Tests Unitaires et chaîne d'intégration, cas d'une application web MVC avec Spring Boot

1. Objectif de l'atelier

Mettre en place :

1. Une application Spring Boot MVC
2. Des tests unitaires avec JUnit
3. Un rapport de couverture de code via JaCoCo
4. Une **intégration continue (CI)** avec GitHub Actions
5. Une vérification de la sécurité de dépendances (optionnelle, via GitHub Advanced Security ou OWASP dependency-check)

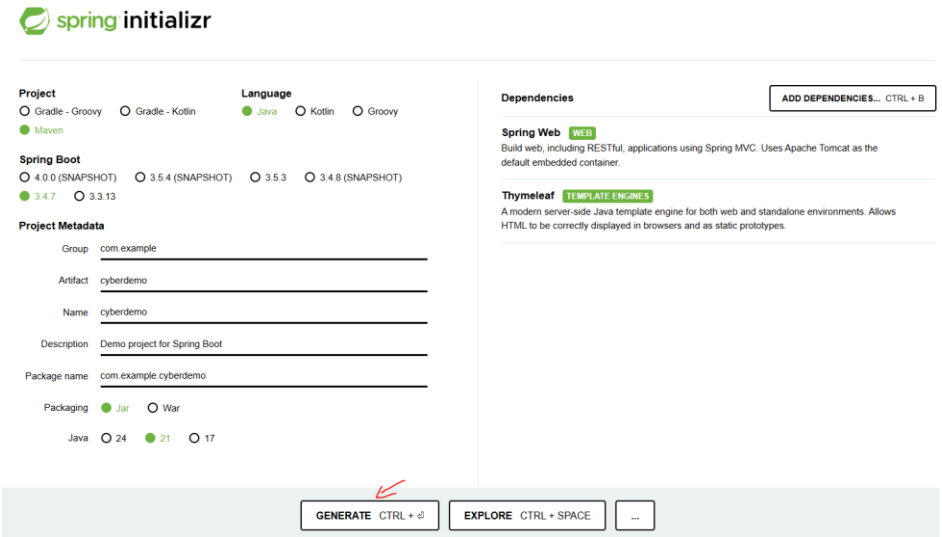
2. Pré requis

- Connaissances basiques de Git
- Avoir un compte GitHub
- VS Code avec extensions :
  - Java Extension Pack
  - Spring Boot Extension Pack
  - live preview
- Java 17 (ou 21) + Maven installés sur Windows 10

3. Étapes de l'atelier

3.1 Création du projet Spring Boot

a. Aller sur https://start.spring.io
b. Configuration :
   - ✓ **Project**: Maven
   - ✓ **Language**: Java
   - ✓ **Spring Boot**: 3.4.7
   - ✓ **Group**: com.example
   - ✓ **Artifact**: cyberdemo
   - ✓ **Dependencies**: Spring Web, Thymeleaf
c. Télécharger et dézipper le projet.
d. Ouvrir le dossier dans **VS Code**.



e. Ouvrir le projet dans VsCode

- Dezipper l'archive et ouvrir le répertoire contenant le projet dans VsCode
- Ouvrir le fichier src/main/java/com/example/cyberdemo/CyberDemoApplication.java
- Cliquer sur « Run Java » petite flèche en haut à droite.

Le projet s'exécute, le serveur d'application Tomcat est automatiquement lancé.
La page d'accueil est disponible dans votre navigateur sur le port 8080 → http://127.0.0.1:8080/

Comme il n'y a pas de page à cette adresse, vous obtenez une page d'erreur, mais le projet est initialisé !

3.2 Ajouter les fichiers minimums pour l'application MVC

a. Controller

src/main/java/com/example/cyberdemo/controller/CyberController.java :

```
package com.example.cyberdemo.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;

@Controller
public class CyberController {
    @GetMapping("/")
    public String index() {
        return "index";
    }
}
```

b. Vue Thymeleaf

src/main/resources/templates/index.html :

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>CyberEdu</title>
    <link rel="stylesheet" type="text/css" th:href="@{/css/styles.css}" />
</head>
<body>
    <h1>Hello Cyber World</h1>
</body>
</html>
```

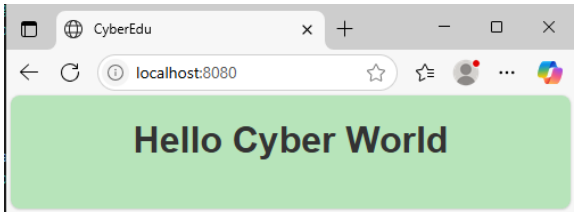
c. Feuille de style

src/main/resources/static/css/styles.css :

```
body {
    font-family: 'Arial', sans-serif;
    margin: 20px;
    background-color: #b7e4ba;
}

h1 {
    color: #333;
    text-align: center;
}
```

Exécuter l'application

http://127.0.0.1:8080/ 

3.3 Tests unitaires

Créer, ou modifier, le fichier CyberControllerTest.java, dans \src\test\java\com\example\cyberdemo :

```
package com.example.cyberdemo;

import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

@SpringBootTest
@AutoConfigureMockMvc
public class CyberControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    void testIndexPage() throws Exception {
        mockMvc.perform(get("/"))
                .andExpect(status().isOk());
    }
}
```

Exécuter les TU du projet en local

Depuis l'IDE

- Sélectionner le fichier src/test/java/com/example/cyberdemo/CyberdemoApplicationTests.java
- Cliquer sur la flèche à coté du test

Depuis le terminal : mvn test

3.4 Etape de taux de couverture du code avec JaCoCo

Ajouter dans pom.xml : cyberdemo\ pom.xml

```
xml
CopierModifier
<build>
  <plugins>
    <!-- Autres plugins -->

    <plugin>
      <groupId>org.jacoco</groupId>
      <artifactId>jacoco-maven-plugin</artifactId>
      <version>0.8.11</version>
      <executions>
        <execution>
          <goals>
            <goal>prepare-agent</goal>
          </goals>
        </execution>
        <execution>
          <id>report</id>
          <phase>verify</phase>
          <goals>
            <goal>report</goal>
          </goals>
        </execution>
      </executions>
    </plugin>

  </plugins>
</build>
```

a. Lancer en local :

Vous devez vous **placer dans le dossier qui contient le fichier pom.xml** de votre projet avant d'exécuter la commande, *« exp : C:\Users\ciel\Downloads\Cyber\Cyber ».*

```
mvn test
mvn verify
```

Le rapport est dans : target/site/jacoco/index.html

b. *Visualiser le coverage du projet en local*

Depuis l'IDE

fichier src/test/java/com/example/cyberdemo/CyberdemoApplicationTests.java

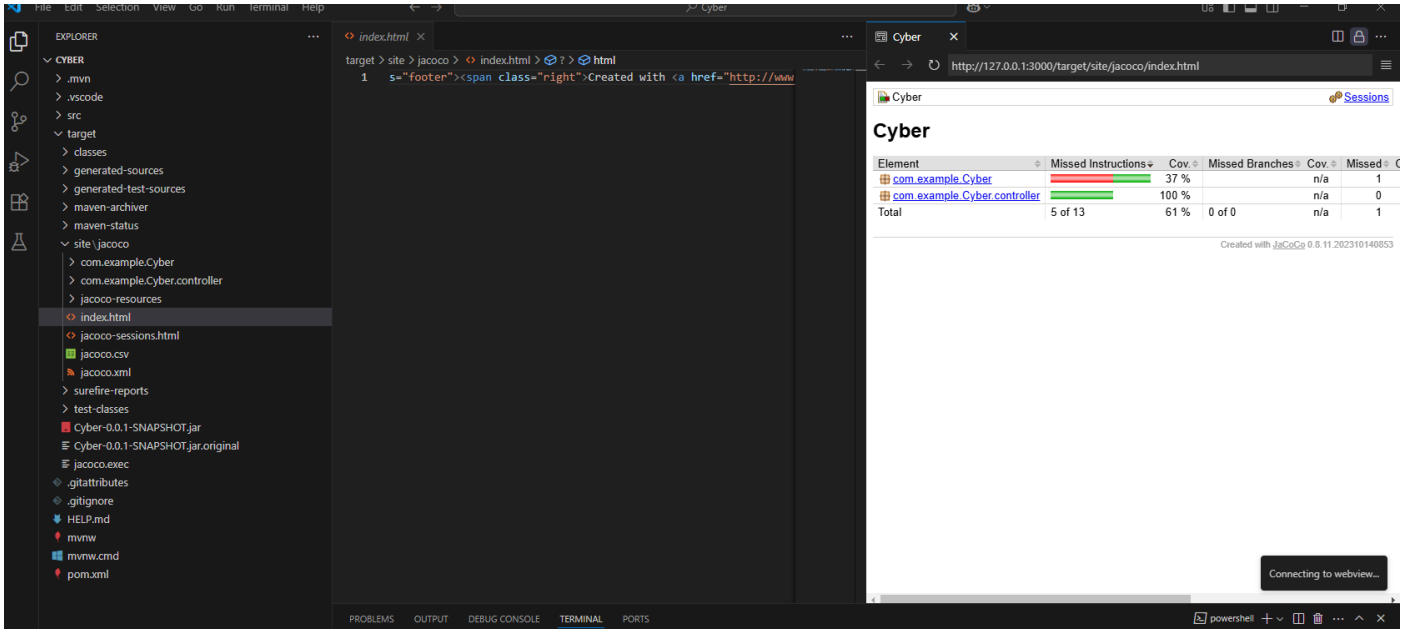
- Cliquer droit sur la flèche à coté du test → run with coverage

Dans la console

mvn jacoco:prepare-agent test install jacoco:report

Le rapport est généré dans target/sites/jacoco

- Cliquer sur index.html → Show Preview (en haut à droite)



3.5 Mise en place du pipeline d'intégration

Étape A. Initialiser git

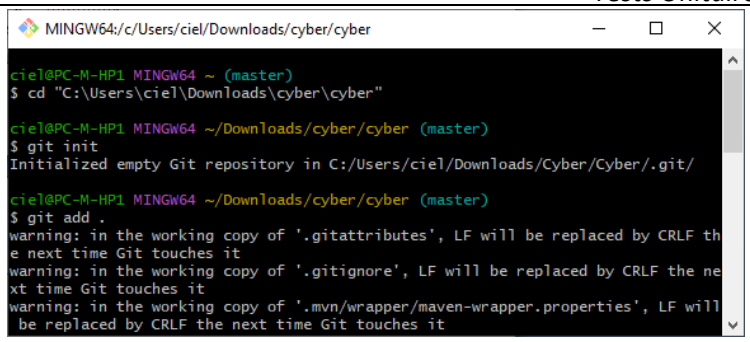
Alors tu ouvres un terminal (PowerShell ou Git Bash) **dans ce dossier** et tu exécutes :

```
cd "C:\Users\ciel\Downloads\cyberdemo\cyberdemo"
git init
git add .
git commit -m "Initial commit with tests and CI"
```

Pourquoi à cet endroit *« C:\Users\ciel\Downloads\cyberdemo\cyberdemo »***?**

Parce que :

- C'est là que se trouve le pom.xml (le fichier racine du projet Maven).
- C'est là que le répertoire .github/workflows/ci.yml doit exister.
- C'est ce dossier que tu vas *lier* au dépôt distant GitHub avec git remote.



Étape B. Créer un dépôt GitHub vide

Rendez-vous sur https://github.com/new, nommez-le cyberdemo, **ne cochez aucune case** (README, gitignore, licence...).

Étape C. Lier le dépôt local au dépôt distant

Dans Git Bash : **git remote add origin https://github.com/<utilisateur>/cyberdemo.git**

Exemple : git remote add origin https://github.com/DJILI-K/cyberdemo.git

Étape D. Pousser le projet

```
git push -u origin main
```

Que se passe-t-il après le push ?

- GitHub détecte .github/workflows/ci.yml
- Il exécute automatiquement mvn verify via GitHub Actions
- Le badge dans le README.md devient **actif**
- Le rapport JaCoCo peut être intégré plus tard via un outil externe (ex: Codecov)

si la branche locale main **n'existe pas encore**. Et il y a que la branche master (comme indiqué ici dans le prompt : cyberdemo (master)), donc Git ne trouve pas de branche main à pousser.

Deux solutions dans ce cas :

Option 1 : Pousser master à la place de main

Tu peux simplement pousser la branche existante master vers GitHub :

```
git push -u origin master
```

GitHub va créer la branche distante master dans ton dépôt.

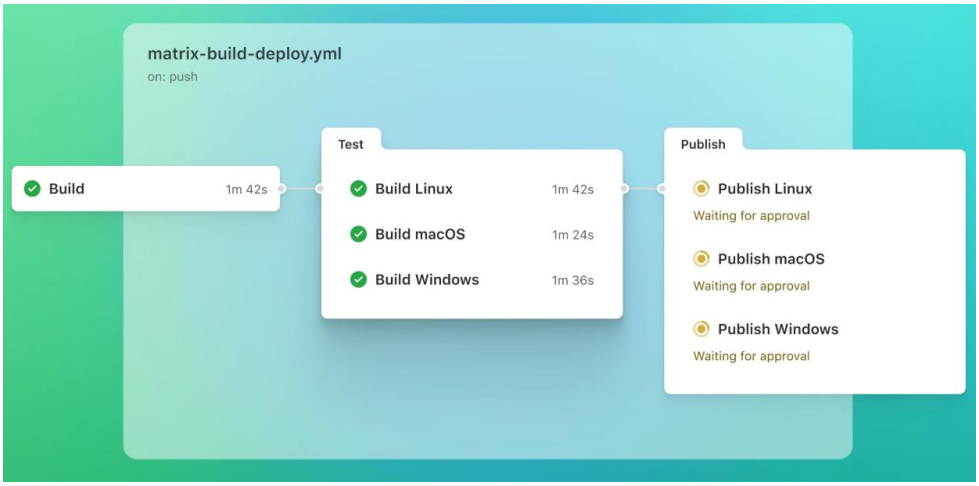
Option 2 : Renommer ta branche master en main (option moderne)

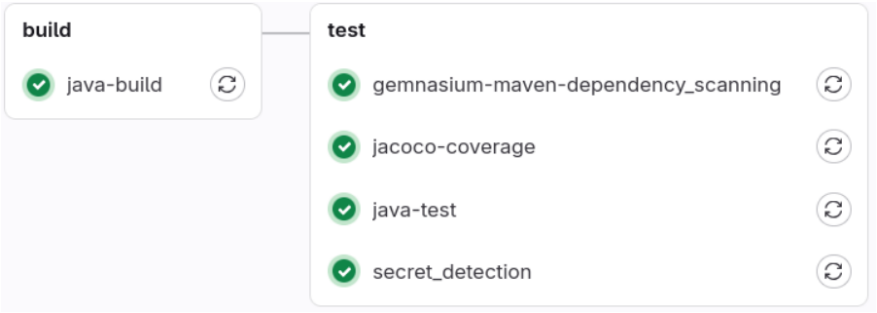
Si tu veux être cohérent avec les conventions modernes (GitHub utilise main par défaut), tu peux renommer ta branche :

```
git branch -m main          # Renomme master en main
git push -u origin main     # Pousse la branche main
```

GitHub créera la branche main comme branche par défaut du dépôt.

Étape E. Intégration continue avec GitHub Actions





a. Créer .github/workflows/ci.yml :

Aller sur Github : Actions → Choose a workflow → set up a workflow yourself

Fichier ci.yml :

```
name: Java CI with Maven

on:
  push:
  pull_request:

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout source
        uses: actions/checkout@v4

      - name: Setup JDK
        uses: actions/setup-java@v4
        with:
          distribution: 'temurin'
          java-version: '21'
```

```
    - name: Build with Maven
      run: mvn clean compile

    - name: Run Tests and Coverage
      run: mvn verify

    - name: Upload JaCoCo Report
      uses: actions/upload-artifact@v4
      with:
        name: jacoco-report
        path: target/site/jacoco/

    - name: Run Gitleaks (secret detection)
      uses: gitleaks/gitleaks-action@v2
      with:
        fail: true


    - name: Upload Dependency Report
      uses: actions/upload-artifact@v4
      with:
        name: dependency-report
        path: reports/
```

Exemple de Fichier README.md avec un badge :

```
# cyberdemo

![Java CI](https://github.com/DJILI-K/Cyber/actions/workflows/ci.yml/badge.svg)

> Un projet d'exemple Spring Boot avec tests unitaires, couverture de code avec JaCoCo et intégration continue via GitHub Actions.

---

##  Technologies utilisées

- Java 21
- Spring Boot
- Maven
- JUnit 5
- JaCoCo
- GitHub Actions

---

##  Structure du projet
cyber/
├── src/
│   ├── main/java/... # Code source de l'application
│   └── test/java/... # Tests unitaires
├── .github/workflows/ci.yml # Fichier d'intégration continue
├── pom.xml # Configuration Maven
└── README.md # Ce fichier

---
```

```
## Lancer les tests

```bash
mvn test

mvn verify

target/site/jacoco/index.html
```

3.6 Etapes DevSecOps

3.6.1 Vérification des dépendances

Option 1 : GitHub Advanced Security (si activé)

- Allez dans l'onglet **Security** du repo GitHub
- Activez **Dependabot alerts**

Option 2 : OWASP Dependency Check local ou CI

a. Ajouter un plugin Maven dans le fichier pom.xml :

```
<plugin>
  <groupId>org.owasp</groupId>
  <artifactId>dependency-check-maven</artifactId>
  <version>8.4.0</version>
  <executions>
    <execution>
      <goals>
        <goal>check</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Puis exécuter :

```
mvn org.owasp:dependency-check-maven:check
```

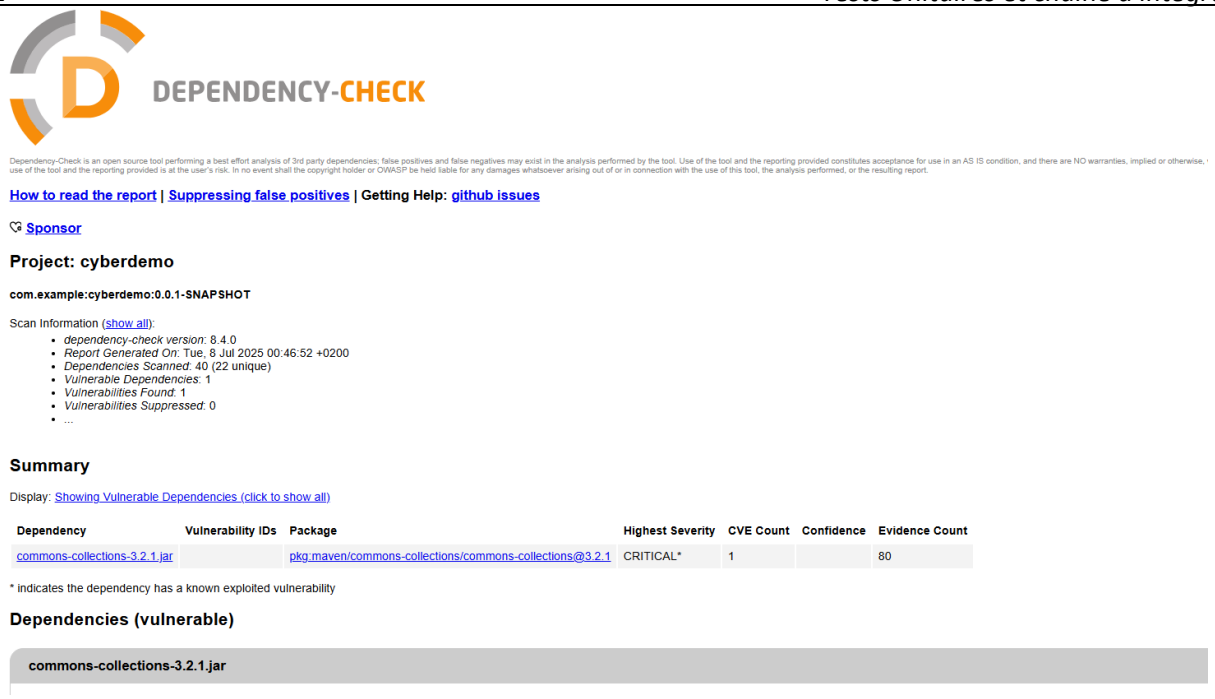
Lancer l'analyse localement

```
mvn verify
```

b. Ouvrir le rapport HTML généré **:**

- Fichier : target/dependency-check-report.html
- Tu peux l'ouvrir avec ton navigateur : double-clique sur ce fichier

  Ou http://127.0.0.1:3000/target/dependency-check-report.html
```



DEPENDENCY-CHECK

Dependency-Check is an open source tool performing a best effort analysis of 3rd party dependencies; false positives and false negatives may exist in the analysis performed by the tool. Use of the tool and the reporting provided constitutes acceptance for use in an AS IS condition, and there are NO warranties, implied or otherwise, use of the tool and the reporting provided is at the user's risk. In no event shall the copyright holder or OWASP be held liable for any damages whatsoever arising out of or in connection with the use of this tool, the analysis performed, or the resulting report.

How to read the report | Suppressing false positives | Getting Help: github issues

Sponsor

Project: cyberdemo

com.example:cyberdemo:0.0.1-SNAPSHOT

Scan Information (show all):

- *dependency-check version*: 8.4.0
- *Report Generated On*: Tue, 8 Jul 2025 00:46:52 +0200
- *Dependencies Scanned*: 40 (22 unique)
- *Vulnerable Dependencies*: 1
- *Vulnerabilities Found*: 1
- *Vulnerabilities Suppressed*: 0
- ...

Summary

Display: Showing Vulnerable Dependencies (click to show all)

Dependency	Vulnerability IDs	Package	Highest Severity	CVE Count	Confidence	Evidence Count
commons-collections-3.2.1.jar		pkg:maven/commons-collections/commons-collections@3.2.1	CRITICAL*	1		80

* indicates the dependency has a known exploited vulnerability

Dependencies (vulnerable)

commons-collections-3.2.1.jar

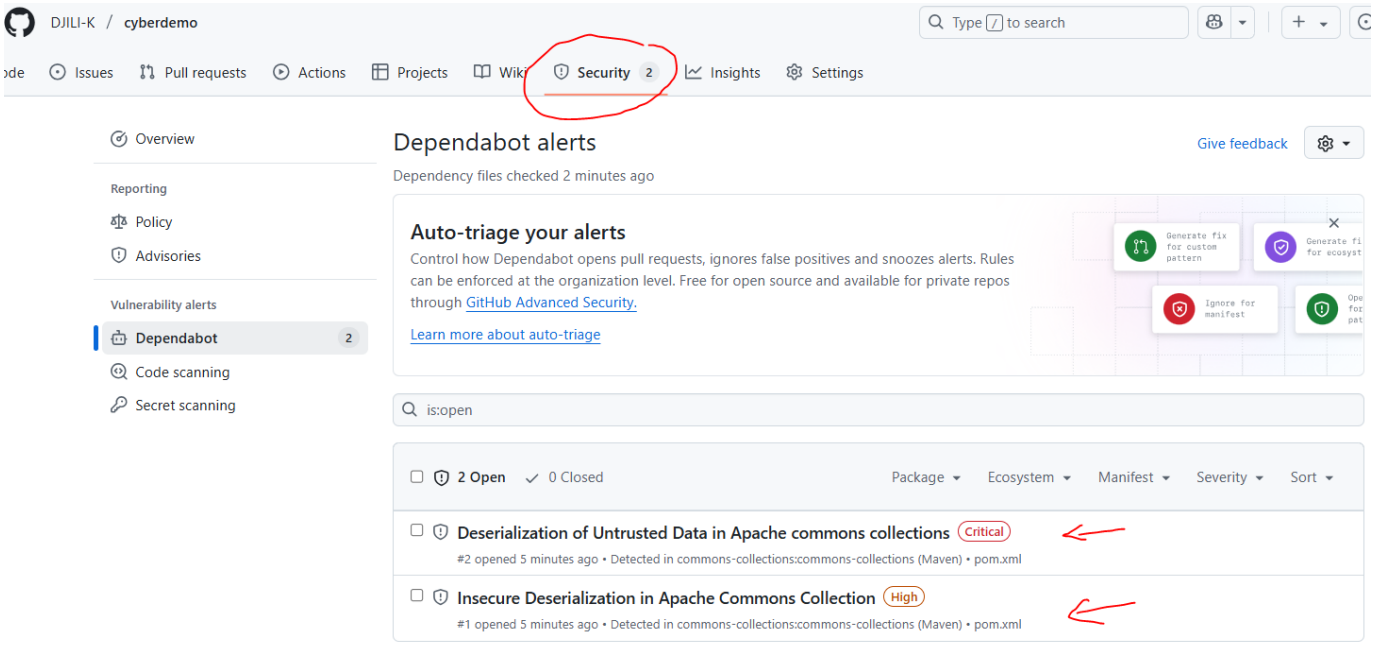
3.6.2 Modification du projet pour créer des problèmes

a. Ajout d'une dépendance avec une vulnérabilité connue

Ajouter cette dépendance dans le fichier pom.xml

```
<dependency>
        <groupId>commons-collections</groupId>
        <artifactId>commons-collections</artifactId>
        <version>3.2.1</version>
</dependency>
```

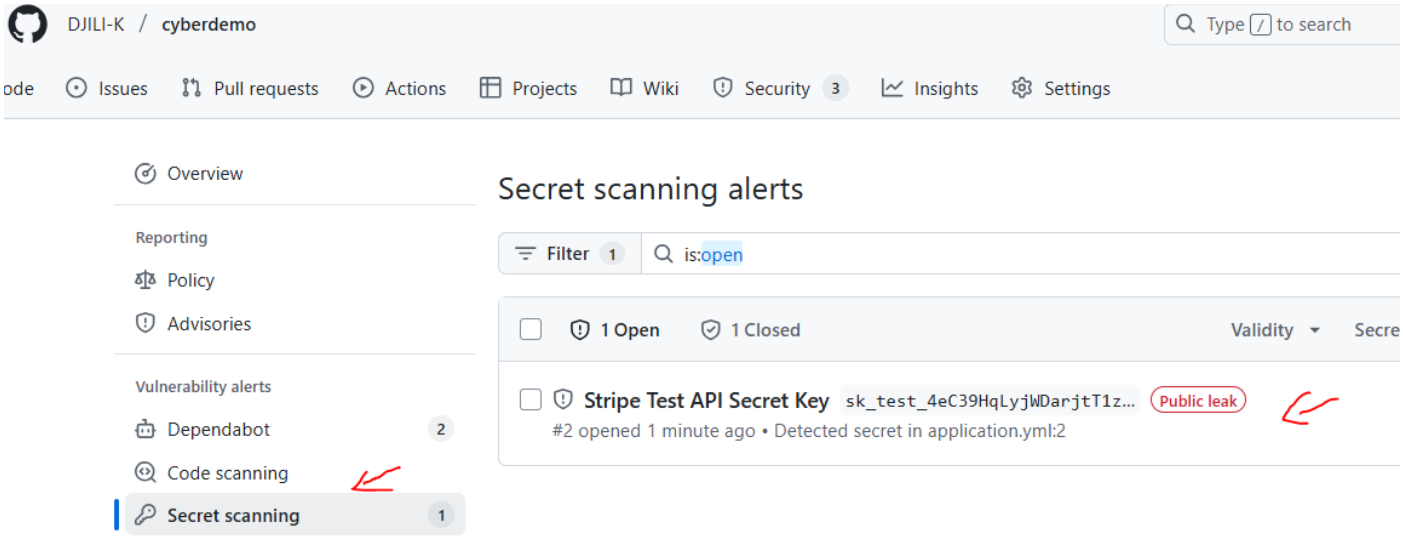
Remarque : dès l'ajout de la dépendance, Red Hat Dependency Analysis nous avertit d'un problème.



**Ajout d'une clé API **

b. Créer le fichier « application.yml » à la racine du projet et ajouter dans ce fichier

```
api:
  key: sk_test_4eC39HqLyjWDarjtT1zdp7dc
```



Bonne pratique concernant les clés 1\. Utiliser des variables d'environnement L'approche la plus courante et sécurisée est de stocker les clés d'API et autres secrets dans des variables d'environnement. Spring Boot peut les lire facilement en utilisant des placeholders dans le fichier \`application.yml\`Exemple avec application.yml

```
api:
  key: ${API_KEY:default_value}
```

Vous pouvez configurer cette variable dans votre environnement en utilisant des fichiers .env

2. Dans le cloud, utilisez des services de gestions de secret comme **AWS Secrets Manager (pour AWS)**