

## TP 5 - Créer un projet CI/CD

### Objectifs :

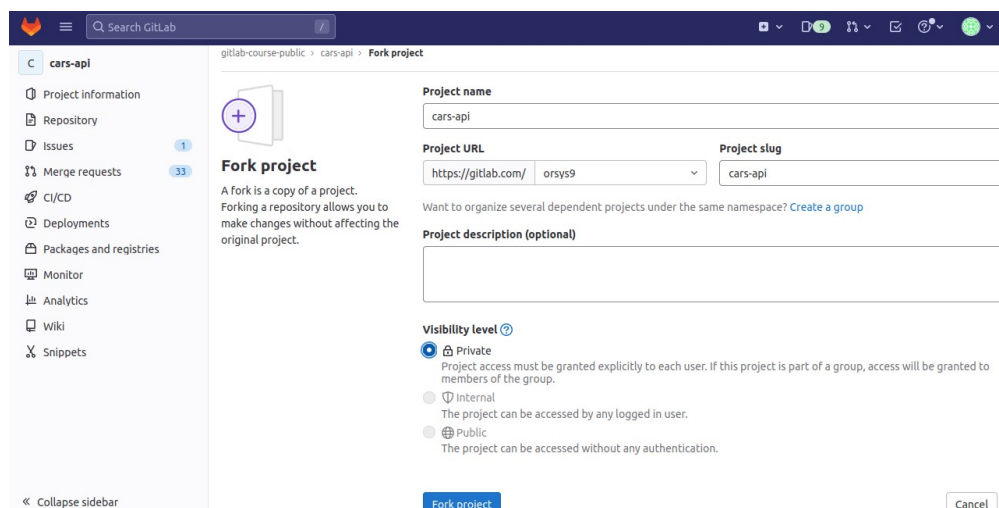
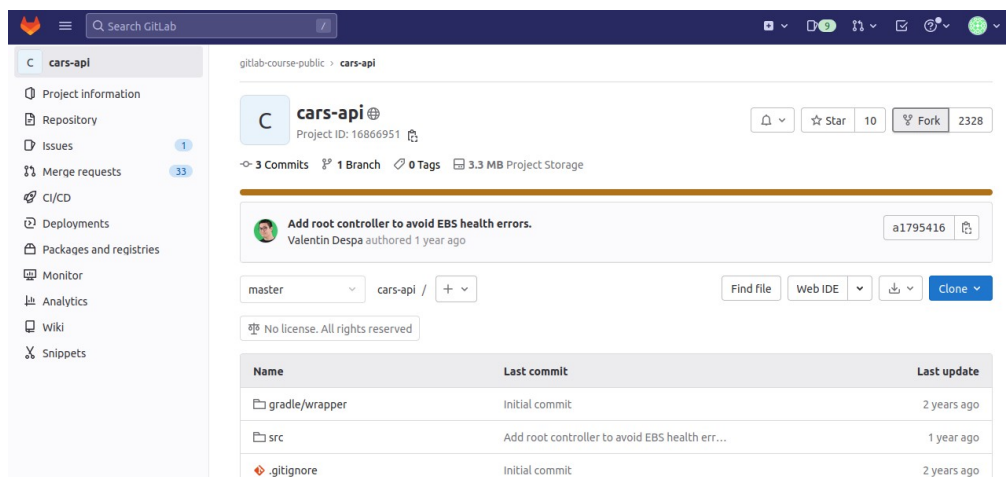
- Mise en place d'un pipeline CI/CD

### Création du projet et du runner spécifique :

- Nous allons commencer par un fork de dépôt suivant :

<https://gitlab.com/gitlab-course-public/cars-api>

Un fork est une copie d'un dépôt distant qui nous permet d'effectuer des modifications sans affecter le projet original.



2. Dans ce TP on va utiliser des runners spécifiques. Assurez-vous que les runners partagés sont désactivés.
3. Enregistrez un nouveau runner docker avec la commande suivante, en utilisant le token que vous devez le copier d'abord depuis l'interface du projet :

```
gitlab-runner register --non-interactive --url "https://gitlab.com/" --registration-token "PROJECT_REGISTRATION_TOKEN" --executor "docker" --docker-image alpine:latest --run-untagged="true"
```

- Modifiez le fichier de configuration ~/.gitlab-runner/config.toml de gitlab-runner :

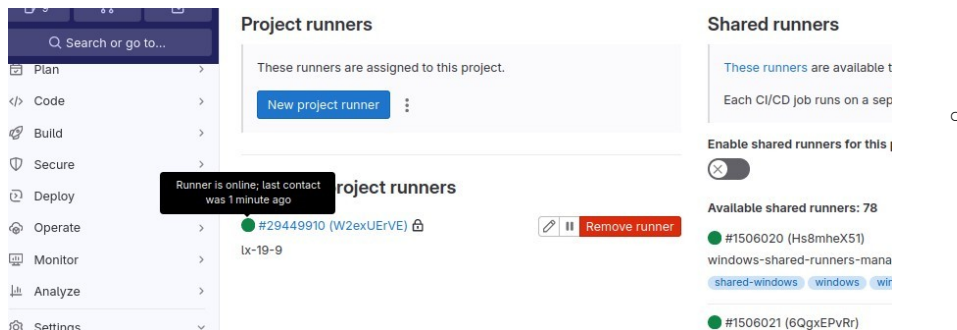
```
tls_verify = true
```

```
privileged = true
```

```
volumes = ["/certs/client", "/cache"]
```

- Vérifiez, avec la commande suivante, que le runner spécifique est bien créé et a contacté gitlab.com :

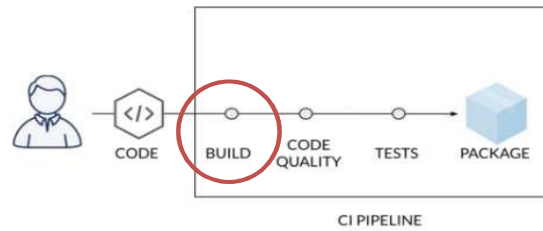
```
gitlab-runner verify
```



- Lancez les runners spécifiques avec la commande suivante puisqu'on les gère avec un utilisateur simple :

```
gitlab-runner run
```

## Stage de Build



#### 4. Créez un nouveau fichier **.gitlab-ci.yml**.

- Ajoutez le fichier **.gitlab-ci.yml** à la racine du projet, et y copiez le contenu suivant pour créer le job build sous le Stage de build.

*stages:*

*- build*

*build:*

*stage: build*

*image: openjdk:12-alpine*

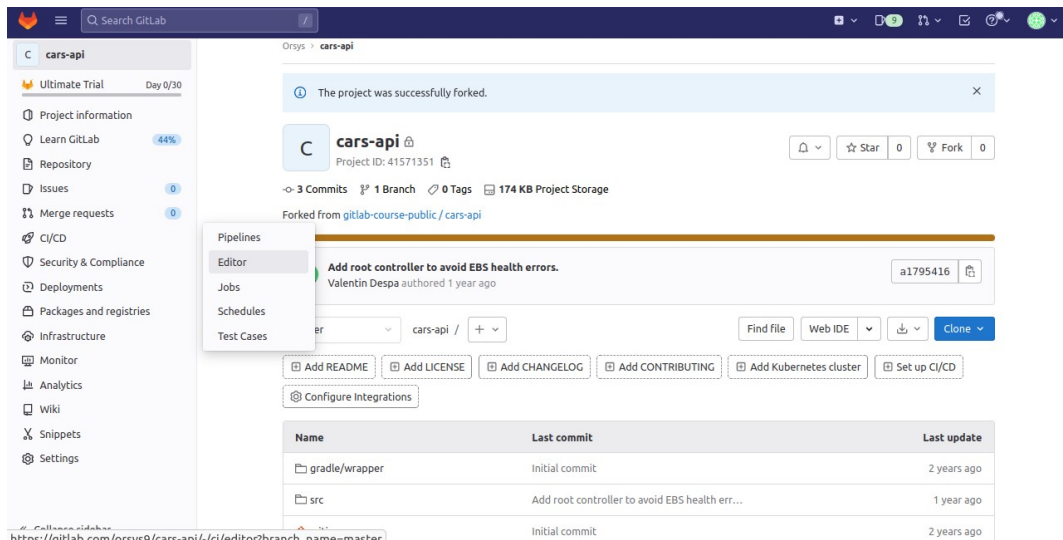
*script:*

*- ./gradlew build*

*artifacts:*

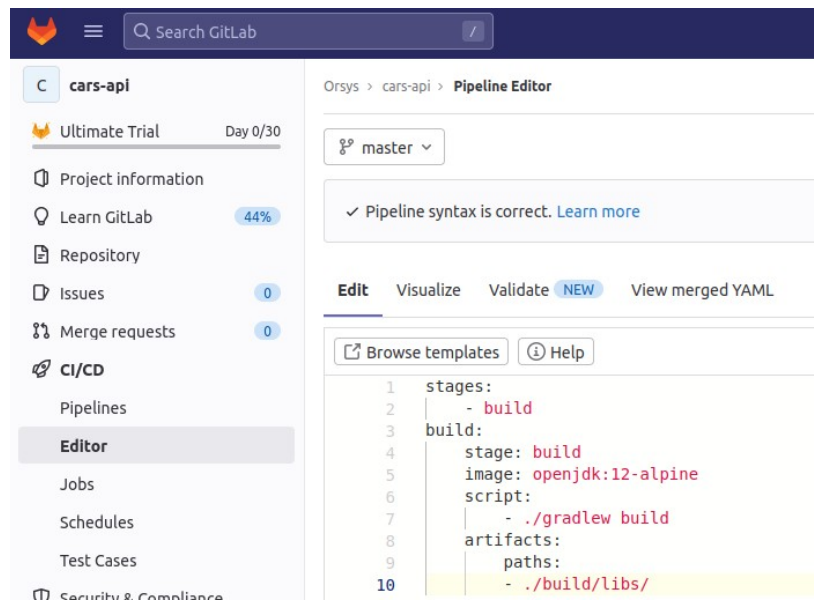
*paths:*

*- ./build/libs/*



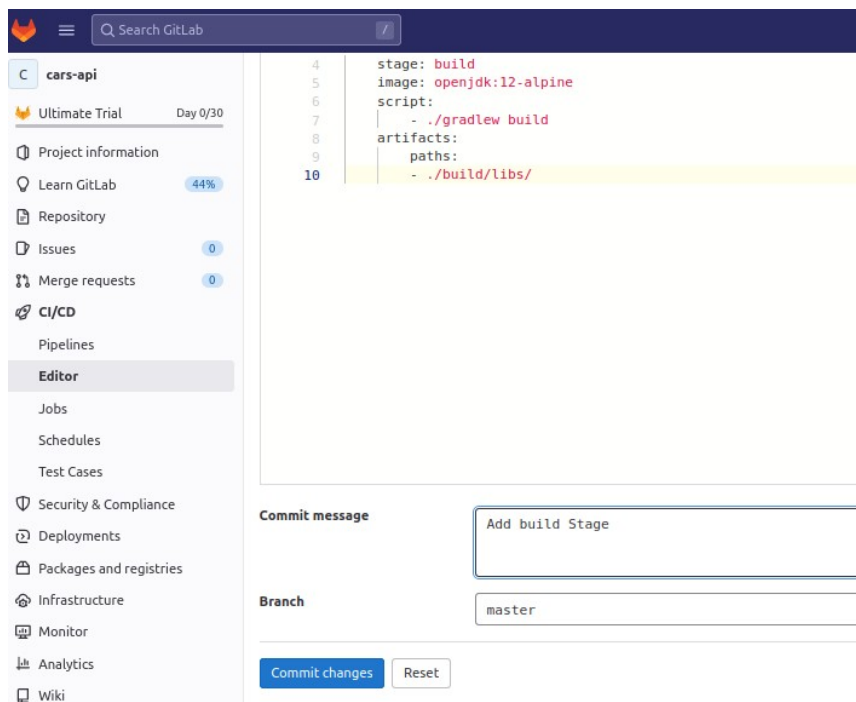
The screenshot shows the GitLab web interface for a project named "cars-api". The interface includes a sidebar with navigation links, a main content area with project information, and a table of commits. A "Pipelines" dropdown menu is open, showing options like "Editor", "Jobs", "Schedules", and "Test Cases". The "Jobs" option is selected, showing a job named "Add root controller to avoid EBS health errors".

Name	Last commit	Last update
gradle/wrapper	Initial commit	2 years ago
src	Add root controller to avoid EBS health err...	1 year ago
	Initial commit	2 years ago



Dans ce *stage*, on utilise *gradle* pour compiler notre projet **api-cars**. Les *artifacts* de ce *stage* seront sauvegardés dans *./build/libs*

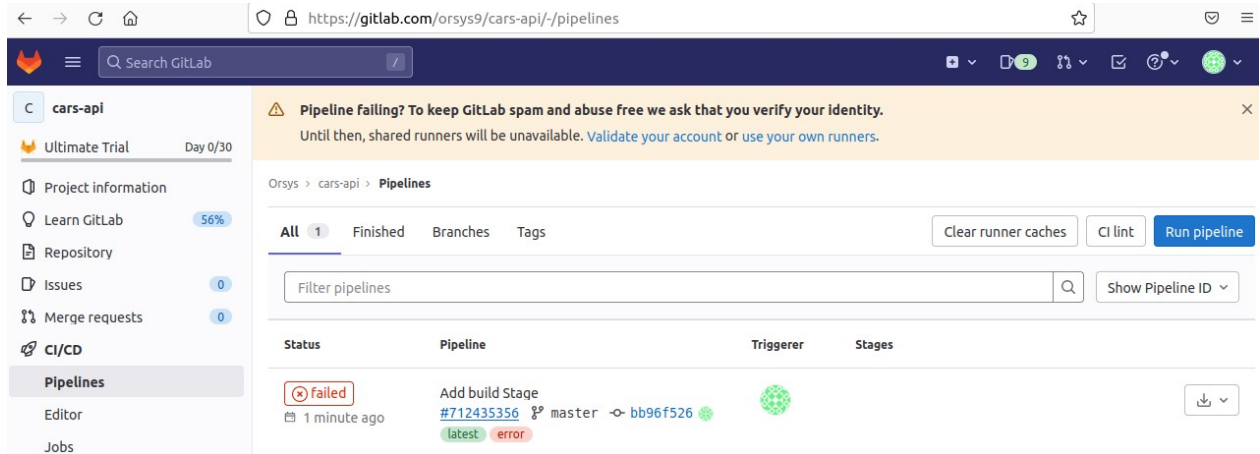
- Créez un nouveau commit avec le message suivant : « Add build Stage »



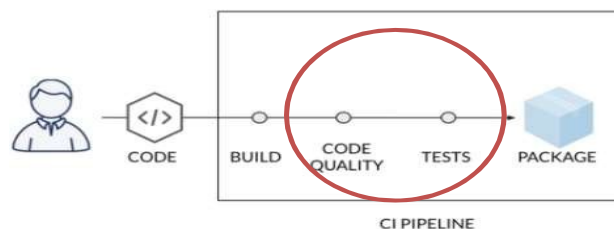
L'exécution du pipeline est lancée automatiquement à chaque commit.

Verrez la progression et la console de l'exécution du job dans **Build > Pipelines** de Gitlab.

L'exécution du job est-il passé ou échoué ?



## Stage de Test

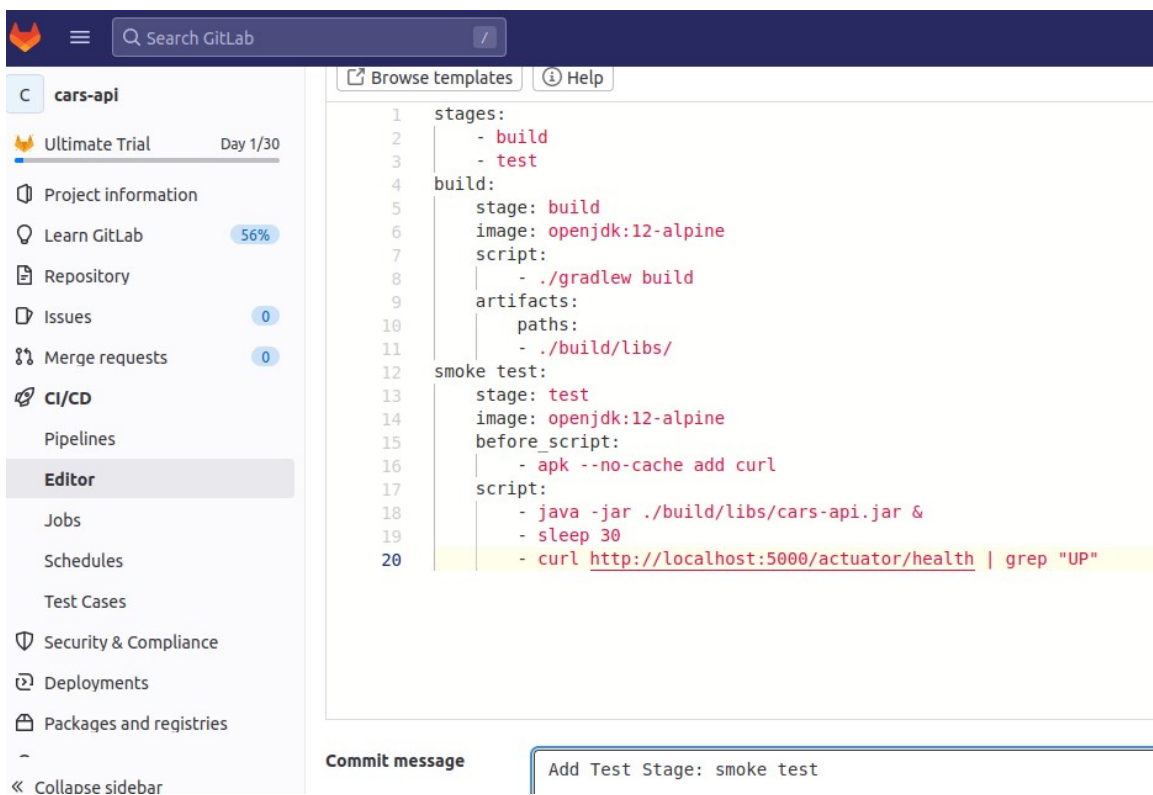


1. Dans cette étape (Stage), nous allons créer et lancer deux tests en parallèle :
  - **smoke test**: un test simple qui vérifie le bon fonctionnement de notre microservice en utilisant l'URL: <http://localhost:5000/actuator/health>
  - **unit test**: ce sont des tests chargés de tester des unités de code, généralement des classes. Pour les projets Java, **JUnit** est le framework le plus populaire pour l'écriture de tests unitaires.
- Ajoutez au fichier **.gitlab-ci.yml** le job **smoke test** du premier test.

```
stages
- build
- test
```

```
build:
  stage: build
  image: openjdk:12-alpine
  script:
    - ./gradlew build
  artifacts:
    paths:
      - ./build/libs/
smoke test:
  stage: test
  image: openjdk:12-alpine
  before_script:
    - apk --no-cache add curl
  script:
    - java -jar ./build/libs/cars-api.jar &
    - sleep 30
    - curl http://localhost:5000/actuator/health | grep "UP"
```

- Créer un nouveau commit avec le message suivant : « Add Test Stage: smoke test »

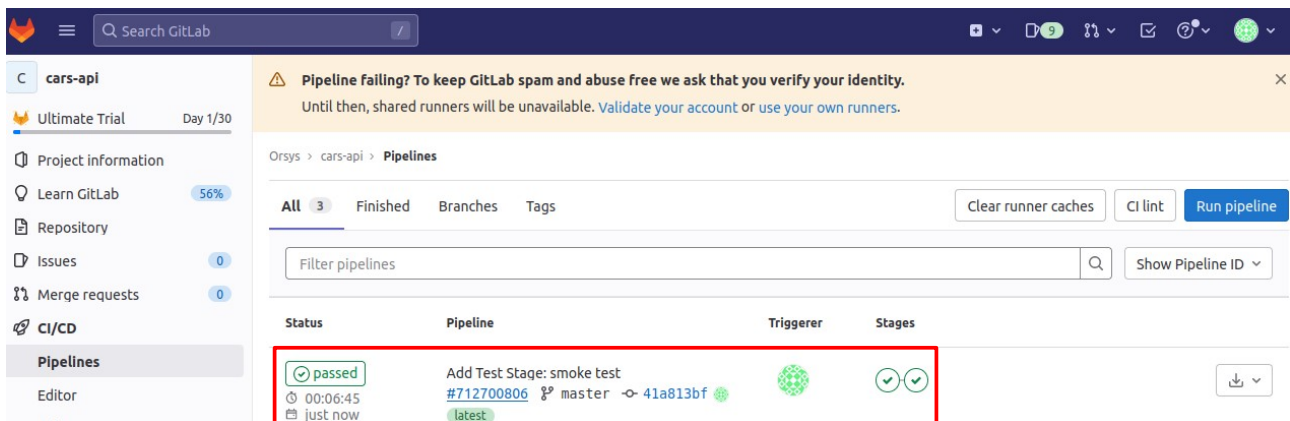


```

1  stages:
2    - build
3    - test
4  build:
5    stage: build
6    image: openjdk:12-alpine
7    script:
8      - ./gradlew build
9    artifacts:
10     paths:
11       - ./build/libs/
12  smoke test:
13    stage: test
14    image: openjdk:12-alpine
15    before_script:
16      - apk --no-cache add curl
17    script:
18      - java -jar ./build/libs/cars-api.jar &
19      - sleep 30
20      - curl http://localhost:5000/actuator/health | grep "UP"
  
```

Commit message: Add Test Stage: smoke test

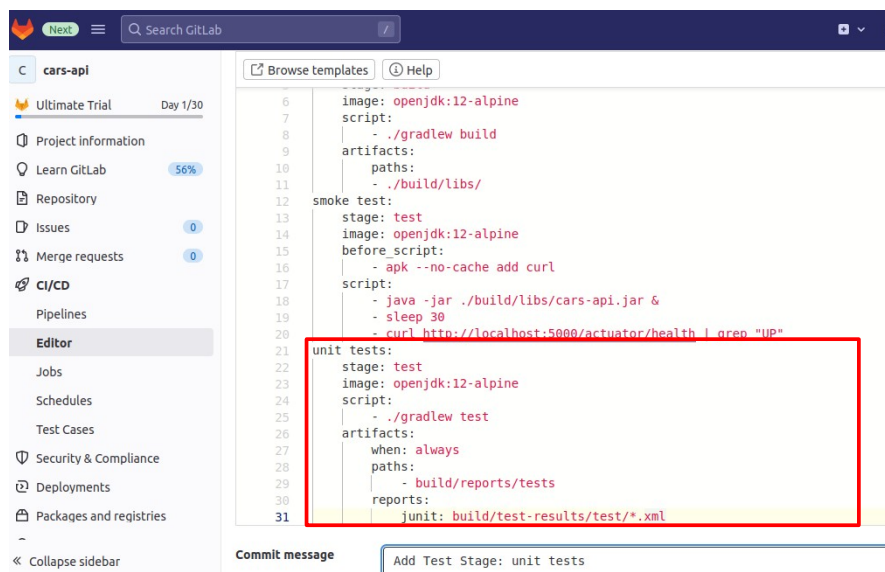
- Vérifiez l'exécution de cette étape dans **Build Pipelines** de GitLab.



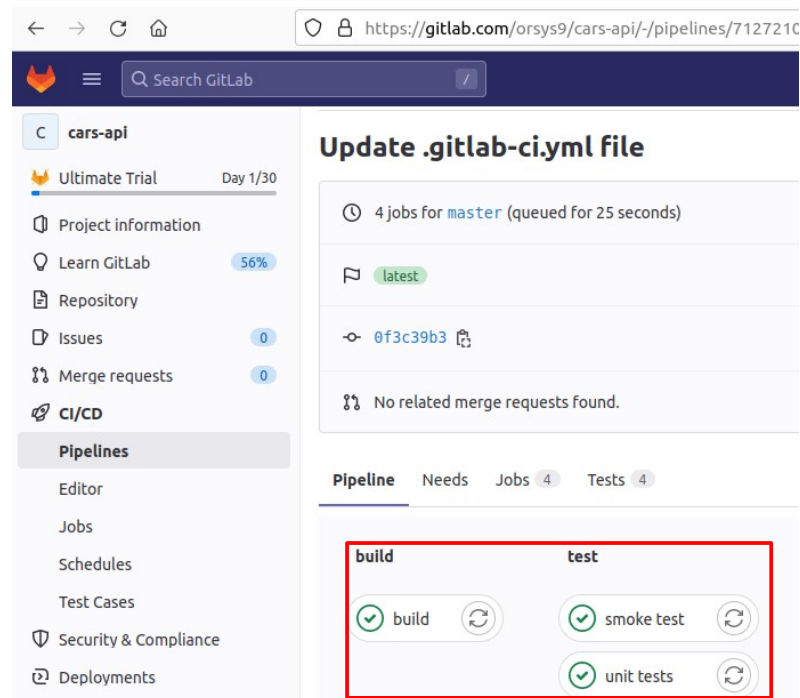
2. Ajoutez à la fin de votre fichier **.gitlab-ci.yml**, le script de deuxième test : **unit tests**

```
unit tests:
  stage: test
  image: openjdk:12-alpine
  script:
    - ./gradlew test
  artifacts:
    when: always
    paths:
      - build/reports/tests
  reports:
    junit: build/test-results/test/*.xml
```

- Créer un nouveau commit avec le message suivant : « Add Test Stage: unit tests »

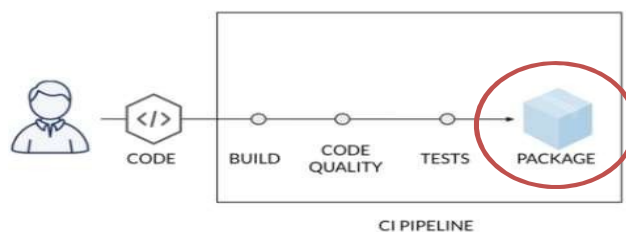


- Vérifiez l'exécution des deux tests en parallèle.



- Verrez le rapport de test unitaires dans votre *Artifact*. Quel est le résultat du rapport ?

## Stage de Package



1. Ce processus nous permettra, dans la livraison continue, de pouvoir déployer facilement le même code sur différents environnements.

Il sert aussi à figer le code compilé dans un package immutable.

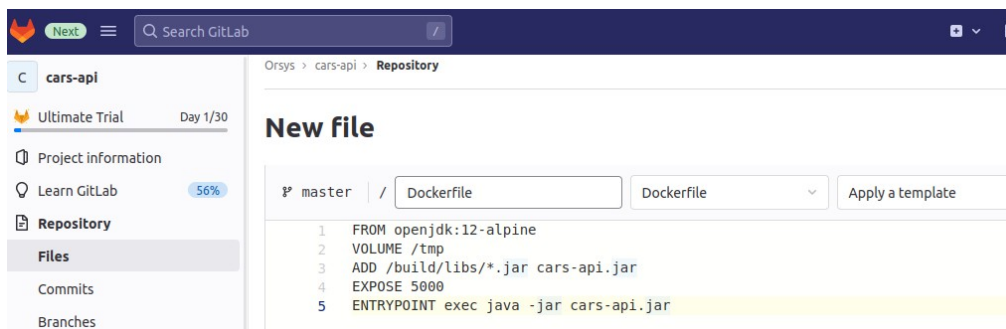


Un nouveau job supplémentaire compile le projet et l'encapsule dans un conteneur Docker.

Il est ensuite poussé sur la *registry* de GitLab.

- Tout d'abord, nous nous allons créer un nouveau fichier *Dockerfile* dans la racine de votre dépôt.
- Ajouter dans ce fichier le script permettant d'encapsuler dans un conteneur notre microservice à partir de l'image *openjdk:12-alpine* :

```
FROM openjdk:12-alpine
VOLUME /tmp
ADD /build/libs/*.jar cars-api.jar
EXPOSE 5000
ENTRYPOINT exec java -jar cars-api.jar
```



- Ajoutez à la fin de votre fichier *.gitlab-ci.yml*, le job **release** de troisième stage **package**.

*stages:*

- *build*
- *test*
- *package*

...

*release:*

*stage: package*

*image: docker:latest*

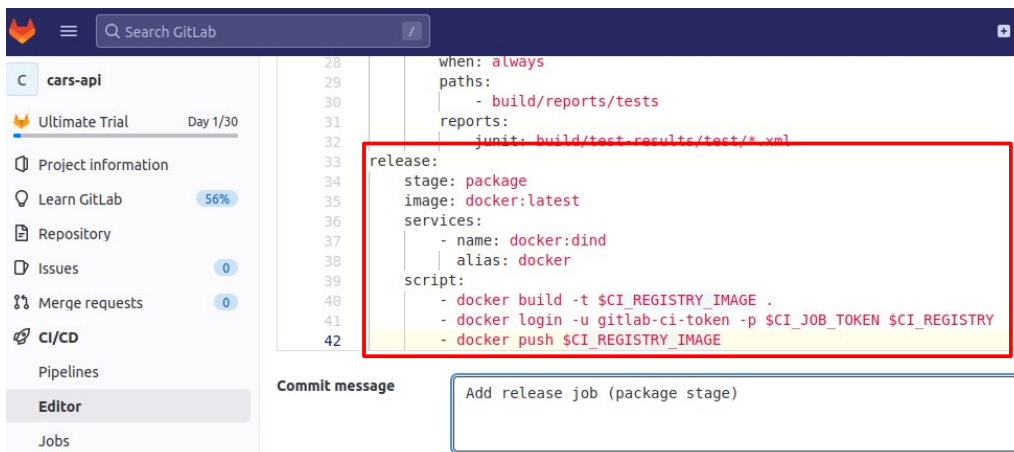
*services:*

- *name: docker:dind*
- alias: docker*

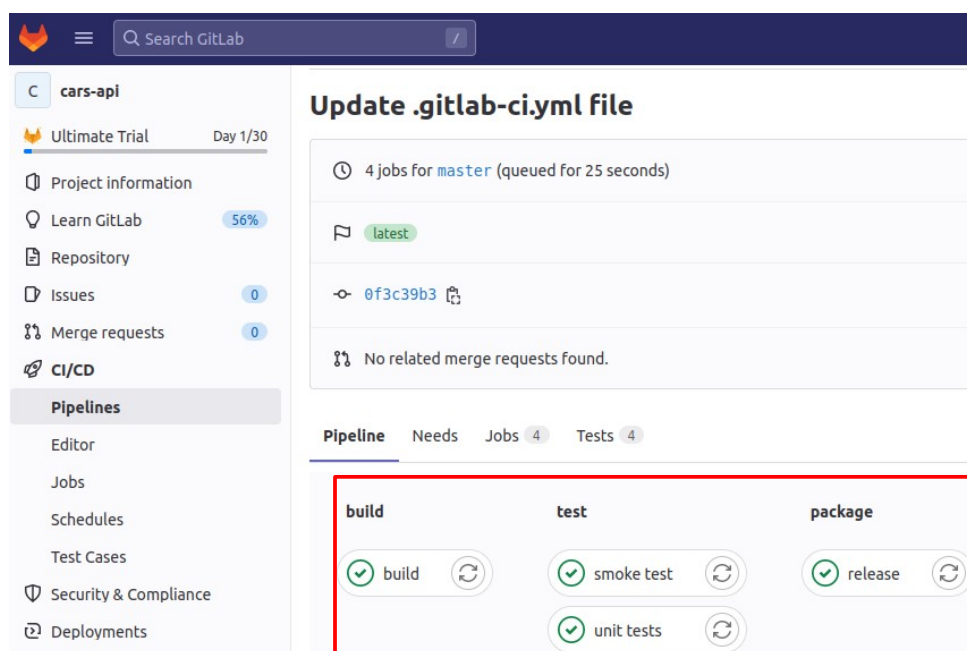
*script:*

- *docker build -t \$CI\_REGISTRY\_IMAGE .*
- *docker login -u gitlab-ci-token -p \$CI\_JOB\_TOKEN \$CI\_REGISTRY*
- *docker push \$CI\_REGISTRY\_IMAGE*

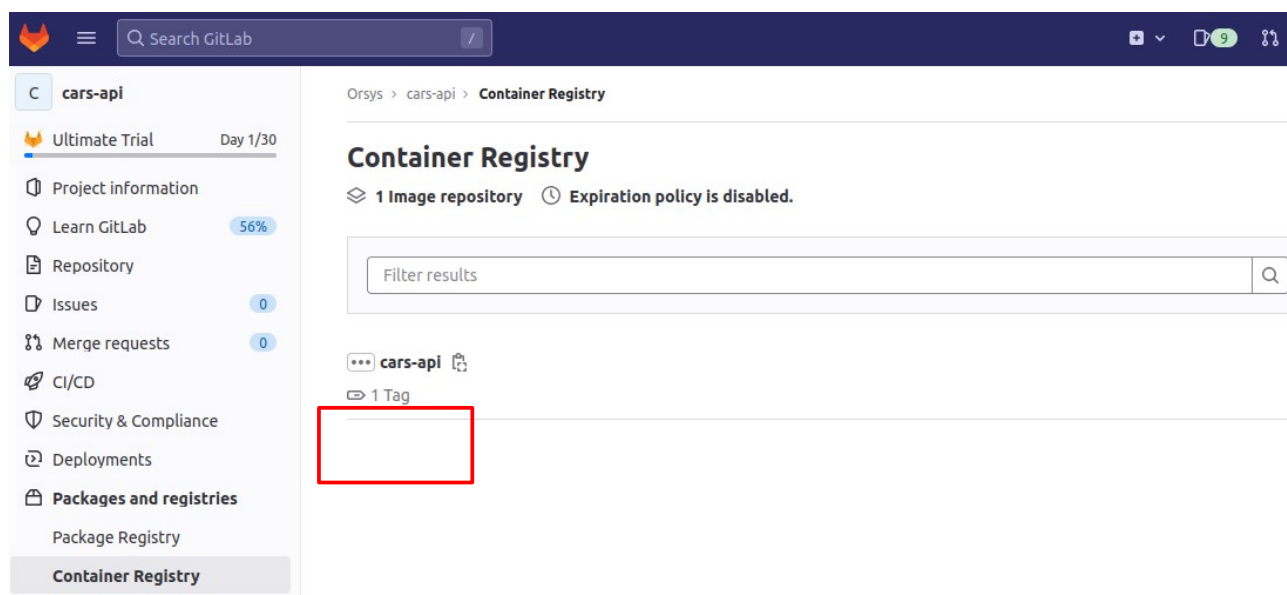
Nous connectons sur la *registry* interne de GitLab afin de pouvoir pousser (push) notre image Docker de façon sécurisée.



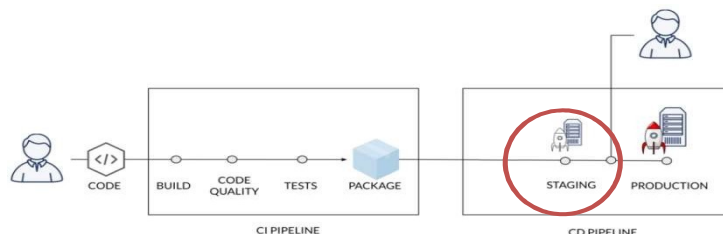
- Vérifiez l'exécution de votre pipeline CI.



- Enfin, vérifiez aussi la création de votre image Docker dans **Deploy > Container Registry** de Gitlab



## Déploiement sur l'environnement de Staging



1. Avant de passer à la production, nous allons créer une étape de pré-production afin de tester le bon fonctionnement de notre microservice dans un **environnement** de test.
  - Créez un nouveau fichier **docker-compose.yml** dans la racine de votre dépôt dont le contenu est le suivant (remplacez brahimhamdi par votre login gitlab) :

*version: '2'*

*services:*

*customers-service:*

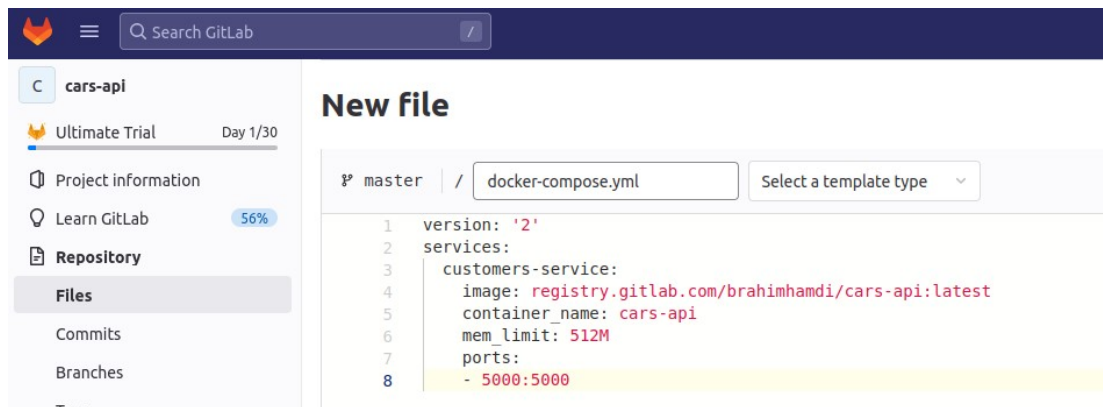
*image: registry.gitlab.com/brahimhamdi/cars-api:latest*

*container\_name: cars-api*

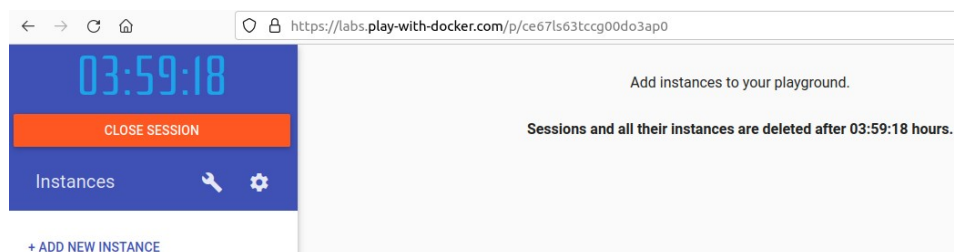
*mem\_limit: 512M*

*ports:*

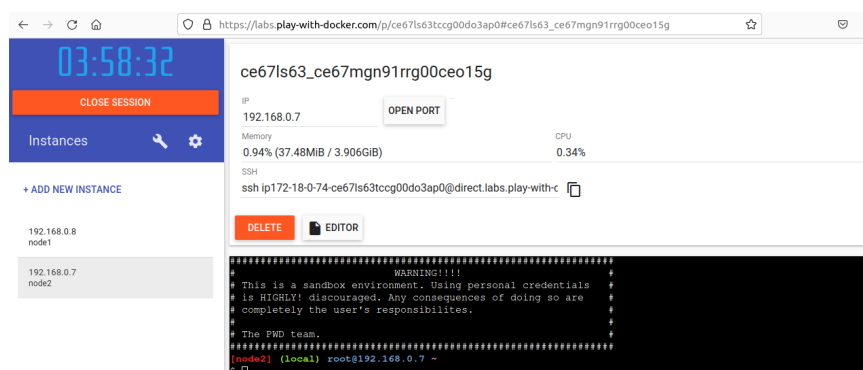
*- 5000:5000*



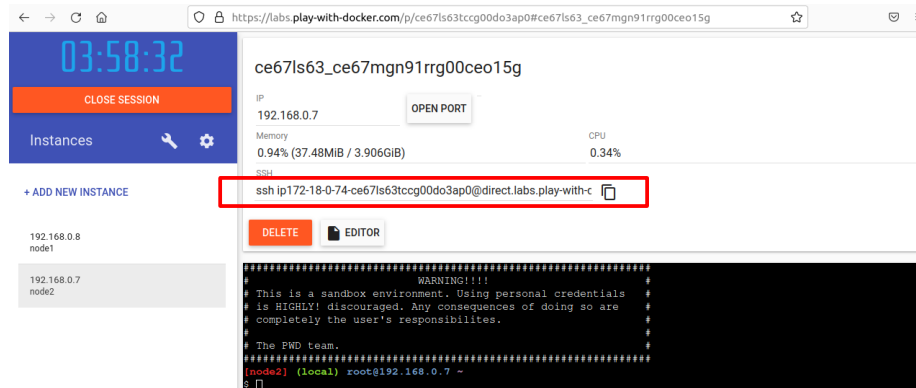
2. Pour créer une infrastructure Docker rapidement, rendez-vous sur le site <https://labs.play-with-docker.com/> et connectez-vous avec vos identifiants Docker Hub (créer un compte si vous n'avez pas).
  - Une fois connecté, une session de 4 heures est créée afin de vous permettre de déployer vos images.



- Sur la page d'accueil, cliquez deux fois sur l'icône **+ ADD NEW INSTAN** pour créer deux instances pour le déploiement en STAGING et en PRODUCTION.



- Une fois créés, récupérez l'URL de l'environnement. Il suffit de copier l'URL de la case SSH.



Nous utiliserons cette URL pour configurer l'environnement de déploiement dans le fichier `.gitlab-ci.yml`.

3. Avant d'ajouter le job de staging, vous devez créer un runner spécifique avec le tag **deploy\_stage**.

```
gitlab-runner register --non-interactive --url "https://gitlab.com/" --registration-token  
[remplacez-par-votre-token] --executor "docker" --docker-image docker:latest --tag-list deploy_stage
```

4. Ajoutez par la suite le job **staging** dans l'étape de **deploy staging**.

**stages:**

- build
- test
- package
- deploy staging

...

**staging:**

stage: deploy staging

tags :

- deploy\_stage

**variables:**

PLAYWD: ip172-18-0-74-ce67ls63tccg00do3ap0 //Mettre l'URL de votre env.

**script:**

- export DOCKER\_HOST=tcp://\$PLAYWD.direct.labs.play-with-docker.com:2375
- docker login -u gitlab-ci-token -p \$CI\_JOB\_TOKEN \$CI\_REGISTRY
- docker-compose down
- docker-compose up -d

**environment:**

name: staging

url: http://\$PLAYWD-5000.direct.labs.play-with-docker.com



```

45 staging:
46   stage: deploy staging
47   tags:
48     - deploy1
49   variables:
50     PLAYWD: ip172-18-0-132-ce7dkhf91rrg008c3jh0
51   script:
52     - apk add --no-cache --quiet py-pip
53     - pip install --quiet docker-compose~=1.23.0
54     - export DOCKER_HOST=tcp://$PLAYWD.direct.labs.play-with-docker.com:2375
55     - docker login -u gitlab-ci-token -p $CI_JOB_TOKEN $CI_REGISTRY
56     - docker-compose down
57     - docker-compose up -d
58   environment:
59     name: staging
60     url: http://$PLAYWD-5000.direct.labs.play-with-docker.com

```

La nouvelle variable PLAYWD contient l'URL copiée précédemment

- Vérifiez l'exécution de votre pipeline CI/CD.
5. Si tout s'est bien passé, vous devriez voir apparaître dans vos environnements **Deploy** > **Environments**, le nouvel environnement **Staging**.
- Cliquez sur **Operate** > **Environments**, puis le lien "**Open live environment**", pour voir l'interface de l'application déployée (ajoutez **/cars** à la fin de votre URL).

## Déploiement sur Production

1. Ajoutez par la suite dans votre manifeste le code source de l'étape de **production**.

```

stages:
  - build
  - test
  - package
  - deploy staging
  - deploy production
...
production:
  stage: deploy production
  tags :
  - deploy_stage
  image: docker:latest
  variables:

```

```
PLAYWD: ip172-18-0-103-c2gksclmrep00arjblg
script:
- apk add --no-cache --quiet py-pip
- pip install --quiet docker-compose~=1.23.0
- export DOCKER_HOST=tcp://$PLAYWD.direct.labs.play-with-docker.com:2375
- docker login -u gitlab-ci-token -p $CI_JOB_TOKEN $CI_REGISTRY
- docker-compose down
- docker-compose up -d
when: manual
allow_failure: false
environment:
  name: master
  url: http://$PLAYWD-5000.direct.labs.play-with-docker.com
```

- Vous devez utiliser l'URL de la deuxième instance déjà créée dans **https://labs.play-with-docker.com/**
- Gitlab offre la possibilité de déclencher manuellement des jobs avec la déclaration **when:manual**.
- S'il y a des étapes supplémentaires après ce job, elles seront toujours exécutées.
- Pour bloquer ces jobs, nous devons ajouter la déclaration **allow\_failure:false**

2. Sur votre pipeline de livraison continue, le déploiement manuel est symbolisé par l'icône à côté de l'étape deploy production.

- Lancer le déploiement manuel en cliquant sur l'icône
- Vous devriez voir apparaître dans vos environnements (*Operations > Environnements*), le nouvel environnement production.