Selenium

Framework d'Automatisation des Tests Web

Testing • Automatisation • WebDriver

Plan du Cours

- Introduction
- Architecture WebDriver
- Prise en main & Premières interactions
- Sélection d'éléments (XPath vs CSS)
- Actions simples
- Attentes (Implicites & Explicites)
- Actions complexes
- POM (Page Object Model)

Introduction

- Framework **open-source** pour automatiser les tests web
- Langages supportés: **Java, Python, C#, JavaScript, Ruby**
- Navigateurs: **Chrome, Firefox, Safari, Edge, Opera**
- Simule les interactions d'un **utilisateur réel**
- Standard **W3C WebDriver Protocol** pour la communication
- **Gratuit** et largement adopté dans l'industrie

Architecture WebDriver

Les 4 composants principaux

- API disponible en : **Java, Python, C#, JavaScript, Ruby**
- **W3C WebDriver Protocol :** Protocole de communication standard
- **WebDriver Server :** Service qui reçoit et exécute les commandes
- **Browser Driver :** ChromeDriver, GeckoDriver, EdgeDriver, etc.

Inbtégrer Selenium

Dépendances dans le pom.xml

```
<dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-java</artifactId>
    <version>4.38.0</version>
</dependency>

<dependency>
    <groupId>io.github.bonigarcia</groupId>
    <artifactId>webdrivermanager</artifactId>
    <version>5.7.0</version>
</dependency>
```

Première Classe de Test

```
import io.github.bonigarcia.wdm.WebDriverManager;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

public class BaseTest {

    private WebDriver driver;

    @BeforeEach
    public void setUp() {
        WebDriverManager.chromedriver().setup();
        driver = new ChromeDriver();
        driver.manage().window().maximize();
    }

    @AfterEach
    public void tearDown() {
        if (driver != null) {
            driver.quit();
        }
    }

}
```

- @BeforeEach : attribut Junit pour une méthode appelée avant l'exécution des tests

- @AfterEAch = attribute Junit pour une méthode appelée après l'execution des tests

DOM : accéder à une page

Méthodes de base

```
// Charger une URL
driver.get("https://www.example.com");

// Naviguer (avec historique)
driver.navigate().to("https://www.example.com");
driver.navigate().back();
driver.navigate().forward();
driver.navigate().refresh();

// Informations du navigateur
String title = driver.getTitle();
String currentUrl = driver.getCurrentUrl();
```

DOM : locators “By”

```
// Trouver un seul élément
WebElement element = driver.findElement(By.id("myId"));

// Trouver plusieurs éléments
List<WebElement> elements =
driver.findElements(By.className("myClass"));

// Les 8 locators disponibles
By.id("elementId");
By.name("elementName");
By.className("myClass");
By.tagName("input");
By.linkText("Click Here");
By.partialLinkText("Click");
By.cssSelector("div.classname");
By.xpath("//div[@id='myId']");
```

Sélection d'Éléments - CSS Selectors

```
button              // tag
.submitBtn          // class
#myId               // id
button.primary      // tag + class

// Attributs
input[type="email"]
input[name="username"]
button[disabled]

// Pseudo-classes
input:first-child
button:last-of-type
li:nth-child(2)
li:nth-of-type(odd)
```

- input:first-child : le premier “input” enfant de leur parent
- button:last-of-type : le dernier élément “button” parmis ses frères directs
- li:nth-child(2) : sélectionne les éléments “li” qui sont le deuxième enfant de leur parent
- li:nth-of-type(odd) : le premier “input” enfant de leur parent

Sélection d'Éléments – Xpath basiques

```
// Chemin absolu : 1er bouton de la page
/html/body/div/button

// Chemin relatif : bouton dont l'id est vaut "submitBtn"
//button[@id='submitBtn']

// Attributs
//input[@type='email'] // récupère le 1er input de type email
//input[@name='username'] // récupère l'input de nom "username"
//div[@class='container'] // récupère la 1ère div ayant la classe "container"

// Text nodes
//button[text()='Click Me'] // récupère le 1er bouton avec le texte "Click Me"
//button[contains(text(), 'Click')] // récupère le 1er bouton dont le texte contient "Click"


// Index
//tr[1] // récupère le 1er "tr"
//tr[position()>2] // récupère les les lignes après la 2ème

// Combinaisons
//form//input[@type='text' and @required] // le 1er input text + required dans un form
```

Sélection d'Éléments - XPath Avancé

```
// Parent navigation
//button/ancestor::form // Le formulaire contenant le bouton
//input/parent::div // La div parente de l'input

// Siblings
//button/following-sibling::span // Tous les "span" qui ont le même parent que le button
//input/preceding-sibling::label // Tous les "label" qui sont des frères ainés à l'input

// Descendants
//form/descendant::input // Tous les "input" qui sont descendants de n'importe quel form
//div//p // Tous les "p" qui sont enfants, direct ou non, dans une div

// Wildcards
//*[@class='error'] // N'importe quel élément qui possède l'attribut class de valeur "error"
//*/button // Tous les "button"

// Functions
//button[starts-with(@id, 'btn_')] // Tous les "button" dont l'id commence par "btn_"
//input[contains(@placeholder, 'Enter')] // Tous les "input" dont le placeholder contient "Enter"
//span[normalize-space()='Username'] // Tous les "span" dont le texte, après avoir supprimé les
espaces, contient "Username"
```

XPath vs CSS - Comparaison

Critère	XPath	CSS
Navigation ascendante	Oui	Non
Performance	Lente	Rapide
Complexité	Plus complexe	Simple
Contenu spécifique	Oui	Non

Recommandation : Utiliser CSS quand possible (performance), XPath pour cas spéciaux (navigation ascendante, contenu spécifique)

DOM : locators “actions simples”

```
WebElement textField = driver.findElement(By.id("username"));
WebElement button = driver.findElement(By.id("sendForm"));

// Texte et input
textField.sendKeys("mon_utilisateur");
textField.clear();

// Clics et soumission
button.click();
textField.submit();

// Récupérer des informations
String text = textField.getText();
String attribute = textField.getAttribute("placeholder");
boolean isDisplayed = textField.isDisplayed();
boolean isEnabled = textField.isEnabled();
```

« sendKeys » simule l’action dans le form

Attentes - Waits Implicites

```
@BeforeEach
public void setUp() {
    WebDriverManager.chromedriver().setup();
    driver = new ChromeDriver();
    driver.manage().window().maximize();
    driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(10));
}
```

Permet d'imposer une temporisation au « driver », permettant ainsi de laisser le temps aux éléments d'apparaître sur la page

Attentes - Waits Explicites (Recommandé)

Instancie le « WebDriverWait » :

```
WebDriverWait  wait = new WebDriverWait(driver, Duration.ofSeconds(10));
```

Attends que l'élément soit visible :

```
wait.until(ExpectedConditions.visibilityOfElementLocated(By.name("username")));
```

Attends que l'élément soit cliquable :

```
wait.until(ExpectedConditions.elementToBeClickable(By.cssSelector("button[type='submit']")));
```

```
protected WebElement waitForVisible(By locator) {
   return wait.until(ExpectedConditions.visibilityOfElementLocated(locator));
}
```

Attentes - Conditions Courantes

```
// Elements
wait.until(ExpectedConditions.elementToBeClickable(By.name("username")));
wait.until(ExpectedConditions.visibilityOfElementLocated(By.name("username")));
wait.until(ExpectedConditions.invisibilityOfElementLocated(By.name("username")));
wait.until(ExpectedConditions.presenceOfElementLocated(By.name("username")));

// Texte
wait.until(ExpectedConditions.textToBePresentInElement(element, "Text"));

// URL & Titre
wait.until(ExpectedConditions.urlContains("/page"));
wait.until(ExpectedConditions.titleContains("Title"));
wait.until(ExpectedConditions.titleIs("Exact title"));

// Attributes
wait.until(ExpectedConditions.attributeContains(element, "class", "active"));
```

Actions Complexes - Remplir des Formulaires

```
WebElement emailField = driver.findElement(By.id("email"));
WebElement passwordField = driver.findElement(By.id("password"));
WebElement rememberMe = driver.findElement(By.id("rememberMe"));

// Fill up the form
emailField.sendKeys("user@example.com");
passwordField.sendKeys("verygoodpassword123");

// Check a checkbox / radio
rememberMe.click();

// Type a specific key
passwordField.sendKeys(Keys.ENTER);
```

Actions Complexes - Menus Déroulants (Select)

```
WebElement dropdown = driver.findElement(By.id("country"));
Select select = new Select(dropdown);

// Select by visible text
select.selectByVisibleText("France");

// Select by value
select.selectByValue("FR");

// Select by index
select.selectByIndex(0);

// Select multiple (if allowed)
select.selectByVisibleText("Option 1");
select.selectByVisibleText("Option 2");

// Unselect
select.deselectByVisibleText("Option 1");
select.deselectAll();

// Get the selected options
List<WebElement> selected = select.getAllSelectedOptions();
```

Actions Complexes - Clics Avancés & Hover

```
Actions actions = new Actions(driver);
WebElement element = driver.findElement(By.id("myElement"));
WebElement menu = driver.findElement(By.id("menu"));

// Right click
actions.contextClick(element).perform();

// Double click
actions.doubleClick(element).perform();

// Hover
actions.moveToElement(menu).perform();

// Drag and Drop
WebElement source = driver.findElement(By.id("draggable"));
WebElement target = driver.findElement(By.id("droppable"));
actions.dragAndDrop(source, target).perform();
```

Actions Complexes - Fenêtres & Pop-ups

```
// Handle windows
String mainWindowHandle = driver.getWindowHandle();
Set<String> allHandles = driver.getWindowHandles();

for (String handle : allHandles) {
   driver.switchTo().window(handle);
}

driver.switchTo().window(mainWindowHandle);

// Handle les frames/iframes
driver.switchTo().frame(0);
driver.switchTo().frame("frameName");
driver.switchTo().frame(frameElement);
driver.switchTo().defaultContent();

// Handle les alertes
Alert alert = driver.switchTo().alert();
alert.accept();
alert.dismiss();
String alertText = alert.getText();
alert.sendKeys("input text");
```

POM (Page Object Model) - Concept

- **Selenium WebDriver** : Approche standard, universelle
- **Encapsulation :** Chaque page = 1 classe Java héritant de BasePage
- **Locators privés** : Les **By** sont déclarés comme attributs privés
- **Méthodes publiques :** représentent les actions utilisateurs
- **Séparation des responsabilités :** logique de chaque page différente de la logique de test
- **Maintenabilité :** modification du DOM = un seul endroit à modifier

POM – BasePage

- **Convention de design recommandée par la communauté Selenium pour implémenter le pattern « POM »**
- Cette classe va implementer les méthodes utilitaires réutilisables
  - waitUntil
  - waitClickable
  - type
  - getText
- Elle va avoir la gestion du WebDriver
- Les attentes explicites communes

POM – BaseTest implémentée

- Elle ne gère que le navigateur maintenant !

```
public class BaseTest {

    protected WebDriver driver;

    @BeforeEach
    protected void setUp() {
        WebDriverManager.chromedriver().setup();
        driver = new ChromeDriver();
        driver.manage().window().maximize();
    }

    @AfterEach
    protected void tearDown() {
        if (driver != null) {
            driver.quit();
        }
    }

}
```

POM – BasePage implémentée

- Regroupement de méthodes utilitaires pour les réutiliser dans les tests
- Utilisation du « driver »
- Instanciation du « wait »
- Aucune méthode annotée de « **@Test** » !

```
public class BasePage {

    protected WebDriver driver;
    protected WebDriverWait wait;

    public BasePage(WebDriver driver) {
        this.driver = driver;
        this.wait = new WebDriverWait(driver, Duration.ofSeconds(10));
    }

    protected void goTo(String url) {
        driver.get(url);
    }

    protected WebElement waitUntil(By locator) {
        return wait.until(ExpectedConditions.visibilityOfElementLocated(locator));
    }

    protected WebElement waitClickable(By locator) {
        return wait.until(ExpectedConditions.elementToBeClickable(locator));
    }

    protected String getText(By locator) {
        return waitUntil(locator).getText();
    }

    protected void type(By locator, String text) {
        WebElement element = waitUntil(locator);
        element.click();
        element.clear();
        element.sendKeys(text);
    }

}
```

POM - Exemple “LoginPage”

- Elle étend « BasePage »
- On définit les propriétés des éléments relatifs à la page, sous forme de « By »
- Elle peut contenir des méthodes correspondant aux actions de la page
- Getter pour les propriétés

```
public class LoginPage extends BasePage {

    private final By usernameField = By.cssSelector("input[name='username']");
    private final By passwordField = By.cssSelector("input[name='password']");
    private final By submitButton = By.cssSelector("input[id='login-button']");

    public LoginPage(WebDriver driver) {
        super(driver);
    }

    public void login(String username, String password) {
        goTo(ROOT_URL);
        type(usernameField, username);
        type(passwordField, password);
        waitClickable(submitButton).click();
    }

    public By getUsernameField() {
        return usernameField;
    }

    public By getPasswordField() {
        return passwordField;
    }

    public By getSubmitButton() {
        return submitButton;
    }
}
```

POM - Exemple “LoginTest”

- Elle étend « BaseTest »
- Elle contient les méthodes annotées de « @Test »
- On instancie nos objets « Page » en fonction du besoin en passant le « driver » en paramètre (récupérer via « BaseTest »)

```
public class LoginTest extends BaseTest {

   @Test
   public void testLoginWithValidCredentials() {
      LoginPage loginPage = new LoginPage(driver);
      loginPage.login("user", "motdepasse");
    // assert on redirect
   }

}
```

POM – Notion de “chaînage”

- Le but est de fluidifier l’écriture du code
- Ici on a un exemple de « login », non chaîné, c’est assez lourd à écrire

```
@Test
public void testLogin() {
    LoginPage loginPage = new LoginPage(driver);
    loginPage.open();
    loginPage.enterUsername("username");
    loginPage.enterPassword("password");
    ProductsPage productsPage = loginPage.clickLogin();

    String title = productsPage.getTitle();
    assertEquals("Products", title);
}
```

POM – Notion de “chaînage”

- Il s’agit du même code que précédemment, sauf que cette fois, tout est « chaîné », cela permet d’écrire les tests plus simplement
- Pour cela la méthode « **open()** » renvoie « **this** », donc un objet de type « **LoginPage** », depuis lequel on récupère un objet de type « **ProductsPage** », auquel on appelle la méthode « **getTitle** » qui renvoie une chaîne de caractère

```
@Test
public void testLogin() {
   String title = new LoginPage(driver)
         .open()
         .login("username", "password ")
         .getTitle();

   assertEquals("Products", title);
}
```

Rapports - Allure

Avantages

- Framework de reporting pour tests automatisés
- Rapports visuels avec graphiques et statistiques
- Historique des tests et tendances
- Screenshots automatiques en cas d’erreur

Rapports - Allure

```
<dependency>
   <groupId>io.qameta.allure</groupId>
   <artifactId>allure-junit5</artifactId>
   <version>2.21.0</version>
</dependency>
```

```
<properties>
   <aspectj.version>
      1.9.21
   </aspectj.version>
</properties>
```

- Une dépendance est nécessaire pour installer “Allure” au projet
- “AspectJ” permet l’utilisation d’annotations afin d’enrichir les tests

Rapports – Annotations Allure

- **@Feature(“Fonction”)** : fonctionnalité spécifique
- **@Story(“User story”)** : Fait référence à une user story
- **@Severity(SeverityLevel.CRITICAL)** : importance du tests
- **@Description(“Super description”)** : décrit le rôle du test

Rapports - Allure Test

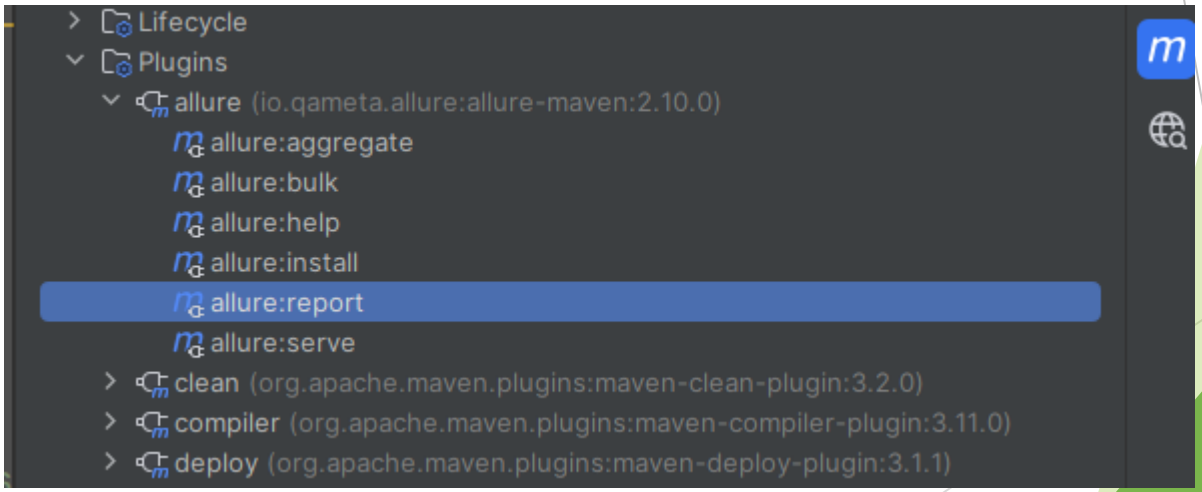
```
@Test
@Feature("Test login OK")
@Description("Tester la redirection vers la page login, lors de l'échec de la connexion")
void testFailedLogin() {
    goTo(loginPage);
    loginPage.isAt();
    LoginPage page = loginPage
            .setUsername("user@example.com")
            .setPassword("password123")
            .submit();

    assertThat(page.url()).isEqualTo("login?error");
    assertThat(page.isErrorMessageDisplayed()).isEqualTo(true);
}
```

Rapports - Allure Reports commandes

- **mvn clean test :** commande de lancement des tests
- **mvn allure:report :** génère le rapport de tests Allure
- **mvn allure:serve** : génère le rapport de test Allure et l'ouvre dans le navigateur
- **Note importante :** pensez à ajouter les dossiers "**.allure**" et "**allure-results**" dans le .gitignore du projet !

Rapports - Allure Reports Plugin

- **IntelliJ** permet, grâce à **Maven,** de pouvoir lancer la génération du rapport de tests directement via l'interface graphique
- CTRL + ALT + S : dans "marketplace" rechercher "allure"



Rapports – Allure screenshot

- **Allure** permet de prendre des screenshots de l'application lorsqu'il y a une erreur, et les intègre automatiquement au rapport généré

```
@Attachment(value = "Screenshot for the test", type = "image/png", fileExtension = ".png")
public byte[] saveScreenshot() {
   return ((TakesScreenshot) driver).getScreenshotAs(OutputType.BYTES);
}
```

- **@Attachment :** permet d'ajouter un screenshot au rapport généré

Rapports – Allure screenshot

- Afin que la prise de screenshot soit effectué correctement il faut ajouter le plugin “maven-surefire”, il se place dans la balise <**plugins**> du “**pom.xml**” :

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>3.0.0</version>
  <configuration>
    <systemPropertyVariables>
      <allure.results.directory>${project.build.directory}/allure-results</allure.results.directory>
    </systemPropertyVariables>
    <argLine>
      -javaagent:"${settings.localRepository}/org/aspectj/aspectjweaver/${aspectj.version}/aspectjweaver-${aspectj.version}.jar"
    </argLine>
  </configuration>
  <dependencies>
    <dependency>
      <groupId>org.aspectj</groupId>
      <artifactId>aspectjweaver</artifactId>
      <version>${aspectj.version}</version>
    </dependency>
  </dependencies>
</plugin>
<plugin>
  <groupId>io.qameta.allure</groupId>
  <artifactId>allure-maven</artifactId>
  <version>2.10.0</version>
  <configuration>
    <reportVersion>2.27.0</reportVersion>
  </configuration>
</plugin>
```

Rapports – Allure screenshot

- On va devoir modifier la function “**tearDown**” afin de prendre le screenshot avant que le driver ne soit fermer
- “**TestInfo**” est un objet propre à Junit, qui permet de récupérer des métadonnées sur le test en cours d’execution.
  On peut l’injecter automatiquement dans des méthodes annotées par “**@Test**”, “**@BeforeEach**”, “**@AfterEach**”

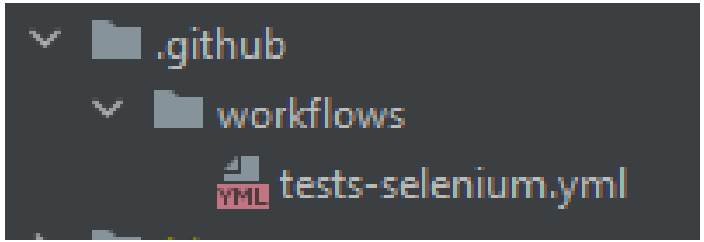
```
@AfterEach
protected void tearDown(TestInfo testInfo) {
    if (driver != null) {
        if (testInfo.getTags().contains("failed") ||
            testInfo.getTestMethod().isPresent()
        ) {
            try {
                saveScreenshot();
            } catch (Exception e) {
                System.out.println(e.getMessage());
            }
        }

        driver.quit();
    }
}
```

Selenium – CI/CD

- CI : Continuous Integration
- CD : Continuous Delivery
- Le principe est d'automatiser l'intégration, les tests et le déploiement du code. Chaque modification de code est automatiquement compilée, testée et validée dès qu'elle est push vers le dépôt, permettant ainsi une detection précoce des erreurs.
- Grâce à "GitHub page" on peut déployer le rapport de test directement en ligne après l'exécution de la CI.

Selenium – GitHub actions

- Afin d'automatiser les "**GitHub**" actions il faut créer un dossier "**.github**" à la racine du projet
- À l'intérieur de celui-ci on créé un autre dossier "**workflows**"
- À l'intérieur de celui-ci on créé un fichier "**.yml**" du nom de votre choix



Selenium – tests.yml

- La clé “**name**” représente le nom affiché dans les “**actions**”
- On declare ici le nom de la branche qui déclenchera l’action
- “**workflow_dispatch**” permet de pouvoir lancer l’action à la main, au cas où

```
name: Selenium Tests with Allure report

on:
  push:
    branches:
      - nom_branch
  workflow_dispatch:
```

Selenium – tests.yml

- On déclare ici les “**jobs**” à effectuer, ici le job s’appelle “**test**”, c’est un nom que vous définissez, il apparaîtra dans les actions de GitHub.
  On utilisera la dernière version de “ubuntu” pour exécuter la tâche
- “**steps**” représente les étapes à réaliser pour la tâche “**test**”
- Une étape peut avoir plusieurs sous-tâche, en “**yml**” on représente un tableau par un “**-**”.

```yaml
jobs:
  test:
    runs-on: ubuntu-latest

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

      - name: Set up JDK
        uses: actions/setup-java@v4
        with:
          distribution: 'temurin'
          java-version: '21'
          cache: 'maven'
```

Selenium – tests.yml

- On va ajouter le navigateur chrome dans la tâche
- On exécute les tests maven avec la commande

- Cette étape récupère l'historique des rapports Allure depuis la branche "**gh-pages**" et le stocke dans un dossier local de nom "**gh-pages**".
Cela permet de générer un graphique des évolutions des tests passes / échoués au fil du temps

```yaml
- name: Setup Chrome
  uses: browser-actions/setup-chrome@v1

- name: Run tests
  run: mvn clean test
  continue-on-error: true
  env:
    HEADLESS: true

- name: Get Allure history
  uses: actions/checkout@v4
  if: always()
  continue-on-error: true
  with:
    ref: gh-pages
    path: gh-pages
```

Selenium – tests.yml

- Pour générer le rapport Allure, il y a une “github action” permettant de le faire : **“simple-elf”.**
  Le **“with”** permet de configure les différents paramètres de l’action **“simple-elf/allure-report-action”** :
  - Où trouver les résultats
  - Où générer le rapport
  - Le chemin où récupérer l’historique
  - Où stocker le résultat final

- Puis l’étape de déploiement :
  - **publish_branch**: la branche où deployer (conventionnellement “gh-pages”
  - **publish-dir**: le même dossier que définit précédemment

```yaml
- name: Generate Allure Report
  uses: simple-elf/allure-report-action@master
  if: always()
  with:
    allure_results: target/allure-results
    allure_report: allure-report
    gh_pages: gh-pages
    allure_history: allure-history

- name: Deploy to GitHub Pages
  if: always()
  uses: peaceiris/actions-gh-pages@v3
  with:
    github_token: ${{ secrets.GITHUB_TOKEN }}
    publish_branch: gh-pages
    publish_dir: allure-history
```

Selenium – BaseTest

- Il faut modifier “**BaseTest**” pour ajouter la gestion de la variable d’environnement définie via notre “**job**” : “**HEADLESS**”

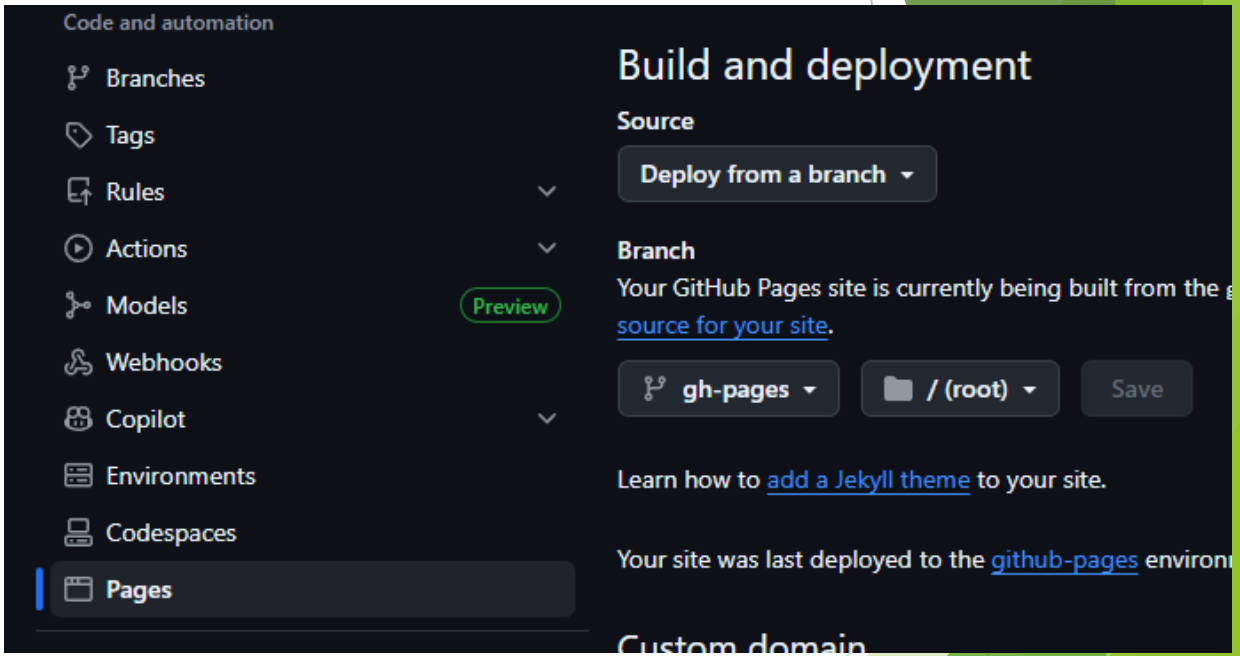
- Son rôle est de pouvoir lancer un chrome en mode “**HEADLESS**” pour les tests, et continuer d’avoir la version non-HEADLESS pour nos tests en local

```
@BeforeEach
protected void setUp() {
    WebDriverManager.chromedriver().setup();
    ChromeOptions options = new ChromeOptions();

    String headless = System.getenv("HEADLESS");
    if ("true".equalsIgnoreCase(headless)) {
        options.addArguments("--headless=new");
        options.addArguments("--no-sandbox");
        options.addArguments("--disable-dev-shm-usage");
        options.addArguments("--disable-gpu");
        options.addArguments("--window-size=1920,1080");
    }

    driver = new ChromeDriver(options);
    driver.manage().window().maximize();
}
```

Selenium – GitHub Pages

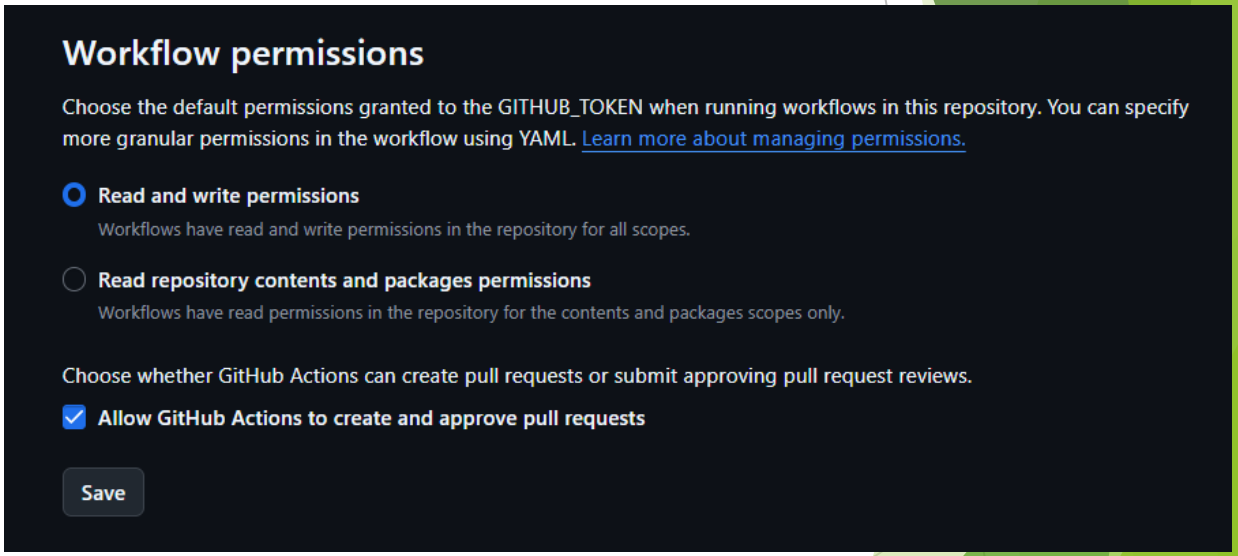
- Dernière configuration dans “**GitHub**” :
  - Settings
  - Pages
  - Choisir “**gh-pages**”
  - Save

  - Source
  - “Deploy from a branch”



Selenium – GitHub Pages

- Settings > Actions > General
  - “Read and write permissions”
  - “Allow GitHub Actions to create and approve pull requests”



Selenium Grid – Role

- C’est un outil de la suite Selenium qui permet d’exécuter des tests automatisés sur **plusieurs machines et navigateurs simultanément.** C’est une solution de parallélisation et de distribution des tests pour optimiser les temps d’exécution.

- On utilise des “nodes” (machines distantes) qui exécutent réellement les tests sur leurs navigateurs installés

Selenium Grid – Architecture

- Elle se compose en quatre processus :
  - **Router** : Point d'entrée central qui reçoit toutes les requêtes externes et les dirige vers les composants appropriés
  - **Distributor** : maintient un modèle des emplacements disponibles dans la grid et assignes les nouvelles session
  - **Session map** : conserve la correspondance entre l'ID de session et l'adresse du noeud exécutant la session
  - **Node** : exécute une session WebDriver, chaque noeud ayant un ou plusieurs slots

Selenium Grid – Testing Parallel

- Les avantages du “Testing parallel” :
  - Permet de **l’exécution parallèle** de plusieurs tests simultanément plutôt que séquentiellement.
  - Tests multi-plateformes simplifiés
  - Scalabilité
  - Gain de temps

Selenium Grid – Testing Parallel

```
RemoteWebDriver driver = new RemoteWebDriver(
      new URL("http://localhost:4444"),
      new ChromeOptions()
);
driver.get("https://www.selenium.dev");
```

- Utilisation du **“RemoteWebDriver”**, au lieu du **“WebDriver”**

- “URL” définie via un environnement Docker (pour les nodes)

Selenium Grid – Testing parallel

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "https://testng.org/testng-1.0.dtd">
<suite thread-count="2" parallel="methods" name="Suite">
   <test name="Test">
      <classes>
         <class name="parallelExecution.ParallelTestExecution"/>
      </classes>
   </test>
</suite>
```

- “TestNG”
  - C’est un framework permettant d’automatiser les tests parallèles, notamment avec Selenium.
  - Il utilise une configuration XML (“testng.xml“)

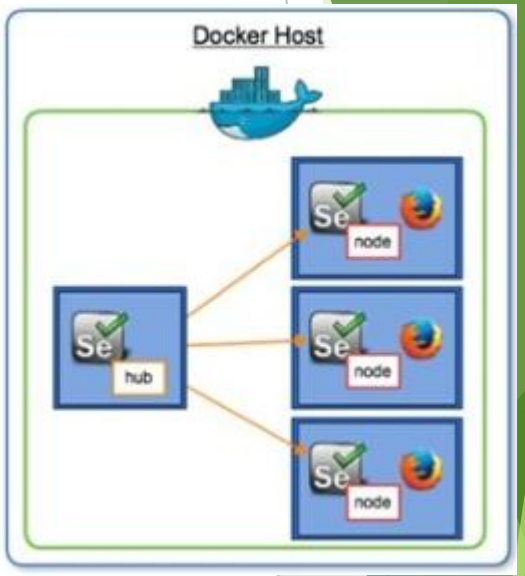
Selenium Grid – Testing parallel

- Pattern “ThreadLocal”
- Ici le but est de définir nos “WebDriver” dasns des threads différents

```
public class DriverFactory {
    private static final ThreadLocal<WebDriver> webDriver = new ThreadLocal<>();

    static void set(WebDriver driver) {
        webDriver.set(driver);
    }

    public static WebDriver get() {
        return Optional.ofNullable(webDriver.get())
                .orElseThrow(() -> new IllegalStateException("Driver NOT initialised"));
    }

    static void remove() {
        webDriver.remove();
    }
}
```

Selenium Grid – Testing parallel

- On peut cumuler **“TestNG”** et le pattern **“ThreadLocal”** avec docker, encapsule notre **“Hub”** Selenium et les conteneurs des WebDriver
- Ainsi nos tests sont plus scalable, car il nous suffira de lancer un nouveau conteneur docker pour augmenter le nombre de tests réalisés simultanéments



Selenium Grid – Testing parallel

- Exemple de “**docker-compose.yml**”
- **GRID_MAX_SESSION** : représente le nombre maximum de tests WebDriver en même temps
- **GRID_BROWSER_TIMEOUT** : délai maximum avant qu’une commande navigateur soit considérée comme échouée
- **GRID_TIMEOUT** : durée maximale pendant laquelle une session peut rester inactive avant d’être automatiquement fermée
- **SE_EVENT_BUS_HOST** : nom du conteneur docker contenant le selenium hub
- **SE_EVENT_BUS_PUBLISH_PORT** : port pour publier les évènements du hub
- **SE_EVENT_BUS_SUBSCRIBE_PORT** : port abonnement aux évènements du hub
- **SE_EVENT_MAX_SESSIONS** : nombre maximum de session **de navigateurs par noeud**

```yaml
version: "3"
services:
  selenium-hub:
    image: selenium/hub:4.x
    ports:
      - "4444:4444"
    environment:
      - GRID_MAX_SESSION=10
      - GRID_BROWSER_TIMEOUT=300
      - GRID_TIMEOUT=300

  chrome:
    image: selenium/node-chrome:4.x
    depends_on:
      - selenium-hub
    environment:
      - SE_EVENT_BUS_HOST=selenium-hub
      - SE_EVENT_BUS_PUBLISH_PORT=4442
      - SE_EVENT_BUS_SUBSCRIBE_PORT=4443
      - SE_NODE_MAX_SESSIONS=3
    deploy:
      replicas: 3
```