pousse des changements, sur n'importe quelle branche, ou bien lorsqu'une pull request est merged à la branche main.

```
on:|
  push:
    branches:
      - '**'
    paths-ignore:
      - '**/README.md'
      - '**/*.md'
      - 'docs/**'
      - '.gitignore'
      - '.editorconfig'
    tags:
      - 'v[0-9]+.[0-9]+.[0-9]+'
  pull_request:
```

Les fichiers ne concernant pas le code sont automatiquement ignorés, comme le fichier README.md, les fichiers markdown de manière générale, les fichiers de documentation, le fichier .gitignore, etc. par souci de performance.

2. Analyse du job verify

Le job `verify` a pour but de valider la qualité du code et la fiabilité du projet à chaque changement significatif dans le dépôt. Il exécute une vérification du style de code Kotlin via `ktlint`, compile le projet avec Maven, exécute les tests automatisés et, en toutes circonstances, et conserve les résultats des tests pour une analyse ultérieure.

2.1. Environnement et configuration de base

Le job s'exécute sur un environnement Linux standard (`ubuntu-latest`) et utilise **JDK 21 (Temurin)**, cohérent avec les pratiques modernes pour les projets Spring Boot ou Kotlin. Le cache Maven est activé pour améliorer les performances, soit via le paramètre `cache: maven` intégré dans l'action `setup-java`, soit via une étape manuelle plus personnalisée.

2.2. Téléchargement et exécution de `ktlint`

Afin de garantir la qualité du code Kotlin, l'outil `ktlint` est utilisé en version 1.0.1. Son binaire est téléchargé localement dans le répertoire du projet, puis mis en cache avec GitHub Actions (`actions/cache`) afin d'éviter de le retélécharger à chaque exécution de pipeline.

Lorsque le cache est manquant ou obsolète, le binaire est automatiquement récupéré depuis le dépôt GitHub officiel de `ktlint`, puis rendu exécutable.

L'analyse de style est lancée sur tous les fichiers `.kt` contenus dans le répertoire `src`. En cas d'infractions aux règles de style, le job échoue immédiatement, et un message personnalisé est affiché dans les logs de la CI.

2.3. Compilation et tests avec Maven

Une étape essentielle de ce job consiste à compiler et tester l'application via Maven. La commande utilisée est `mvn verify`, ce qui implique non seulement la compilation, mais aussi l'exécution de tous les tests unitaires et d'intégration, ainsi que la vérification finale du projet.

Quelques paramètres spécifiques sont appliqués :

- `-T 2C` : active le parallélisme avec deux threads par cœur CPU, ce qui accélère considérablement le build.
- `-B` : supprime les interactions utilisateurs (mode batch), évitant les blocages en environnement CI.
- Le niveau de log des transferts Maven est limité pour rendre les sorties plus lisibles.
- `-Dmaven.test.failure.ignore=false` : garantit que toute erreur de test stoppe immédiatement l'exécution du pipeline.

2.4. Gestion du cache Maven

En plus du cache Maven intégré à `setup-java`, une étape spécifique est ajoutée pour gérer manuellement le cache du dossier `.m2`. Cela permet de restaurer les dépendances plus rapidement si le cache par défaut est insuffisant ou si l'on souhaite personnaliser la stratégie de clé.

Le cache est basé sur un hash des fichiers `pom.xml` du projet. Cela garantit que toute modification de dépendance invalide le cache et force Maven à récupérer les artefacts mis à jour.

2.5. Archivage des résultats de tests

Quelle que soit l'issue des tests (succès ou échec), les résultats sont archivés automatiquement à l'aide de l'action `upload-artifact`. Les fichiers présents dans le répertoire `target/surefire-reports` sont stockés pendant 7 jours et peuvent être téléchargés depuis l'interface GitHub pour consultation.

Cela facilite grandement le diagnostic lors d'un échec de test, notamment si les logs ne sont pas immédiatement visibles ou si l'analyse se fait a posteriori.

IV. Déploiement et Orchestration (CD)

Le déploiement de l'application repose sur une chaîne d'intégration et de déploiement continu (CI/CD) associée à l'orchestration avec K3s et Helm. Cette stratégie permet une mise en production rapide, reproductible et scalable de l'API.

1. Workflow GitHub Actions

On utilise un workflow **CI** (**ci.yml**) : déclenché à chaque **push**, qui effectue les actions suivantes :

- Lint du code
- Exécution des tests
- Build de l'image Docker


L'automatisation CI/CD garantit que chaque nouvelle version du code est immédiatement testée, packagée, puis prête à être déployée.

2. Construction de l'image Docker

L'image Docker utilise un build multi-étapes :

1. Compilation de l'application via Maven
2. Création d'une image légère à partir d'Amazon Corretto pour l'exécution

Cette image est ensuite taguée et poussée sur Docker Hub via la commande suivante :



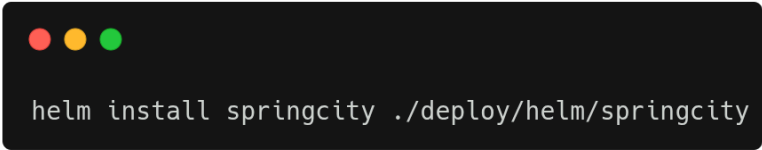
```
docker build -t axelfrache/city-api:latest .  
docker push axelfrache/city-api:latest
```

3. Déploiement avec Helm et K3s

Une fois l'image publiée, le déploiement est réalisé à l'aide d'un **chart Helm** personnalisé. Ce chart définit :

- Le déploiement de l'API (`deployment.yaml`)
- Le service exposant l'API (`service.yaml`)
- Le déploiement de PostgreSQL (`postgres.yaml`)
- Un fichier `values.yaml` pour centraliser la configuration (ports, variables d'environnement, etc.)

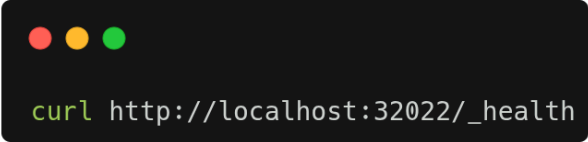
La commande suivante installe l'application sur le cluster local K3s :



```
helm install springcity ./deploy/helm/springcity
```

Le port 32022 est exposé via un service **NodePort**, permettant d'accéder à l'API via <http://localhost:32022>.

Pour tester que notre application fonctionne correctement sur K3s on peut exécuter la commande suivante :



```
curl http://localhost:32022/_health
```

V. Monitoring

Le monitoring constitue un pilier essentiel dans tout système moderne, en particulier lorsqu'il est déployé en production ou en environnement distribué. Dans ce projet, nous avons mis en place une stack complète de supervision composée de trois éléments complémentaires :

- **Spring Boot Actuator**, pour exposer les métriques de l'application
- **Prometheus**, pour collecter les métriques exposées
- **Grafana**, pour visualiser et interpréter les données de performance

Cette architecture permet une observabilité complète du système : du code applicatif jusqu'aux ressources système.

1. Endpoints exposés avec Spring Boot Actuator

Spring Boot Actuator est une bibliothèque qui expose des endpoints REST permettant d'obtenir des informations sur la santé, les métriques et le comportement interne de l'application. Nous avons intégré deux dépendances dans le `pom.xml` :

- `spring-boot-starter-actuator`
- `micrometer-registry-prometheus`

Elles permettent d'exposer automatiquement des endpoints au format Prometheus :

Endpoint	Rôle
<code>/actuator/health</code>	Vérifier l'état global de l'application
<code>/actuator/metrics</code>	Afficher toutes les métriques disponibles
<code>/actuator/prometheus</code>	Exporter les métriques dans un format compatible Prometheus

En complément, nous avons défini dans `application.properties` :

```
management.endpoints.web.exposure.include=health,metrics,prometheus
management.endpoint.health.show-details=always
```

2. Intégration de Prometheus

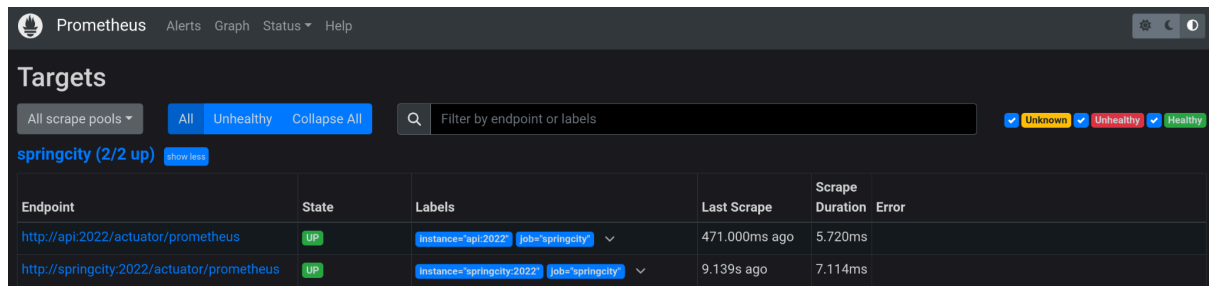
Prometheus a été intégré à l'architecture via un service Docker dans le fichier `docker-compose.yml`. Il est configuré pour interroger l'API toutes les 15 secondes via l'endpoint `/actuator/prometheus`.

1.1. Configuration des cibles

Le fichier `prometheus.yml` contient :

```
scrape_configs:
  - job_name: 'springcity'
    metrics_path: '/actuator/prometheus'
    static_configs:
      - targets:
        - 'api:2022'
        - 'springcity:2022'
```

Prometheus utilise un modèle "pull" : il scrape les métriques à intervalles réguliers. Grâce à cette approche, il peut détecter automatiquement les problèmes (latence, erreurs, charge CPU, etc.) et générer des alertes.



Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://api:2022/actuator/prometheus	UP	instance="api:2022" job="springcity"	471.000ms ago	5.720ms	
http://springcity:2022/actuator/prometheus	UP	instance="springcity:2022" job="springcity"	9.139s ago	7.114ms	

1.2. Annotations Kubernetes

Pour intégrer Prometheus à notre cluster K3s, nous avons ajouté des annotations sur les pods Kubernetes :

```
annotations:
  prometheus.io/scrape: "true"
  prometheus.io/port: "2022"
  prometheus.io/path: "/actuator/prometheus"
```

Ces annotations facilitent la découverte automatique des endpoints à monitorer dans un environnement Kubernetes.

3. Visualisation avec Grafana

Grafana complète la stack de monitoring en fournissant une interface graphique riche et interactive pour explorer les métriques collectées.

2.1. Déploiement

Grafana est configuré dans le même `docker-compose.yml`, avec un volume persistant et des identifiants admin par défaut. L'interface est accessible via `http://localhost:3000`.

2.2. Ajout de la source de données

Dans l'interface Grafana :

- Source : **Prometheus**
- URL : `http://prometheus:9090`
- Scrape interval : `15s`

Il nous est ensuite possible de créer des dashboards pour avoir une vue d'ensemble sur l'application, visualiser les performances ou encore obtenir des métriques système.