Ateliers

Gitlab CI/CD

Pré-requis :

- Bonne connexion Internet
- Environnement Linux de préférence
- Minimum 16Go de RAM
- ***Docker***
- ***git*** et ***gitk***
- JDK21
- Un serveur gitlab avec des runners docker
- et optionnellement un accès à un cluster Kubernetes (*kind, kubectl*)

Table des matières

Atelier 1 : Démarrage

Objectifs : Premier accès, parcours de l'interface utilisateur, mise en place clé ssh

1.1 Accès administrateur et création de son compte

Accèder à : ***gitlab.formation.org*** avec ***:***

- username : **root**
- password : **/Welcome1**

Créer vous un compte et définissez vous un mot de passe.

1.2 Mise en place clé ssh

Commencer par générer une clé ssh sur votre poste :

```
ssh-keygen -t rsa -b 2048 -C "<comment>"
```

Appuyer sur Entrée à chaque question de l'assistant sans préciser de Pass Phrase. A la fin 2 fichiers ont été créés dans le répertoire ***~/.ssh*** :

- ***id_rsa*** : La clé privé. Ne doit pas jamais transiter par le réseau
- ***id_rsa.pub*** : La clé publique. Elle peut être fournie à des tiers

Se connecter avec votre compte et accéder à *Preferences → SSH Keys* et ajouter une nouvelle clé en copiant/collant le contenu de votre clé public

Tester en commande en ligne avec :

```
ssh -T git@gitlab.formation.org
```

Atelier 2: Workflow de collaboration

Objectifs :

- Comprendre les différents acteurs accédant aux projet
- Visualiser le support pour la planification et le suivi de tâches
- Comprendre le workflow de résolution d'issues

2.1 Groupes / Projets / Membres

Avec votre compte,

- Création d'un groupe de projet à votre nom et affecter des membres parmi les stagiaires avec des rôles *developer*
- Création d'un projet privé nommé ***delivery-service***, en initialisant un dépôt. (Présence d'un fichier *README*)

Parcourir les menus du projet ***delivery-service,*** en particulier ***settings***

2.2 Milestones, Issues, Labels, Tableaux de bord

Mise en place des labels, Milestone, et tableaux de bord
En tant que mainteneur de projet, créer 2 milestones :

- **Sprint1**
- **Sprint2**

Au niveau groupe, définir les labels suivants :

- **In progress**
- **Review**

Définir ensuite un tableau de bord ajoutant des colonnes pour les 2 labels précédents

Au niveau projet, utiliser les labels par défaut de Gitlab +

- **API**
- **DevOps**

Création d'issues
Avec le compte *reporter*

- Saisir plusieurs issues dont une s'appelant : « *CRUD pour delivery-service* »
- Tagger avec *API*

Discussion sur une issue entre Développeur/Mainteneur projet :

- Saisir quelques commentaires
- Saisir quelques issues techniques :
  - « Mise en place pipeline CI »,
  - « Configuration repository », …

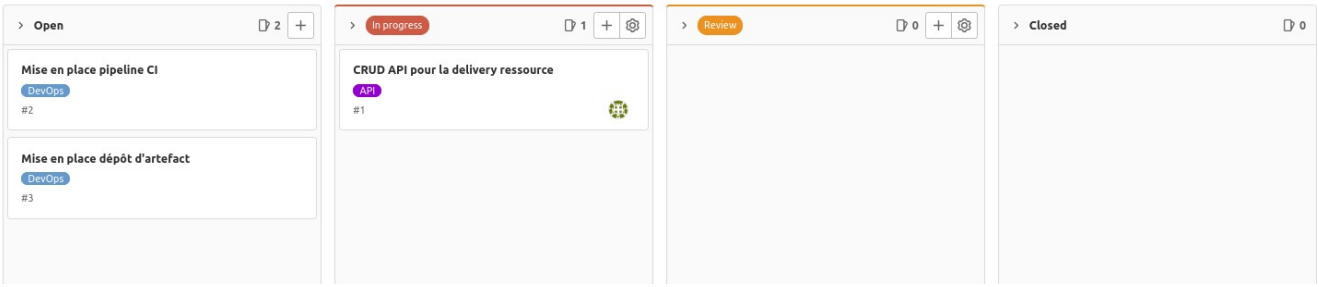
Les tagger avec DevOps

Avec le compte *owner/mainteneur :*

- mise au planning et affectation
- Tagger les issues

Avec le compte *developer*, accès au tableau de bord et déplacement du post-it «« *CRUD pour delivery-service* »
To Do -> In progress

A la fin de ces opérations, le tableau de bord pourra ressembler à ce qui suit :



2.3 Branche protégée

Vérifier que la branche main est votre branche par défaut. Si ce n’est pas le cas faites les opérations nécessaires.

En tant que mainteneur de votre projet configurer une branche protégée ***production*** avec les permissions suivantes :

- *Allow to Merge* : Mainteneur
- *Allow to push and merge* : No one
- *Force push* : Non

2.4 MergeRequest

1. Création de merge request à partir d’une issue

Commencer par créer une Issue '*CRUD pour delivery-service*',
A partir de cette, créer une *Merge Request et* l’affecter à un développeur
=> La merge request est préfixée par ***Draft*** et a pour effet de créer une branche portant le nom de l’issue

2. Mise en place environnement de développement + développement

En tant que développeur sur votre poste de travail :

- Récupérer la branche de la merge request :

```
git clone <url-ssh-depot>
git checkout <nom-de-branche>
```

- **Reprendre le code source fourni dans le répertoire TPs/2.3_MergeRequest/FirstCommit**

Construire l'application :

```
./mvnw clean package
```

Exécuter l'application :

```
java -jar application/target/delivery-service-0.0.1-SNAPSHOT.jar
```

Accéder à l'application :

http://localhost:8080/swagger-ui.html

3. Pousser les modifications

Le développeur pousse les modifications

```
git add .
git commit -m 'Implémentation CRUD'
git push
```

En tant que *developer* sur gitlab, supprimer le préfixe *Draft* de la MR

4. Revue de code

En tant que *owner/mainteneur*, faire une revue de code et refuser la fusion.

5. Compléments de développement et maj dépôt

Reprise du code fourni dans le répertoire TPs/2.3_MergeRequest/SecondCommit

Exécuter les tests et s'assurer qu'ils passent :

```
./mvnw test
```

Push les modifications vers gitlab

6. Accepter la MR

En tant que *Owner/Mainteneur* faire une revue de code

Accepter le Merge Request, supprimer la branche et éventuellement un *« squash commit »*

7. Nettoyage local

En local, en tant que développeur supprimer la branche locale et exécuter

```
git  remote prune origin
```

Ateliers 3 : Syntaxe .gitlab-ci.yml

3.1 Stages et Job

Mettre en place un fichier ***.gitlab-ci.yml*** qui effectue une phase de compilation et de tests unitaires.

La pipeline peut utiliser l'image : *openjdk:21-jdk-oracle*

Visualise le tag du runner disponible et déclarer ce tag dans la pipeline

La commande pour compiler est :

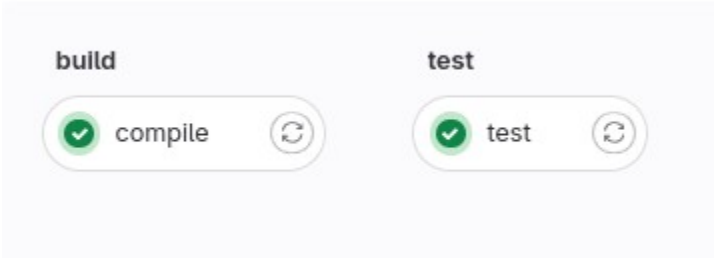
`./mvnw $MAVEN_OPTS compile`

Pour tester :

`./mvnw $MAVEN_OPTS test`

Les options maven suivantes peuvent être positionnées en une variable MAVEN_OPTS:

```
-Dhttps.protocols=TLSv1.2 -Dmaven.repo.local=$CI_PROJECT_DIR/.m2/repository
-Dorg.slf4j.simpleLogger.log.org.apache.maven.cli.transfer.Slf4jMavenTransferListener=WARN
-Dorg.slf4j.simpleLogger.showDateTime=true
-Djava.awt.headless=true
```



Observer l'exécution de la pipeline, les dépendances sont elles cachées ?

3.2 Cache, Artefact et dependencies

Modifier la pipeline afin qu'elle intègre 3 phases :

- 1 phase de **packaging** stockant l'artefact généré :
  Commande Maven :

  `./mvnw clean package`

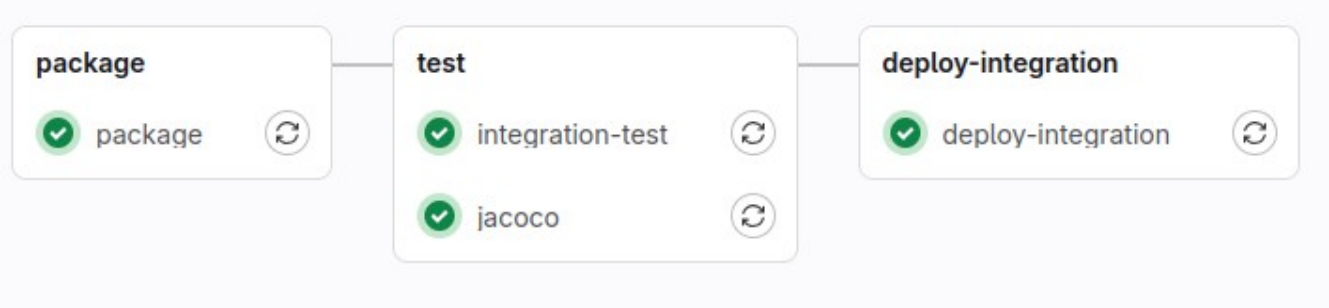
  L'artefact à uploader se trouve alors dans le répertoire target

- Une phase **de test et analyse** incluant 2 jobs :
  - Une calcul de la couverture de test, pour cela on a pas besoin de l'artefact précédent
    `./mvnw test`

    Le calcul de couverture présent dans le répertoire ***target/site/jacocco*** est exposé dans les merge request sous le nom de 'Rapport Jacocco'

  - Un job démarrant l'application construite précédemment et exécutant des tests avec curl.

- Une phase de **déploiement** récupérant l'artefact généré et le copiant dans une arborescence variabilisée. La variable est renseignée dans l'interface Gitlab. Utiliser l'exécuteur shell pour ce job



Vous pouvez partir du fichier .gitlab-ci.yml fourni **TPs/3.3_CacheArtifactsDependencies**

3.3 Conditions et GITLAB_STRATEGY

Pour les job ***deploy-integration*** *et* ***integration-test*** s'assurer que le dépôt n'est pas cloné en positionnant la variable *GIT_STRATEGY*

S'assurer que le job ***deploy-integration*** ne s'exécute que si l'on est sur une branche protégée et lors d'une approbation de MR

Interdire l'exécution simultanée de 2 pipelines sur des commits différents

3.4 Gabarits et inclusions

3.4.1 Gabarit de pipeline complète

Créer un nouveau projet nommé ***maven-ci-template***

Ajouter un fichier ***maven.yml*** qui reprend toutes les phases définies dans la pipeline précédente

Commiter dans la branche main.

Dans le projet ***delivery-service***, créer une nouvelle branche ***template*** modifier le fichier *.gitlab-ci.yml* afin qu'il utilise le gabarit précédent et positionne la variable ***APP_JAR***

Visualiser les gabarits proposés par Gitlab

3.4.2 Inclusion du job fourni par Gitlab

Reprendre le tronc principal et inclure les job d'analyse de sécurité :

```
include:
  - template: Security/SAST.gitlab-ci.yml
  - template: Security/Secret-Detection.gitlab-ci.yml
```

3.5 Intégration docker

3.5.1 Utilisation de service

Reprendre la branche principale. Modifier le job de test intégration en :

- Démarrant l'application SpringBoot en tant que service. Vous pouvez utiliser l'image ***dthibau/delivery-service:0.0.1-SNAPSHOT***
- Utiliser l'image ***dthibau/jmeter*** pour pouvoir exécuter des tests fonctionnels.
  La commande pour exécuter les tests est :

```
jmeter -n -JSERVEUR=delivery-service -t
src/test/jmeter/Fonctionnel.jmx -l results.jtl -e -o /jmeter-
report
```

Le rapport des tests fonctionnels se trouve alors dans **/jmeter-report**

Vous pouvez partir du fichier fourni dans **Tps/3.5.1_ServiceImage**

3.5.2 Construction d'une image docker et publication dans un registre

Modifier le fichier *.gitlab-ci.yml* en ajoutant un job utilisant l'image ***docker*** :

- Déplacement du fichier jar créé au job précédent à la racine :

  ```
  mv target/$APP_JAR .
  ```

- Construit une image docker préfixée par le dépôt interne de Gitlab:

  ```
  docker build -t $CI_REGISTRY/$CI_PROJECT_NAMESPACE/delivery-
  service:$CI_COMMIT_BRANCH .
  ```

- Se connecte sur dockerHub

  ```
  docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD
  $CI_REGISTRY
  ```

- Pousser l'image vers DockerHub

  ```
  docker push $CI_REGISTRY/$CI_PROJECT_NAMESPACE/delivery-service:
  $CI_COMMIT_BRANCH
  ```

Utiliser cette image dans les tests d'intégration

3.6 Mise en place d'environnements

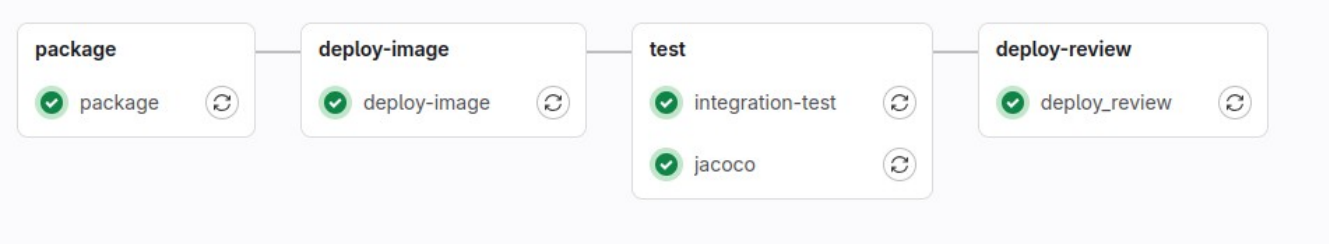
Reprendre la pipeline précédente

- Définir des jobs de déploiements dans les environnements suivants :

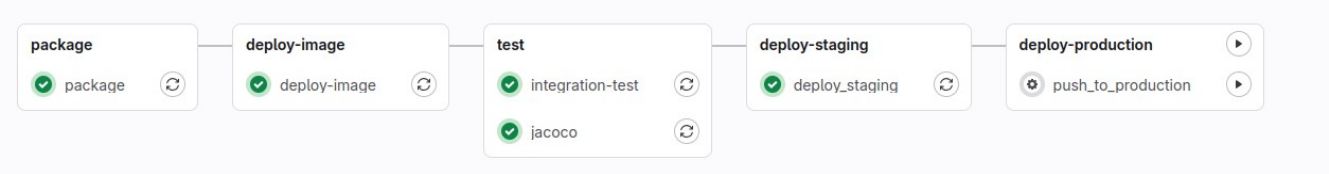
- Environnement de revue, nom dynamique reprenant le nom de branche
- Environnement de staging, réservé à la branche main, le déploiement est automatique
- Environnement de production, réservé à la branche main, le déploiement est manuel, il s'effectue en 2 jobs :

- 1 job qui reprend l’état de main et pousse dans la branche production
- 1 job qui ne s’exécute que sur la branche production et qui déploie

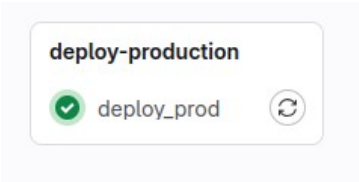
Pipeline sur une branche non protégée ET non déclenché par une MR



Pipeline sur la branche main



Pipeline sur la branche de production



Scripts *deploy-review*

```
- echo "Deploy a review app"
- env
- echo $CI_COMMIT_BRANCH
- docker stop $CI_COMMIT_BRANCH || true
- docker rm $CI_COMMIT_BRANCH || true
- docker run -d -p 80$CI_PIPELINE_ID:8080 --name $CI_COMMIT_BRANCH
$CI_REGISTRY/$CI_PROJECT_NAMESPACE/delivery-service:$CI_COMMIT_BRANCH
```

Scripts *push-to-production*
Nécessite la création d’un token production-token ayant les droits de lecture et d’écriture sur le repository créé par le mainteneur de projet et d’une variable PUSH_TOKEN contenant la valeur du jeton

```
- git config --global user.email "ci-bot@example.com"
- git config --global user.name "CI Bot"
- git branch
- git remote -v
- git checkout production  # Bascule sur la branche de production
- git merge origin/main  # Fusionne les changements de la branche
main
- git push
http://production-token:$PUSH_TOKEN@$CI_SERVER_HOST/$CI_PROJECT_PATH.
git production  # Pousse vers la branche protégée production
```

Script de deploy-production

```
- echo "Deploy to production server"
- docker tag $CI_REGISTRY/$CI_PROJECT_NAMESPACE/delivery-service:main
$CI_REGISTRY/$CI_PROJECT_NAMESPACE/delivery-service:production
- docker stop production || true
- docker rm production || true
- docker run -d -p 8080:8080 --name $CI_COMMIT_BRANCH
$CI_REGISTRY/$CI_PROJECT_NAMESPACE/delivery-service:production
```

Vérifier les environnements créés dans l'interface gitlab

3.7 Packaging et Release

3.7.1 Gitlab Registry (Maven)

Ajouter un fichier ***settings.xml*** comme suit :

```
<settings>
 <servers>
  <server>
   <id>gitlab-maven</id>
   <configuration>
    <httpHeaders>
     <property>
      <name>Job-Token</name>
      <value>${env.CI_JOB_TOKEN}</value>
     </property>
    </httpHeaders>
   </configuration>
```

```
    </server>
  </servers>
</settings>
```

Ajouter dans le fichier ***pom.xml*** la configuration de déploiement

```
<repositories>
  <repository>
    <id>gitlab-maven</id>
    <url>${CI_API_V4_URL}/projects/${CI_PROJECT_ID}/packages/maven</url>
  </repository>
</repositories>
<distributionManagement>
  <repository>
    <id>gitlab-maven</id>
    <url>${CI_API_V4_URL}/projects/${CI_PROJECT_ID}/packages/maven</url>
  </repository>
  <snapshotRepository>
    <id>gitlab-maven</id>
    <url>${CI_API_V4_URL}/projects/${CI_PROJECT_ID}/packages/maven</url>
  </snapshotRepository>
</distributionManagement>
```

Ajouter un script dans le premier job de build :

```
./mvnw -s settings.xml deploy
```

3.7.2 Release

Ajouter un job de release ne s'exécutant que sur la branche production

```
release_job:
  stage: release
  image: registry.gitlab.com/gitlab-org/release-cli:latest
  rules:
  - if: $CI_COMMIT_TAG
  when: never # Ne pas exécuter si création manuelle de tag
  - if: $CI_COMMIT_BRANCH == 'production'
  script:
  - echo "running release_job for $TAG"
  release:
    tag_name: 'v0.$CI_PIPELINE_IID'
    description: 'v0.$CI_PIPELINE_IID'
```

```
    ref: '$CI_COMMIT_SHA' # tag créé à partir du SHA1.
```