

SEPTEMBRE 2024

▼ VERSION 0.1

FORMATION PLAYWRIGHT

Sommaire

CONTENTS	2
INTRODUCTION	7
1. Contexte.....	8
Playwright, le metteur en scène des tests.	8
Qu'est-ce que Playwright ?	8
Pourquoi utiliser Playwright ?.....	9
Comparaison avec d'autres outils d'automatisation.....	9
Environnements et cas d'utilisation	10
2. Installation	12
Objectif	12
Prérequis (Node.js, npm).....	12
Installation de Playwright.....	13
Configuration du projet.....	14
Mise en pratique guidée	18
PRISE EN MAIN	21
3. Automatisation	22
Objectif	22
Les bases de l'automatisation web.....	22
Sélection des éléments.....	23
Gestion des actions utilisateur.....	24
Gestion des assertions	25
Conclusion	25
4. Codegen	27

Objectif	27
Introduction à Playwright Codegen.....	27
Lancement de Playwright Codegen.....	28
Enregistrement d'un test.....	28
Génération de Locator.....	29
Émulation des environnements et des états.....	30
Bonnes pratiques avec Codegen	31
Conclusion	32
5. Scénario avancés	33
Objectif	33
Manipulation des frames et des iframes	33
Gestion des fenêtres multiples et des contextes de navigation	34
Gestion des dialogues, alertes et pop-ups	35
Gestion des temps d'attente	36
6. API & Transactions.....	37
Objectif	37
Interception des requêtes et des réponses HTTP	37
Simulation de réponses réseau	39
Tests avec API mockées	41
Conclusion	43
7. Gestion des fichiers et des téléchargements.....	44
Objectif	44
Automatisation du téléchargement de fichiers	44
Automatisation de l'upload de fichiers	45
Manipulation des fichiers téléchargés.....	46
Conclusion	47
8. Exécution en parallèle & dans le cloud	48
Objectif	48
Configuration des tests parallèles	48

Tests sur différents navigateurs	49
Intégration avec des plateformes d'exécution de tests dans le cloud	51
Mise en pratique guidée	51
Conclusion	54
9. Rapports & logs.....	56
Objectif	56
Génération de rapports de tests (HTML, JSON)	56
Génération de rapports JSON.....	57
Combiner plusieurs reporters	58
Analyse des logs et des captures d'écran.....	58
Gestion des enregistrements vidéo des sessions de test	60
Conclusion	61
10.....Tests d'accessibilité & performance	62
Objectif	62
Introduction aux tests d'accessibilité avec Playwright	62
Test des performances des pages.....	64
Conclusion	66
11.....Conseils & bonnes pratiques	67
Objectif	67
Organisation des tests et structuration du code	67
Tests robustes et résilients (éviter les faux positifs/négatifs)	69
Maintenance des tests et gestion des changements dans les applications	70
Conclusion	73
12.....QCM	75
Correction QCM	91

13.....	TP01
.....	112
Mise en situation.....	112
14.....	TP02
.....	116
Mise en situation.....	116
15.....	TP03
.....	118
Mise en situation.....	118
16.....	TP04
.....	121
Mise en situation.....	121
17.....	TP05
.....	125
Mise en situation.....	125
18.....	TP06
.....	129
Mise en situation.....	129
19.....	TP07
.....	132
Mise en situation.....	132

Versioning

VERSION	DATE	LIBELLÉ DE LA MISE À JOUR	AUTEUR
0.1	Septembre 2024	Création du document	Karim Hafsi

INTRODUCTION

1. Contexte

PLAYWRIGHT, LE METTEUR EN SCENE DES TESTS.

Au moment où ce document est rédigé Playwright est l'outil d'automatisation des tests web le plus moderne. Développé par Microsoft, il permet d'effectuer des tests de bout en bout (end-to-end) sur des applications web en simulant les interactions de l'utilisateur à travers plusieurs navigateurs (Chromium, Firefox, WebKit). Il est réputé pour être rapide, fiable et flexible.

QU'EST-CE QUE PLAYWRIGHT ?

Grâce à Playwright, les développeurs et les testeurs peuvent écrire des tests qui fonctionnent sur différents navigateurs en choisissant un des nombreux langages supportés (JavaScript, TypeScript, Python, Java, .Net), rendant ainsi le processus de test plus efficace. De plus, Playwright supporte des fonctionnalités avancées comme l'interception des transactions réseau, les tests sur plusieurs onglets ou fenêtres, et la manipulation d'éléments complexes comme les iframes et les pop-ups.

POURQUOI UTILISER PLAYWRIGHT ?

Playwright se distingue des autres outils d'automatisation par plusieurs atouts :

- Multi-navigateurs : Il supporte nativement plusieurs webdrivers (Chromium, Firefox, WebKit) et permet de tester une application sur différentes navigateurs, permettant une meilleure couverture des tests.
- Fiabilité : Playwright attend automatiquement que les éléments de la page soient prêts (chargement, animations terminées, etc.) avant d'exécuter les actions. Cela permet d'éliminer la majorité des problèmes liés aux temps d'attente et rend les tests plus stables. La fonction `Retry`, qui rejoue un test selon une condition prédefinie est aussi nativement disponible avec Playwright.
- Parallélisme et performance : Grâce à ses capacités d'exécution parallèle, Playwright permet de réaliser des tests rapidement, même à grande échelle, tout en assurant une répartition efficace des ressources. Pour y parvenir, il exécute plusieurs « workers » qui s'exécutent en même temps. Par défaut, les fichiers de test sont exécutés **en parallèle**. Les tests dans un seul fichier sont exécutés dans l'ordre, dans le même processus de travail.
- Fixtures : Playwright est basé sur le concept de fixtures. Elles sont utilisées pour établir l'environnement de chaque test, donnant au test tout ce dont il a besoin et rien d'autre. Les fixtures de test sont isolés entre les tests. Avec les elles, vous pouvez regrouper les tests en fonction de leur Feature, au lieu **d'une** configuration commune.

COMPARAISON AVEC D'AUTRES OUTILS D'AUTOMATISATION

Bien que Playwright partage beaucoup de similarités avec des outils comme Selenium ou Puppeteer, il apporte plusieurs avantages qui le rendent plus adapté à certains projets :

SELENIUM

Bien que Selenium soit l'un des outils d'automatisation les plus populaires à l'heure actuelle et offre une compatibilité multi-navigateurs, Playwright est cependant plus performant et offre des fonctionnalités plus récentes, comme la gestion automatique des temps d'attente, la capture vidéo, le reporting... De plus, Playwright se concentre sur la simplicité et la rapidité, là où Selenium peut nécessiter plus de configuration pour obtenir des tests stables et fiables.

PUPPETEER

Playwright est une évolution de Puppeteer, offrant des capacités multi-navigateurs là où Puppeteer se limite principalement à Chromium. Si Puppeteer est idéal pour des projets centrés sur Chrome, Playwright offre une couverture plus large en prenant en charge également Firefox et WebKit (Safari), ce qui le rend plus polyvalent.

ENVIRONNEMENTS ET CAS D'UTILISATION

Playwright est conçu pour fonctionner dans des environnements variés, qu'il s'agisse de tests en local ou d'intégration avec des systèmes de CI/CD. Voici quelques cas d'utilisation où Playwright peut être utile :

- Tests de régression : S'assurer que les nouvelles modifications n'introduisent pas de bugs dans les fonctionnalités existantes.
- Tests cross-navigateurs : Garantir que l'application fonctionne de manière cohérente sur différents navigateurs.
- Tests d'API et interactions réseau : Intercepter et vérifier les requêtes réseau pour s'assurer que l'application interagit correctement avec ses services backend.
- Tests de performance : Mesurer les temps de chargement et d'interaction pour améliorer les performances de l'application et identifier les points de latence dans une optique d'optimisation.

Avec Playwright, les équipes de développement disposent d'un outil puissant pour automatiser les tests web, leur permettant de livrer des applications plus rapidement, avec moins de bugs et une meilleure expérience utilisateur. **Et en plus c'est gratuit !**

A TOI DE JOUER !

Répondez aux questions 1 à 5 du QCM de fin de document

2. Installation

OBJECTIF

Dans ce chapitre, nous allons voir comment préparer l'environnement de travail pour utiliser Playwright, installer l'outil, configurer un projet, et exécuter nos premiers tests. En passant par l'installation des prérequis nécessaires, la configuration initiale du projet, et l'exécution d'un test basique pour s'assurer que tout fonctionne correctement.

PREREQUIS (NODE.JS, NPM)

Avant d'installer Playwright, il est nécessaire de s'assurer que Node.js est installé sur votre machine, car pour cette formation nous utiliserons Playwright avec l'écosystème JavaScript/TypeScript. Node.js permet d'exécuter des scripts côté serveur et gère les dépendances via le gestionnaire de paquets npm.

Installer Node.js :

- Rendez-vous sur le site officiel de Node.js : <https://nodejs.org>
- Téléchargez la version LTS (Long Term Support) pour une stabilité maximale.
- Suivez les instructions pour l'installation en fonction de votre système d'exploitation.

Vérifier l'installation :

Une fois l'installation terminée, ouvrez un terminal et exécutez les commandes suivantes pour vérifier que Node.js et npm sont bien installés :

```
PS D:\ZENITY> node -v  
v19.8.1  
PS D:\ZENITY> npm -v  
9.5.1
```

Comme ci-dessus, cela affichera la version installée de Node.js et de npm.

INSTALLATION DE PLAYWRIGHT

Une fois Node.js installé, nous allons installer Playwright à l'aide de npm. Playwright se trouve dans le registre npm, ce qui le rend facile à installer.

Initialiser un projet Node.js :

Si vous n'avez pas encore de projet Node.js, commencez par initialiser un nouveau projet :

```
PS D:\ZENITY\Projects> mkdir formation-playwright  
  
Directory: D:\ZENITY\Projects  
  
Mode           LastWriteTime         Length Name  
----           -----          -----  
d----  9/16/2024  2:53 PM            formation-playwright  
  
PS D:\ZENITY\Projects> cd .\formation-playwright\  
PS D:\ZENITY\Projects\formation-playwright> npm init -y  
Wrote to D:\ZENITY\Projects\formation-playwright\package.json:  
  
{  
  "name": "formation-playwright",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "test": "echo \\\"Error: no test specified\\\" && exit 1"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC"  
}
```

Installer Playwright :

Pour installer Playwright et ses dépendances, exécutez la commande suivante :

```
PS D:\ZENITY\Projects\formation-playwright> npm install playwright
added 2 packages, and audited 3 packages in 1s
found 0 vulnerabilities
```

Télécharger les navigateurs :

Lors de l'installation, Playwright télécharge automatiquement les binaires des navigateurs (Chromium, Firefox, WebKit). Si vous souhaitez les télécharger manuellement, utilisez la commande suivante :

```
PS D:\ZENITY\Projects\formation-playwright> npx playwright install
Removing unused browser at C:\Users\HAFSI\AppData\Local\npm-playwright\chromium-1055
Removing unused browser at C:\Users\HAFSI\AppData\Local\npm-playwright\chromium-1060
Removing unused browser at C:\Users\HAFSI\AppData\Local\npm-playwright\ffmpeg-1008
Removing unused browser at C:\Users\HAFSI\AppData\Local\npm-playwright\firefox-1391
Removing unused browser at C:\Users\HAFSI\AppData\Local\npm-playwright\firefox-1403
Removing unused browser at C:\Users\HAFSI\AppData\Local\npm-playwright\webkit-1811
Removing unused browser at C:\Users\HAFSI\AppData\Local\npm-playwright\webkit-1837
Downloading Chromium 129.0.6668.29 (playwright build v1134) from https://playwright.azureedge.net/builds/chromium/1134/chromium-win64.zip
139 MiB [=====] 100% 0.0s
Chromium 129.0.6668.29 (playwright build v1134) downloaded to C:\Users\HAFSI\AppData\Local\npm-playwright\chromium-1134
Downloading FFmpeg playwright build v1010 from https://playwright.azureedge.net/builds/ffmpeg/1010/ffmpeg-win64.zip
1.3 MiB [=====] 100% 0.0s
FFmpeg playwright build v1010 downloaded to C:\Users\HAFSI\AppData\Local\npm-playwright\ffmpeg-1010
Downloading Firefox 130.0 (playwright build v1463) from https://playwright.azureedge.net/builds/firefox/1463/firefox-win64.zip
84.6 MiB [=====] 100% 0.0s
Firefox 130.0 (playwright build v1463) downloaded to C:\Users\HAFSI\AppData\Local\npm-playwright\firefox-1463
Downloading Webkit 18.0 (playwright build v2070) from https://playwright.azureedge.net/builds/webkit/2070/webkit-win64.zip
46.3 MiB [=====] 100% 0.0s
webkit 18.0 (playwright build v2070) downloaded to C:\Users\HAFSI\AppData\Local\npm-playwright\webkit-2070
```

CONFIGURATION DU PROJET

Maintenant que Playwright est installé, nous allons configurer le projet pour organiser les tests. Playwright propose une configuration par défaut, mais il est possible de personnaliser certains aspects via un fichier de configuration dédié.

Initialiser la configuration Playwright

Playwright offre une commande pratique pour initialiser la configuration :

Cette commande crée un dossier `tests` avec quelques exemples de tests et un fichier `playwright.config.ts` pour la configuration. Ce fichier est au cœur de votre projet de tests et contient des options comme les navigateurs à tester, les options d'exécution, et les chemins des rapports de tests.

```
PS D:\ZENITY\Projects\formation-playwright> npm init playwright@latest
Need to install the following packages:
  create-playwright@1.17.133
Ok to proceed? (y) y
Getting started with writing end-to-end tests with Playwright:
Initializing project in '.'.
✓ Do you want to use TypeScript or JavaScript? · TypeScript
✓ Where to put your end-to-end tests? · tests
✓ Add a GitHub Actions workflow? (y/N) · false
✓ Install Playwright browsers (can be done manually via 'npx playwright install')? (Y/n) · true
Installing Playwright Test (npm install --save-dev @playwright/test)...

added 2 packages, and audited 5 packages in 487ms

found 0 vulnerabilities
Installing Types (npm install --save-dev @types/node)...

added 2 packages, and audited 7 packages in 531ms

found 0 vulnerabilities
Writing playwright.config.ts.
Writing tests\example.spec.ts.
Writing tests-examples\demo-todo-app.spec.ts.
Writing package.json.
Downloading browsers (npx playwright install)...
✓ Success! Created a Playwright Test project at D:\ZENITY\Projects\formation-playwright
```

Inside that directory, you can run several commands:

`npx playwright test`
Runs the end-to-end tests.

`npx playwright test --ui`
Starts the interactive UI mode.

`npx playwright test --project=chromium`
Runs the tests only on Desktop Chrome.

`npx playwright test example`
Runs the tests in a specific file.

`npx playwright test --debug`
Runs the tests in debug mode.

`npx playwright codegen`
Auto generate tests with Codegen.

We suggest that you begin by typing:

`npx playwright test`

And check out the following files:

- .\tests\example.spec.ts - Example end-to-end test
- .\tests-examples\demo-todo-app.spec.ts - Demo Todo App end-to-end tests
- .\playwright.config.ts - Playwright Test configuration

Visit <https://playwright.dev/docs/intro> for more information. ♦

Happy hacking! 🎉

Comprendre playwright.config.ts :

Le fichier de configuration permet de personnaliser plusieurs aspects de l'exécution des tests. Voici quelques-unes des options les plus importantes :

Cette commande crée un dossier `tests` avec quelques exemples de tests et un fichier `playwright.config.ts` pour la configuration. Ce fichier est au cœur de votre projet de tests et contient des options comme les navigateurs à tester, les options d'exécution, et les chemins des rapports de tests.

Navigateurs

Vous pouvez spécifier les navigateurs que Playwright doit utiliser pour les tests :

```
34  /* Configure projects for major browsers */
35  projects: [
36    {
37      name: 'chromium',
38      use: { ...devices['Desktop Chrome'] },
39    },
40    {
41      name: 'firefox',
42      use: { ...devices['Desktop Firefox'] },
43    },
44    {
45      name: 'webkit',
46      use: { ...devices['Desktop Safari'] },
47    },
48  },
```

Options globales

Comme le délai maximal d'attente pour les actions ou les timeouts :

```

export default defineConfig({
  testDir: './tests',
  /* Run tests in files in parallel */
  fullyParallel: true,
  /* Fail the build on CI if you accidentally left test.only in the source code. */
  forbidOnly: !!process.env.CI,
  /* Retry on CI only */
  retries: process.env.CI ? 2 : 0,
  /* Opt out of parallel tests on CI. */
  workers: process.env.CI ? 1 : undefined,
  /* Reporter to use. See https://playwright.dev/docs/test-reporters */
  reporter: 'html',
  /* Shared settings for all the projects below. See https://playwright.dev/docs/api/class-testoptions. */
  timeout: 30000, // Timeout par défaut de 30s pour chaque test

  use: {
    headless: true, // Exécuter les tests en mode headless (sans interface graphique)
    viewport: { width: 1280, height: 720 }, // Taille de la fenêtre du navigateur

    /* Base URL to use in actions like `await page.goto('/')`. */
    // baseURL: 'http://127.0.0.1:3000',

    /* Collect trace when retrying the failed test. See https://playwright.dev/docs/trace-viewer */
    trace: 'on-first-retry',
  },
},

```

Rapports et traces

Vous pouvez configurer des options pour enregistrer des captures d'écran, des vidéos ou des rapports en cas d'échec des tests.

Modifier playwright.config.ts

Pour personnaliser votre configuration, vous pouvez ajuster le fichier en fonction de vos besoins. Par exemple, pour exécuter uniquement des tests sur Chrome en mode visible (non headless), vous pouvez modifier la configuration comme suit :

```

/* Configure projects for major browsers */
projects: [
  {
    name: 'chromium',
    use: { browserName: 'chromium', headless: false },
  },
],

```



Mise en pratique guidée

Une fois la configuration en place, il est temps d'écrire et d'exécuter votre premier test. Voici un exemple simple qui vérifie le titre d'une page web.

1. Écrire un test simple :

Créez un fichier de test dans le dossier `tests`, par exemple `tests/mep1.test.ts` :

```
tests > 🎨 mep1.test.ts > ...
1  import { test, expect } from '@playwright/test';
2
3  test('vérifie le titre de la page', async ({ page }) => {
4    // Aller à l'URL de la page
5    await page.goto('https://zenity.fr');
6
7    // Vérifier que le titre de la page est correct
8    const title = await page.title();
9    console.log(title);
10   expect(title).toBe('Pure Player du Test Logiciel | Zenity');
11});
```

2. Exécuter le test :

```
PS D:\ZENITY\Projects\formation-playwright> npx playwright test mep1
Running 1 test using 1 worker
[chromium] > mep1.test.ts:3:5 > vérifie le titre de la page
Pure Player du Test Logiciel | Zenity
  1 passed (1.9s)

To open last HTML report run:
  npx playwright show-report
```

Cette commande va chercher tous les fichiers de test dans le dossier `tests` et les exécuter. Vous verrez un rapport des tests dans le terminal, indiquant si le test a réussi ou échoué.

3. Consulter les rapports :

Playwright génère également un rapport de test interactif que vous pouvez consulter avec la commande :

```
PS D:\ZENITY\Projects\formation-playwright> npx playwright show-report
```

```
Serving HTML report at http://localhost:9323. Press Ctrl+C to quit.
```

Cela ouvrira un rapport HTML dans votre navigateur, vous permettant de voir les détails des tests, y compris les captures d'écran et vidéos en cas d'échec.

The screenshot shows a detailed test report for a file named 'mep1.test.ts'. At the top, there's a search bar and a status bar indicating 'All 1' tests passed ('Passed 1'), with 'Failed 0', 'Flaky 0', and 'Skipped 0'. Below this, the title 'vérifie le titre de la page' is displayed, along with the file path 'mep1.test.ts:3' and the browser used, 'chromium'. A green checkmark next to 'Run' indicates the test was executed successfully. The main content area is titled 'Test Steps' and contains a hierarchical tree of test steps. It includes sections for 'Before Hooks' (with 'fixture: browser', 'fixture: context', and 'fixture: page' all passing), a single test step 'page.goto(https://zenity.fr)' which failed ('926ms'), another 'page.title' step that passed ('5ms'), and 'expect.toBe' which also passed ('0ms'). Following this is an 'After Hooks' section with 'fixture: page' and 'fixture: context' both passing ('32ms'). At the bottom, there's an 'Attachments' section showing a single 'stdout' entry. The entire interface has a clean, modern design with dark mode styling.

Vous avez maintenant installé et configuré Playwright avec succès, et vous êtes prêt à écrire des tests automatisés. Le prochain chapitre approfondira l'automatisation des interactions avec les pages web et la manière de structurer vos tests de manière plus avancée.



Super, tu avances bien !

A TOI DE JOUER !

Répondez aux questions 5 à 10 du QCM de fin de document et exécutez le TP01.

PRISE EN MAIN

3. Automatisation

OBJECTIF

Ce chapitre explore les principes fondamentaux de l'automatisation des tests web avec Playwright. Nous aborderons les concepts clés, tels que la navigation dans les pages, l'interaction avec les éléments, la sélection des éléments à l'aide de différents sélecteurs, et la gestion des assertions pour valider les comportements et les états de l'application.

LES BASES DE L'AUTOMATISATION WEB

Navigation :

Pour naviguer vers une URL spécifique, utilisez la méthode `goto` :

```
await page.goto('https://zenity.fr');
```

Cette méthode charge la page et attend que le chargement soit complet. Vous pouvez également spécifier des options comme le timeout pour la navigation.

Interaction avec les éléments :

Playwright permet d'interagir avec divers éléments sur la page. Vous pouvez cliquer sur des boutons, entrer du texte dans des champs de saisie, ou sélectionner des options dans des menus déroulants.

Cliquer sur un élément :

```
await page.click('button#submit');
```

Saisir du texte :

```
await page.fill('input[name="username"]', 'monUsername');
```

Sélectionner une option dans un menu déroulant :

```
await page.selectOption('select#country', 'FR');
```

Ces interactions simulent les actions d'un utilisateur réel sur l'application web, vous permettant de tester des scénarios variés.

SELECTION DES ELEMENTS

Sélecteurs CSS :

Les sélecteurs CSS sont les plus couramment utilisés pour cibler des éléments sur une page. Ils permettent de sélectionner les éléments en fonction de leurs classes, ID, attributs, ou relations hiérarchiques.

```
await page.click('.button-primary'); // Sélectionner un élément avec la classe 'button-primary'  
await page.fill('#search-input', 'Playwright'); // Sélectionner un élément avec l'ID 'search-input'
```

Cette méthode charge la page et attend que le chargement soit complet. Vous pouvez également spécifier des options comme le timeout pour la navigation.

XPath :

XPath est un langage de requête utilisé pour localiser des éléments dans un document XML ou HTML. Playwright supporte XPath, ce qui peut être utile pour des sélections complexes ou pour des éléments avec des attributs dynamiques.

```
await page.click('//button[text()="Submit"]); // Sélectionner un bouton avec le texte 'Submit'
```

Sélecteurs de texte :

Playwright permet également de sélectionner des éléments basés sur leur texte visible :

```
await page.locator('//button[text()="Submit"]'); // sélectionner un bouton avec le texte 'Submit'
```

Les sélecteurs CSS sont généralement préférés pour leur simplicité et leur performance, mais XPath et les sélecteurs de texte offrent des options supplémentaires pour des scénarios spécifiques.

GESTION DES ACTIONS UTILISATEUR

Playwright permet de simuler diverses actions utilisateur pour tester les interactions dans l'application :

Déplacement libre de la souris :

Pour simuler un clic sur un élément, utilisez la méthode `click` :

```
// Utiliser 'page.mouse' pour tracer un carré de 100px x 100px.  
await page.mouse.move(0, 0);  
await page.mouse.down();  
await page.mouse.move(0, 100);  
await page.mouse.move(100, 100);  
await page.mouse.move(100, 0);  
await page.mouse.move(0, 0);  
await page.mouse.up();
```

Cette méthode charge la page et attend que le chargement soit complet. Vous pouvez également spécifier des options comme le timeout pour la navigation.

Défilement et mouvements :

Pour simuler le défilement ou les mouvements de la souris, utilisez les méthodes `scroll` ou `hover` :

```
await page.locator('div#section').scrollIntoViewIfNeeded;  
await page.hover('button#tooltip');
```

Ces actions permettent de tester le comportement de l'application face aux interactions des utilisateurs, garantissant que les fonctionnalités sont correctement implémentées.

GESTION DES ASSERTIONS

Les assertions sont utilisées pour vérifier que l'application fonctionne comme prévu. Playwright offre des méthodes d'assertion intégrées pour valider les états et les comportements des éléments :

Vérification de la visibilité des éléments :

```
await expect(page.locator('text=Bienvenue')).toBeVisible();
```

Vérification du texte d'un élément :

```
await expect(page.locator('h1')).toHaveText('Titre de la page');
```

Vérification de l'état des éléments :

```
await expect(page.locator('input#checkbox')).toBeChecked(); // Vérifier si une case à cocher est sélectionnée
```

Vérification des URL et des titres de page :

```
await expect(page).toHaveURL('https://example.com/dashboard');
await expect(page).toHaveTitle('Dashboard');
```

Les assertions permettent de s'assurer que les actions effectuées produisent les résultats attendus, et sont essentielles pour valider les fonctionnalités des tests automatisés.

CONCLUSION

En comprenant ces principes fondamentaux, vous pouvez commencer à écrire des tests automatisés efficaces avec Playwright, en simulant des interactions utilisateur, en sélectionnant des éléments de manière précise, et en validant les comportements et les états de votre application web. Le prochain chapitre explorera la gestion des scénarios de test avancés, y compris les frames, les fenêtres multiples et les dialogues.

A TOI DE JOUER !

Répondez donc aux questions 11 à 14 du QCM de fin de document et exécutez le TP02.

4. Codegen

OBJECTIF

L'outil **Codegen** est un générateur de tests qui permet de capturer et d'enregistrer des actions utilisateurs directement depuis le navigateur. Il est particulièrement utile pour les débutants en automatisation, ainsi que pour les utilisateurs confirmés souhaitant créer rapidement des scripts de tests à partir de leurs interactions avec une page web. Ce chapitre couvre l'installation, l'utilisation et les bonnes pratiques liées à cet outil puissant.

INTRODUCTION A PLAYWRIGHT CODEGEN

Playwright Codegen est une fonctionnalité intégrée à Playwright qui permet de générer automatiquement des scripts de test basés sur les interactions utilisateur.

Lorsqu'il est activé, il ouvre deux fenêtres :

- Une fenêtre de navigateur dans laquelle vous pouvez interagir avec le site que vous souhaitez tester.
- Une fenêtre Playwright Inspector où le script correspondant à vos actions est généré en temps réel. Vous pouvez y enregistrer, copier, effacer et configurer le code généré, tout en choisissant le langage de votre choix (TypeScript, JavaScript, Python, C#).

Avantages de Codegen :

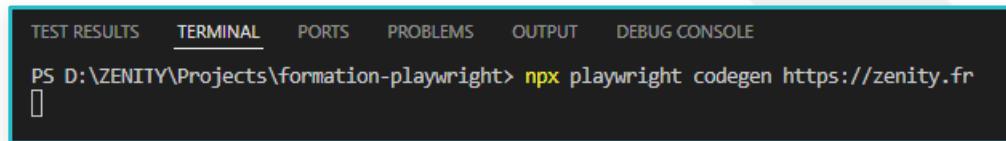
- Il facilite la création rapide de tests en capturant les actions utilisateur.
- Il génère des Locator robustes, minimisant ainsi les risques de défaillances de tests dues à des sélecteurs instables.
- Il permet de simuler différents scénarios grâce à l'émulation (vue mobile, localisation, état d'authentification, etc.).

Utilisations typiques de Codegen :

- Générer des tests pour les applications web.
- Enregistrer des interactions pour identifier les sélecteurs.
- Débuter rapidement l'automatisation d'une fonctionnalité.

LANCLEMENT DE PLAYWRIGHT CODEGEN

Pour lancer le générateur de tests, utilisez la commande suivante en précisant l'URL de la page à tester :



```
TEST RESULTS TERMINAL PORTS PROBLEMS OUTPUT DEBUG CONSOLE
PS D:\ZENITY\Projects\formation-playwright> npx playwright codegen https://zenity.fr
[]
```

Note : L'URL est facultative. Vous pouvez également lancer codegen sans URL, puis naviguer vers le site de votre choix depuis le navigateur ouvert.

Lorsque la commande est exécutée :

- La fenêtre du navigateur s'ouvre et vous pouvez interagir avec le site.
- La fenêtre Playwright Inspector affiche en temps réel le code correspondant à vos actions.

ENREGISTREMENT D'UN TEST

Enregistrer un test avec Codegen est voulu simple et intuitif. Suivez les étapes ci-dessous :

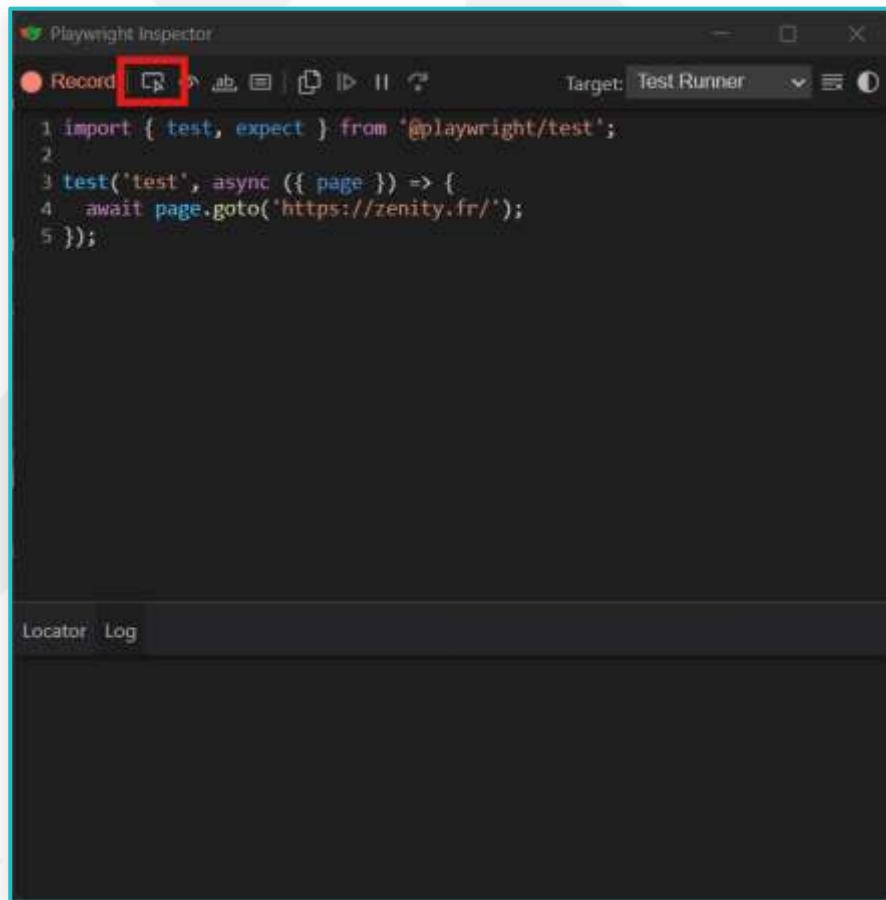
1. Lancer Codegen avec l'URL de la page que vous souhaitez tester.
2. Interagir avec la page web : chaque action que vous effectuez (clic, saisie de texte, navigation) est automatiquement convertie en code.
3. Génération des assertions :
 - Cliquez sur une des icônes dans la barre d'outils de Playwright Inspector.
 - Cliquez ensuite sur l'élément de la page pour générer une assertion :
 - assert visibility : vérifie que l'élément est visible.
 - assert text : vérifie que l'élément contient un texte spécifique.

- assert value : vérifie que l'élément possède une valeur particulière.
4. Arrêter l'enregistrement : une fois que toutes les actions souhaitées ont été effectuées, appuyez sur le bouton record pour arrêter l'enregistrement.
 5. Copier le code généré : utilisez le bouton copy pour copier le code dans votre éditeur et effectuer des modifications si nécessaire.

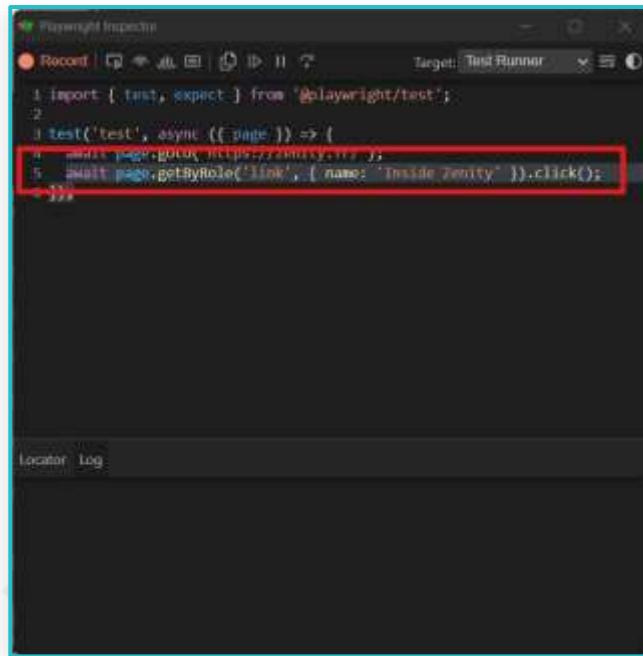
GENERATION DE LOCATOR

Un des avantages de Codegen est la possibilité de générer des locators précis et robustes pour les éléments de la page :

1. Démarrer l'enregistrement avec Codegen.
2. Cliquez sur le bouton `Pick Locator` qui apparaît dans la fenêtre de l'inspecteur.



3. Survolez les éléments dans la fenêtre de navigateur pour voir les Locator suggérés.
4. Cliquez sur l'élément pour générer le Locator correspondant.
5. Le Locator s'affiche dans le Locator Playground à côté du bouton `Pick Locator`. Vous pouvez alors ajuster le Locator dans ce champ et voir les éléments correspondants surlignés dans le navigateur.



```
Playwright Inspector
Record Target: Test Runner
1 import { test, expect } from '@playwright/test';
2
3 test('test', async ({ page }) => {
4   await page.goto('https://zenity.fr');
5   await page.getByRole('link', { name: 'Inside Zenity' }).click();
6 })
7
```

ÉMULATION DES ENVIRONNEMENTS ET DES ETATS

Playwright Codegen prend également en charge l'émulation pour tester des scénarios spécifiques comme :

- Émulation de la vue mobile : testez l'application sur différents types d'appareils mobiles.
- Émulation de la localisation géographique : simulez des tests pour vérifier le comportement de l'application en fonction de la position géographique.
- Émulation de l'état d'authentification : enregistrez des scénarios où l'utilisateur est déjà authentifié.
- Émulation du schéma de couleurs : testez le thème clair/sombre de l'application.

Commande pour lancer Codegen avec émulation :

```
npx playwright codegen --device="iPhone 12" https://zenity.fr
```

Cela lance Codegen avec la configuration d'émission d'un iPhone 12.



Vous pouvez également personnaliser d'autres aspects comme la langue ou le fuseau horaire avec des options supplémentaires.

BONNES PRATIQUES AVEC CODEGEN

- Optimiser les locators générés : bien que Codegen génère automatiquement des locators robustes, il est recommandé de les ajuster manuellement pour s'assurer qu'ils sont suffisamment stables. Par exemple, privilégiez les attributs comme `data-testid` ou `aria-label`.
- Utiliser les assertions à bon escient : évitez de trop utiliser d'assertions, car cela peut rendre les tests plus fragiles. Concentrez-vous sur les points critiques de validation.

- Générer des tests réutilisables : si vous utilisez souvent les mêmes séquences d'actions, comme un processus de connexion, considérez la création de fonctions réutilisables dans vos tests.

CONCLUSION

Playwright Codegen est un outil idéal pour accélérer le processus de création de tests automatisés, en particulier pour les développeurs qui débutent avec Playwright. Il permet non seulement de capturer les interactions de base avec l'application, mais aussi de générer des assertions robustes et des locators stables. En utilisant les fonctionnalités d'émulation et en suivant les bonnes pratiques, vous pourrez créer des tests performants et facilement maintenables, vous assurant ainsi une couverture de test efficace et fiable.

A TOI DE JOUER !

Répondez donc aux questions 15 à 21 du QCM de fin de document et exécutez le TPO 3.

5. Scénario avancés

OBJECTIF

Dans ce chapitre, nous allons explorer des scénarios de test plus complexes avec Playwright, notamment la manipulation des frames et iframes, la gestion des fenêtres multiples et des contextes de navigation, la gestion des dialogues, alertes et pop-ups, ainsi que les techniques pour gérer les temps d'attente et les timeouts.

MANIPULATION DES FRAMES ET DES IFRAMES

Les frames et iframes sont des éléments HTML permettant d'imbriquer une page web dans une autre. Tester des applications avec des frames peut poser des défis supplémentaires, mais Playwright offre une gestion intuitive des frames pour vous permettre d'interagir avec les éléments imbriqués.

Accéder à une frame :

Pour interagir avec une frame, vous devez d'abord accéder à l'élément `frame` ou `iframe` :

```
const frame = page.frame({ name: 'frame-name' }); // Sélection par le nom de la frame
```

Vous pouvez aussi accéder à une frame en utilisant un sélecteur :

```
const frame = page.frameLocator('iframe').first(); // Sélection par un sélecteur d'iframe
```

Interagir avec des éléments dans une frame :

Une fois que vous avez sélectionné la frame, vous pouvez interagir avec les éléments à l'intérieur de celle-ci comme vous le feriez avec une page normale :

```
await frame.click('button#submit');
await frame.fill('input[name="email"]', 'test@example.com');
```

Attendre qu'une frame soit chargée :

Si la frame prend du temps à se charger, vous pouvez utiliser une attente pour vous assurer que la frame est bien disponible :

```
await frame.click('button#submit');
await frame.fill('input[name="email"]', 'test@example.com');
```

GESTION DES FENETRES MULTIPLES ET DES CONTEXTES DE NAVIGATION

Dans certaines applications, il est courant qu'une action ouvre une nouvelle fenêtre ou un nouvel onglet. Playwright permet de manipuler ces fenêtres multiples avec aisance.

Ouvrir et gérer une nouvelle fenêtre :

Pour interagir avec une nouvelle fenêtre ouverte par un lien ou un bouton, vous devez capturer l'événement :

```
const [newPage] = await Promise.all([
  page.waitForEvent('popup'), // Attendre qu'une nouvelle fenêtre s'ouvre
  page.click('a[target="_blank"]') // Cliquer sur un lien qui ouvre une nouvelle fenêtre
]);
await newPage.goto('https://example.com');
```

Gérer les contextes de navigation :

Les contextes de navigateur sont utilisés pour isoler les sessions de navigation. Chaque contexte de navigateur fonctionne comme un navigateur indépendant avec ses propres cookies, sessions et cache.

```
const context = await browser.newContext(); // Crée un nouveau contexte de navigation
const newPage = await context.newPage(); // Ouvre une nouvelle page dans ce contexte
await newPage.goto('https://example.com');
```

Fermer les fenêtres :

Vous pouvez également fermer les fenêtres ouvertes :

```
await newPage.close();
```

GESTION DES DIALOGUES, ALERTES ET POP-UPS

Les applications web peuvent générer des dialogues tels que des alertes, des confirmations ou des invites (prompts). Playwright offre des méthodes spécifiques pour intercepter et manipuler ces dialogues.

Intercepter et gérer une alerte :

Pour gérer une alerte, vous devez écouter l'événement dialog :

```
page.on('dialog', async dialog => {
  console.log(dialog.message()); // Afficher le message de l'alerte
  await dialog.accept(); // Accepter l'alerte
});
await page.click('button#trigger-alert'); // Déclencher l'alerte
```

Gérer une confirmation :

Pour gérer une confirmation, vous pouvez choisir de l'accepter ou de la refuser :

```
page.on('dialog', async dialog => {
  await dialog.accept(); // Accepter la confirmation
  // await dialog.dismiss(); // Rejeter la confirmation
});
await page.click('button#trigger-confirm');
```

Gérer un prompt (invite de saisie) :

Si un prompt demande une entrée utilisateur, vous pouvez fournir une réponse :

```
page.on('dialog', async dialog => {
  await dialog.accept('Réponse saisie'); // Fournir une réponse au prompt
});
await page.click('button#trigger-prompt');
```

GESTION DES TEMPS D'ATTENTE

Les temps d'attente et les conditions d'attente sont essentiels dans les tests automatisés, car certaines actions (comme le chargement de page ou l'apparition d'un élément) peuvent prendre du temps.

Temps d'attente par défaut :

Par défaut, Playwright attend que les actions soient terminées, mais vous pouvez ajuster ces durées globalement ou pour des actions spécifiques :

```
const longTimeout = 30000;
await page.locator('button#submit').click({ timeout: longTimeout }); // Délai global de 30 secondes pour les actions
```

Attendre qu'un élément soit visible ou interactif :

Pour s'assurer qu'un élément est prêt avant d'interagir avec, utilisez les méthodes d'attente conditionnelle :

```
await page.waitForSelector('button#submit', { state: 'visible' }); // Attendre que le bouton soit visible
await page.click('button#submit');
```

Configurer des timeouts spécifiques pour des actions :

Vous pouvez définir des timeouts spécifiques pour chaque action :

```
await page.click('button#submit', { timeout: 5000 }); // Timeout de 5 secondes pour l'action
```

Attendre des événements spécifiques :

Playwright permet d'attendre des événements précis avant de poursuivre l'exécution du test :

```
await page.waitForEvent('load'); // Attendre que la page soit complètement chargée
```

Ces fonctionnalités avancées permettent de gérer des scénarios de test complexes avec Playwright, incluant des éléments imbriqués, plusieurs fenêtres, des dialogues natifs du navigateur, et la gestion fine des temps d'attente pour garantir la fiabilité des tests. Le prochain chapitre explorera l'automatisation des tests parallèles et l'intégration continue avec Playwright.

A TOI DE JOUER !

Répondez aux questions 22 à 28 du QCM de fin de document et exécutez le TP04.

6. API & Transactions

OBJECTIF

Dans ce chapitre, nous allons aborder l'interaction avec les APIs et les réponses réseau dans Playwright. Vous découvrirez comment intercepter les requêtes et les réponses HTTP, simuler des réponses réseau, et effectuer des tests avec des APIs mockées. Ces fonctionnalités vous permettent de tester des scénarios où les requêtes réseau sont cruciales pour le bon fonctionnement de votre application.

INTERCEPTION DES REQUETES ET DES REPONSES HTTP

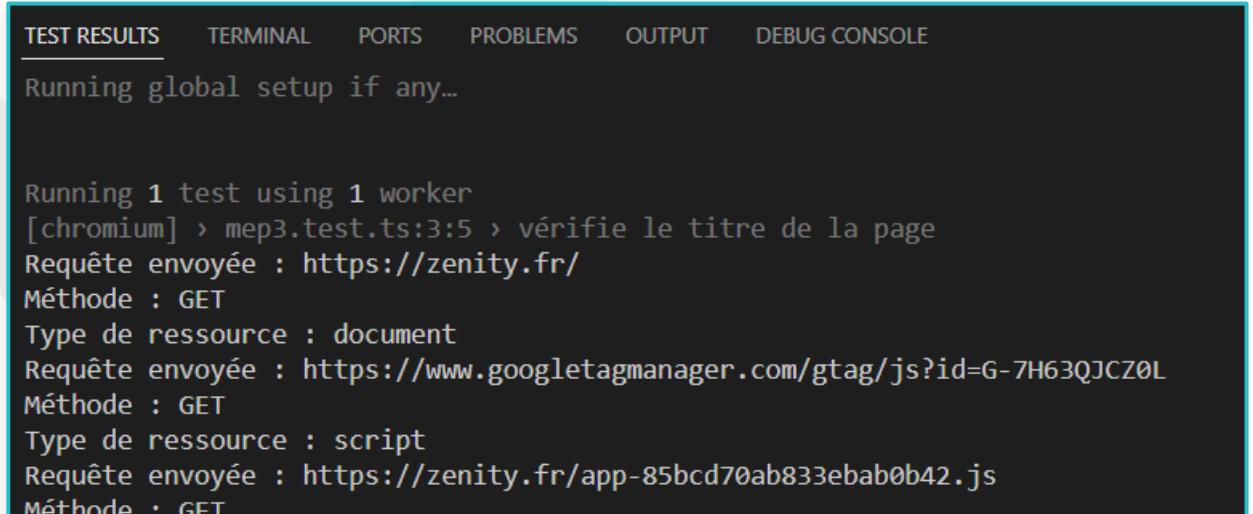
L'une des fonctionnalités puissantes de Playwright est la capacité d'intercepter les requêtes HTTP faites par une page et de contrôler leur comportement. Cela vous permet d'analyser ou de modifier les requêtes et les réponses en cours de route.

Intercepter les requêtes HTTP :

Vous pouvez écouter les requêtes sortantes d'une page et inspecter leurs détails :

```
page.on('request', request => {
  console.log(`Requête envoyée : ${request.url()}`);
  console.log(`Méthode : ${request.method()}`);
  console.log(`Type de ressource : ${request.resourceType()}`);
});
await page.goto('https://zenity.fr'); - 975ms
```

Cette méthode permet de voir toutes les requêtes réseau envoyées par la page au moment de la navigation vers zenity.fr, ce qui est utile pour comprendre le comportement de l'application ou du site web.



The screenshot shows a browser developer tools interface with tabs for TEST RESULTS, TERMINAL, PORTS, PROBLEMS, OUTPUT, and DEBUG CONSOLE. The TERMINAL tab is active, displaying the following log output:

```
TEST RESULTS TERMINAL PORTS PROBLEMS OUTPUT DEBUG CONSOLE
Running global setup if any...

Running 1 test using 1 worker
[chromium] > mep3.test.ts:3:5 > vérifie le titre de la page
Requête envoyée : https://zenity.fr/
Méthode : GET
Type de ressource : document
Requête envoyée : https://www.googletagmanager.com/gtag/js?id=G-7H63QJCZ0L
Méthode : GET
Type de ressource : script
Requête envoyée : https://zenity.fr/app-85bcd70ab833ebab0b42.js
Méthode : GET
```

Intercepter les réponses HTTP :

Vous pouvez également écouter les réponses reçues et inspecter leur contenu :

```
page.on('response', response => {
  console.log(`Réponse reçue : ${response.url()}`);
  console.log(`statut : ${response.status()}`);
});
await page.goto('https://zenity.fr');
```

Cela permet d'observer les réponses des requêtes et de vérifier leur statut ou contenu.

```
Running 1 test using 1 worker
[chromium] > mep3.test.ts:3:5 > vérifie le titre de la page
Réponse reçue : https://zenity.fr/
Statut : 200
Réponse reçue : https://zenity.fr/static/FuturaStdMedium-360d49b0669048a4aadf919dc29ddda0.otf
Statut : 200
Réponse reçue : https://zenity.fr/static/FuturaStdHeavy-4e48a3c9e2b89bb046111fbf2d216595.otf
Statut : 200
Réponse reçue : https://zenity.fr/static/FuturaStdLight-15a7fc5cccd79b959bd5a76820041424.otf
```

Intercepter des requêtes spécifiques :

Si vous souhaitez intercepter une requête particulière (par exemple, un appel API spécifique), vous pouvez utiliser une condition sur l'URL :

```
page.on('request', request => {
  if (request.url().includes('googletagmanager')) {
    console.log(`Requête robot Google : ${request.url()}`);
  }
});

await page.goto('https://zenity.fr'); - 546ms
```

[TEST RESULTS](#) [TERMINAL](#) [PORTS](#) [PROBLEMS](#) [OUTPUT](#) [DEBUG CONSOLE](#)

```
Running 1 test using 1 worker
[chromium] > mep3.test.ts:3:5 > vérifie le titre de la page
Requête robot Google : https://www.googletagmanager.com/gtag/js?id=G-7H63QJCZ0L
1 passed (2.0s)
```

SIMULATION DE REPONSES RESEAU

Playwright permet également de simuler des réponses réseau afin de tester comment votre application réagit à différents scénarios (comme une requête échouée, une réponse lente, ou des données spécifiques).

Bloquer ou modifier des requêtes :

Vous pouvez intercepter une requête avant qu'elle ne soit envoyée et la bloquer ou la modifier :

```

await page.route('https://zenity.fr**', route => {
  route.abort(); // Bloque la requête - 2ms
});

await page.goto('https://zenity.fr'); - 23ms

```

Dans cet exemple, toutes les requêtes vers zenity.fr sont bloquées, simulant un serveur indisponible.

Simuler une réponse réseau personnalisée :

Vous pouvez également répondre à une requête avec des données fictives (mockées) :

```

await page.route('**/api/v1/data', route => {
  route.fulfill({
    status: 200,
    contentType: 'application/json',
    body: JSON.stringify({ message: 'Données simulées' })
  });
};

await page.goto('https://zenity.fr');

```

Ici, chaque requête vers l'API retourne une réponse personnalisée, permettant de tester comment l'application gère les données simulées.

Simuler un délai de réponse (latence) :

Il est parfois nécessaire de tester comment l'application réagit à des délais réseau. Vous pouvez simuler une latence pour une requête spécifique :

```

await page.route('https://zenity.fr', async route => {
  await new Promise(resolve => setTimeout(resolve, 10000)); // Délai de 10 secondes
  route.continue();
});

await page.goto('https://zenity.fr');

```

Cela permet de vérifier si votre application gère correctement les temps d'attente prolongés ou les timeout.

TESTS AVEC API MOCKEES

Les tests avec des APIs mockées permettent de simuler des interactions avec des serveurs backend sans avoir besoin d'une véritable connexion réseau. Cela est particulièrement utile dans des environnements de tests unitaires ou d'intégration où l'on souhaite isoler les tests du réseau.

Mocker une API entière :

Vous pouvez simuler des réponses API pour toute une série de requêtes, ce qui permet de tester des scénarios précis :

```
await page.route('**/api/v1/users', route => {
  route.fulfill({
    status: 200,
    contentType: 'application/json',
    body: JSON.stringify([
      { id: 1, name: 'Karim' },
      { id: 2, name: 'Bob' },
    ]),
  });
});

await page.goto('https://zenity.fr');
```

Dans cet exemple, l'API des utilisateurs retourne toujours la même liste d'utilisateurs fictifs.

Tester des scénarios d'échec :

Simuler une erreur côté serveur est une technique utile pour tester la robustesse de votre application :

```
await page.route('https://zenity.fr', route => {
  route.fulfill({
    status: 500,
    contentType: 'application/json',
    body: JSON.stringify({ error: 'Erreur serveur' }),
  });
});

await page.goto('https://zenity.fr');
await page.screenshot({path: 'screenshot.png'});
```

Cet exemple force une erreur 500, permettant de tester comment l'application réagit à un échec d'API.

```
Pretty-print □  
{"error": "Erreur serveur"}
```

Valider les requêtes envoyées :

Vous pouvez également vérifier que les requêtes envoyées par votre application sont conformes à vos attentes :

```
page.on('request', request => {  
  if (request.url().includes('googletagmanager') && request.method() === 'GET') {  
    console.log('Requête GET vers Google Tag détectée');  
  }  
});  
  
await page.goto('https://zenity.fr');
```

Cette vérification permet de s'assurer que votre application envoie bien les bonnes requêtes au bon moment.

CONCLUSION

En maîtrisant l'interception des requêtes et réponses HTTP, la simulation de scénarios réseau, et les tests avec des APIs mockées, vous pouvez considérablement enrichir vos tests automatisés avec Playwright. Ces techniques vous permettent de contrôler le comportement de l'application dans des scénarios complexes et de valider sa robustesse face à des événements réseau imprévus. Le prochain chapitre explorera l'intégration de Playwright dans des pipelines de CI/CD et les bonnes pratiques pour l'automatisation des tests à grande échelle.



A TOI DE JOUER !

Répondez aux questions 29 à 35 du QCM de fin de document et exécutez le TP05.

7. Gestion des fichiers et des téléchargements

OBJECTIF

Dans ce chapitre, nous explorerons comment Playwright peut être utilisé pour automatiser la gestion des fichiers dans vos tests. Cela inclut le téléchargement et l'upload de fichiers, ainsi que la manipulation des fichiers téléchargés. Ces fonctionnalités permettent de tester des scénarios où les interactions avec des fichiers sont essentielles, comme le téléchargement de rapports ou l'upload de documents.

AUTOMATISATION DU TELECHARGEMENT DE FICHIERS

Les tests impliquant le téléchargement de fichiers peuvent être cruciaux pour vérifier si les fonctionnalités associées fonctionnent correctement (comme les téléchargements de rapports ou de pièces jointes).

Activer le téléchargement de fichiers

Avant de pouvoir automatiser le téléchargement, vous devez définir un répertoire où les fichiers seront enregistrés. Playwright vous permet de spécifier un chemin de téléchargement pour chaque contexte de navigation.

```

const context = await browser.newContext({
  acceptDownloads: true,
});
const page = await context.newPage();
await page.goto('https://fonts.google.com/specimen/Roboto');
page.click('//span[text()=" Get font "]');

// Lancer le téléchargement d'un fichier
const [download] = await Promise.all([
  page.waitForEvent('download'),
  page.click('//span[text()=" Download all (1) "]'), // cliquez sur un lien de téléchargement
]);

// Enregistrer le fichier téléchargé dans un chemin spécifique
await download.saveAs('downloads' + `/${download.suggestedFilename()}`);
console.log(`Fichier téléchargé : ${download.suggestedFilename()}`);

```

Ici, nous écoutons l'événement `download`, puis nous enregistrons le fichier téléchargé dans un dossier spécifique.

[TEST RESULTS](#) [TERMINAL](#) [PORTS](#) [GITLENS](#) [PROBLEMS](#) [OUTPUT](#) [DEBUG CONSOLE](#)

```

Running 1 test using 1 worker
[chromium] > mep5.test.ts:3:5 > vérifie le titre de la page
Fichier téléchargé : Roboto.zip
  1 passed (5.0s)

```

Attendre la fin d'un téléchargement

Pour vous assurer que le fichier est complètement téléchargé avant de passer à l'étape suivante du test, vous pouvez utiliser la méthode `download.path()` :

```

const filePath = await download.path();
console.log(`Chemin du fichier téléchargé : ${filePath}`);

```

Cette méthode permet de s'assurer que le fichier est complètement téléchargé avant de le manipuler.

AUTOMATISATION DE L'UPLOAD DE FICHIERS

Le téléchargement de fichiers est une autre fonctionnalité fréquemment testée dans les applications web, en particulier lorsqu'il s'agit d'upload de documents, d'images, ou de fichiers CSV.

Uploader un fichier à l'aide d'un input HTML

Playwright permet d'uploader des fichiers en simulant l'interaction avec les éléments de formulaire HTML. Pour ce faire, vous utilisez l'élément `input[type="file"]` :

```
const context = await browser.newContext({ - 16ms
| acceptDownloads: true,
});
const page = await context.newPage(); - 152ms
await page.goto('https://www.file.io'); - 767ms

const filePath = 'downloads/Roboto.zip';
await page.setInputFiles('//label[@for="upload-button"]', filePath); - 63ms

await page.waitForTimeout(10000); - 10007ms
```

Cette méthode permet de charger le fichier spécifié dans l'élément de formulaire, puis de soumettre le formulaire.

Uploader plusieurs fichiers

Il est possible de simuler l'upload de plusieurs fichiers à la fois en fournissant un tableau de chemins de fichiers :

```
const context = await browser.newContext({ - 13ms
| acceptDownloads: true,
});
const page = await context.newPage(); - 137ms
await page.goto('https://www.file.io'); - 830ms

const filePaths = ['downloads/Roboto.zip', 'downloads/Roboto copy.zip'];
await page.setInputFiles('//label[@for="upload-button"]', filePaths); - 53ms

await page.waitForTimeout(10000); - 10002ms
```

Cela permet de tester les scénarios d'upload multiple, utile pour des fonctionnalités comme l'envoi de pièces jointes.

MANIPULATION DES FICHIERS TELECHARGES

Une fois le fichier téléchargé, il peut être utile de vérifier son contenu ou de le manipuler pour tester davantage les fonctionnalités de l'application.

Vérifier le contenu des fichiers téléchargés

Playwright ne permet pas directement de lire le contenu des fichiers, mais vous pouvez utiliser des bibliothèques Node.js comme `fs` (file system) pour analyser ou vérifier le contenu des fichiers téléchargés :

```
await page.goto('https://commons.library.stonybrook.edu/egp/1/'); - 2095ms

// Lancer le téléchargement d'un fichier
const [download] = await Promise.all([
  page.waitForEvent('download'), - 5430ms
  page.click('//a[@title="Download Copyright & license"]'), // Cliquez sur un lien de téléchargement - 5428ms
]);

const fs = require('fs');
const filePath = await download.path(); - 466ms

// tire le contenu du fichier téléchargé
fs.readFile(filePath, 'utf8', (err, data) => {
  if (err) throw err;
  console.log(`Contenu du fichier : ${data}`);
});
```

Cela peut être utile pour tester si le fichier téléchargé contient les bonnes données (par exemple, un rapport généré dynamiquement).

CONCLUSION

La gestion des fichiers dans les tests automatisés est une compétence importante lorsque vous travaillez avec des applications nécessitant l'upload et le téléchargement de fichiers. Playwright offre des solutions simples pour automatiser ces interactions et permet également de manipuler et vérifier les fichiers une fois qu'ils sont téléchargés. Grâce à ces outils, vous pouvez tester de bout en bout les processus impliquant des fichiers, garantissant ainsi la fiabilité de votre application dans des scénarios concrets.

A TOI DE JOUER !

Répondez aux questions 43 à 49 du QCM de fin de document et exécutez le TP05.

8. Exécution en parallèle & dans le cloud

OBJECTIF

Dans ce chapitre, nous allons explorer comment maximiser l'efficacité de vos tests en les exécutant en parallèle et sur des plateformes cloud. Nous examinerons la configuration des tests parallèles, l'intégration avec des services cloud comme [BrowserStack](#) et [Sauce Labs](#), ainsi que l'exécution des tests sur différents navigateurs tels que Chromium, WebKit, et Firefox.

CONFIGURATION DES TESTS PARALLELES

L'exécution des tests en parallèle est une technique puissante qui permet de réduire considérablement le temps total d'exécution de votre suite de tests. Playwright prend en charge cette fonctionnalité nativement.

Activer l'exécution parallèle avec Playwright

Par défaut, Playwright exécute les tests en parallèle, sauf indication contraire. Vous pouvez ajuster le nombre de workers (processus parallèles) en modifiant le fichier de configuration `playwright.config.ts`.

Exemple de configuration :

```
// Nombre de workers (tests parallèles)
workers: 4, // Par défaut, Playwright choisit le nombre optimal basé sur les cœurs CPU
use: {
  headless: true, // Exécution des tests sans interface graphique
},
```

Ici, nous spécifions 4 workers pour permettre à Playwright d'exécuter 4 tests en parallèle. Vous pouvez ajuster ce nombre en fonction de la taille de votre projet et des ressources disponibles.

Isoler les tests pour l'exécution parallèle

Lorsque vous exécutez des tests en parallèle, il est important que chaque test soit isolé et indépendant des autres. Cela signifie éviter les dépendances globales et s'assurer que chaque test crée et nettoie son propre contexte de navigation :

```
test('Mon test en parallèle #1', async ({ page }) => {
  await page.goto('https://zenity.fr/identite');
  // Le test s'exécute dans son propre contexte de page isolé
});

test('Mon test en parallèle #2', async ({ page }) => {
  await page.goto('https://zenity.fr/nos-univers');
  // Le test s'exécute dans son propre contexte de page isolé
});
```

Chaque test Playwright s'exécute dans une nouvelle instance de navigateur pour garantir l'isolation.

TESTS SUR DIFFERENTS NAVIGATEURS

Retournez sur le projet initial '*Projects\formation-playwright*'

Playwright prend en charge l'exécution des tests sur plusieurs moteurs de rendu de navigateurs : Chromium, WebKit, et Firefox. Cela permet de tester la compatibilité des applications web sur divers navigateurs sans configuration supplémentaire.

Exécuter les tests sur Chromium, WebKit, et Firefox

Par défaut, Playwright supporte ces trois navigateurs majeurs. Vous pouvez spécifier le navigateur que vous souhaitez tester dans la configuration ou directement dans vos tests.

Exemple de configuration multi-navigateurs :

```
/* Configure projects for major browsers */
projects: [
  {
    name: 'chromium',
    use: { browserName: 'chromium' },
  },
  {
    name: 'firefox',
    use: { browserName: 'firefox' },
  },
  {
    name: 'webkit',
    use: { browserName: 'webkit' },
  },
]
```

Dans cet exemple, chaque test sera exécuté sur Chromium, Firefox, et WebKit. Cela garantit que votre application est compatible avec chacun de ces navigateurs.

Lancer des tests pour des environnements spécifiques

Vous pouvez également cibler un navigateur particulier pour certains tests en utilisant les annotations de projet dans Playwright :

```
test.use({ browserName: 'firefox' });

test('Test sur Firefox uniquement', async ({ page }) => {
  await page.goto('https://zenity.fr'); - 1088ms
  // Ce test sera exécuté uniquement sur Firefox
});
```

Cela vous permet de contrôler précisément les environnements de test pour chaque cas d'usage.

Tests multiplateformes (mobile, desktop)

Playwright propose également des émulations pour des appareils mobiles et tablettes. Vous pouvez tester votre application sur des appareils mobiles en ajoutant des configurations spécifiques :

```
/* Configure projects for major browsers */
projects: [
  {
    name: 'iPhone 12',
    use: { ...devices['iPhone 12'] },
  },
  {
    name: 'Pixel 5',
    use: { ...devices['Pixel 5'] },
  },
],
```

Cela permet d'exécuter vos tests sur des résolutions et des configurations d'appareils spécifiques, ce qui est utile pour valider le comportement responsive de votre application.

Re**définissez les browsers d'exécution pour la suite du cours** :

```
/* Configure projects for major browsers */
projects: [
  {
    name: 'chromium',
    use: { ...devices['Desktop Chrome'] },
  },
],
```

INTEGRATION AVEC DES PLATEFORMES D'EXECUTION DE TESTS DANS LE CLOUD

Les plateformes de test cloud comme BrowserStack, Sauce Labs, et LambdaTest permettent d'exécuter vos tests sur différents environnements (navigateurs et systèmes d'exploitation) sans avoir à les configurer localement. Cela permet de tester sur un large éventail de configurations et de versions de navigateurs.



Mise en pratique guidée

BrowserStack est une plateforme populaire pour exécuter des tests d'automatisation dans le cloud. Playwright propose une intégration native avec BrowserStack. Voici comment configurer votre projet pour exécuter les tests sur cette plateforme :

Prérequis

- Nom d'utilisateur et clé d'accès de BrowserStack. Vous pouvez les trouver dans votre [profil de compte](#). Si vous n'avez pas encore créé de compte, vous pouvez [vous inscrire pour un essai gratuit](#).
- Node v14 est installé sur votre machine.

Exécutez un exemple de Tests

1. Cloner l'exemple de dépôt

Retournez à la racine de dossier « projects » et clonez l'exemple de dépôt Git à l'aide des commandes suivantes :

```
PS D:\ZENITY\Projects> git clone https://github.com/browserstack/typescript-playwright-browserstack
Cloning into 'typescript-playwright-browserstack'...
remote: Enumerating objects: 29, done.
remote: Counting objects: 100% (29/29), done.
remote: Compressing objects: 100% (24/24), done.
remote: Total 29 (delta 4), reused 15 (delta 0), pack-reused 0 (from 0)
Receiving objects: 100% (29/29), 12.85 KiB | 6.42 MiB/s, done.
Resolving deltas: 100% (4/4), done.
PS D:\ZENITY\Projects> cd ..
PS D:\ZENITY> cd .\Projects\typescript-playwright-browserstack\
PS D:\ZENITY\Projects\typescript-playwright-browserstack> █
```

2. Configurer les dépendances

Installez les dépendances requises en exécutant la commande suivante :

```
PS D:\ZENITY\Projects\typescript-playwright-browserstack> npm i
added 32 packages, and audited 33 packages in 720ms

1 package is looking for funding
  run `npm fund` for details

found 0 vulnerabilities
PS D:\ZENITY\Projects\typescript-playwright-browserstack> █
```

3. Configurez votre fichier de configuration browserstack.config.ts

Le fichier `browserstack.config.ts` contient toutes les fonctionnalités requises pour exécuter vos tests sur BrowserStack

- Définir les informations d'identification d'accès

Définissez les propriétés `BROWSERSTACK_USERNAME` et `BROWSERSTACK_ACCESS_KEY` dans `browserstack.config.ts` pour authentifier vos tests sur BrowserStack. Ces valeurs sont disponibles sur [votre profil](#).

- Spécifier les plateformes sur lesquelles tester

Définissez les navigateurs/systèmes d'exploitation que vous souhaitez tester sous l'objet plateformes. Sélectionnez plus de 100 combinaisons navigateurs-OS dans [la liste des navigateurs et systèmes d'exploitation](#) pris en charge.

- Mettre à jour le fichier de configuration `browserstack.config.ts`

Copiez et remplacez la configuration suivante dans le fichier `browserstack.config.ts` disponible dans le répertoire racine du projet.

```
// Browserstack specific capabilities.
// Set 'browserstack.local=true' for local testing
const caps = {
  browser: 'chrome',
  os: 'osx',
  os_version: 'catalina',
  name: 'My first playwright test',
  build: 'playwright-build',
  'browserstack.username': process.env.BROWSERSTACK_USERNAME || 'karimhafsi_pfe7ch',
  'browserstack.accessKey': process.env.BROWSERSTACK_ACCESS_KEY || 'wPPT52RiyG5xhrZATYvu',
  'browserstack.local': process.env.BROWSERSTACK_LOCAL || true,
  'client.playwrightVersion': clientPlaywrightVersion,
};

exports.bsLocal = new BrowserStackLocal.local();

// replace YOUR_ACCESS_KEY with your key. You can also set an environment variable - "BROWSERSTACK_ACCESS_KEY".
exports.bs_LOCAL_ARGS = {
  key: process.env.BROWSERSTACK_ACCESS_KEY || 'wPPT52RiyG5xhrZATYvu',
};
```

4. Exécuter la compilation sur BrowserStack

Vous êtes maintenant prêt à exécuter votre build sur BrowserStack. Depuis le répertoire racine du projet, exécutez la commande suivante.

```
PS D:\ZENITY\Projects\typescript-playwright-browserstack> npm run sample-test
> v1.28_playwright_test@1.0.0 sample-test
> npx playwright test tests/sample_test.ts

--> wss://cdp.browserstack.com/playwright?caps=%7B%22browser%22%3A%22chrome%22%2C%22os%22%3A%22windows%22%2C%
tack.username%22%3A%22karimhafsi_pfe7ch%22%2C%22browserstack.accessKey%22%3A%22whPT52RiyG5xhrZAtvuu%22%2C%22b
--> wss://cdp.browserstack.com/playwright?caps=%7B%22browser%22%3A%22playwright-webkit%22%2C%22os%22%3A%22OSX
C%22browserstack.username%22%3A%22karimhafsi_pfe7ch%22%2C%22browserstack.accessKey%22%3A%22whPT52RiyG5xhrZAtY
--> wss://cdp.browserstack.com/playwright?caps=%7B%22browser%22%3A%22playwright-firefox%22%2C%22os%22%3A%22Wi
C%22browserstack.username%22%3A%22karimhafsi_pfe7ch%22%2C%22browserstack.accessKey%22%3A%22whPT52RiyG5xhrZAtY
Starting BrowserStackLocal ...
BrowserStackLocal Started

Running 3 tests using 3 workers
```

5. Afficher les résultats des tests

Affichez vos tests sur le [tableau de bord BrowserStack Automate](#). Pour en savoir plus sur le tableau de bord, consultez le document [Afficher les résultats des tests](#).

6. Prochaines étapes

Après avoir exécuté avec succès votre premier test sur BrowserStack, essayez [d'intégrer votre suite de tests à BrowserStack](#).

CONCLUSION

L'exécution des tests en parallèle et sur des plateformes cloud permet d'accélérer considérablement le processus de test tout en offrant une couverture étendue sur différents environnements. Playwright rend cette tâche simple en offrant un support natif pour la parallélisation, l'exécution multi-navigateurs, et l'intégration avec des plateformes cloud comme **BrowserStack** et **Sauce Labs**. Grâce à ces outils, vous pouvez améliorer la qualité de vos applications web tout en réduisant le temps nécessaire à la validation de leur compatibilité sur des configurations variées.

A TOI DE JOUER !

Répondez aux questions 50 à 56 du QCM de fin de document.

Restez concentré ! Les notions de ce chapitre seront exercées à l'issue du prochain chapitre.

9. Rapports & logs

OBJECTIF

L'un des aspects essentiels de l'automatisation des tests est la génération de rapports clairs et exploitables, ainsi que l'analyse des logs pour diagnostiquer et résoudre les problèmes. Dans ce chapitre, nous allons explorer comment Playwright permet de créer des rapports de tests, d'analyser les logs, de capturer des captures d'écran et de gérer les enregistrements vidéo des sessions de test.

GENERATION DE RAPPORTS DE TESTS (HTML, JSON)

Les rapports de tests sont cruciaux pour avoir une vue d'ensemble de l'état des tests : réussite, échec, et informations contextuelles. Playwright fournit plusieurs options pour la génération de rapports, telles que les rapports au format **HTML** et **JSON**.

Génération d'un rapport HTML avec Playwright Test Reporter

Playwright inclut un reporter HTML qui génère un rapport interactif récapitulant les tests exécutés, leurs statuts (succès, échec, timeout), ainsi que les logs associés. Voici comment configurer le reporter HTML dans votre projet :

Modifiez le fichier `playwright.config.ts` :

```
playwright.config.ts > [o] default
1 import { defineConfig, devices } from '@playwright/test';
2
3 export default defineConfig({
4
5   reporter: [
6     ['html', { outputFolder: 'playwright-report', open: 'never' }],
7   ],
8
9   use: {
10     trace: 'on-first-retry', // Enregistre des traces en cas d'échec
11   },
12})
```

Lancez vos tests avec la commande suivante :

```
PS D:\ZENITY\Projects\formation-playwright> npx playwright test
```

Après l'exécution, un rapport HTML sera généré dans le dossier `playwright-report`.

Vous pouvez ouvrir ce rapport en utilisant :

```
PS D:\ZENITY\Projects\formation-playwright> npx playwright show-report
Serving HTML report at http://localhost:9323. Press Ctrl+C to quit.
```

Le rapport HTML inclut une vue des tests, des logs, des captures d'écran et des vidéos enregistrées.

GENERATION DE RAPPORTS JSON

Le format JSON est utile pour l'intégration avec d'autres outils ou pour analyser les résultats de tests de manière programmatique. Pour générer un rapport JSON, vous pouvez utiliser le reporter JSON de Playwright :

Modifiez le fichier `playwright.config.ts` :

```
playwright.config.ts > [o] default
1 import { defineConfig, devices } from '@playwright/test';
2
3 export default defineConfig({
4
5   reporter: [['json', { outputFile: 'rapport-tests.json' }]],
6})
```

Cela générera un fichier `rapport-tests.json` avec les résultats des tests, que vous pourrez utiliser pour effectuer des analyses automatisées ou des comparaisons de résultats.

COMBINER PLUSIEURS REPORTERS

Playwright permet d'utiliser plusieurs reporters simultanément. Par exemple, vous pouvez générer un rapport HTML tout en produisant un fichier JSON pour une analyse ultérieure :

```
playwright.config.ts > [?] default
1 import { defineConfig, devices } from '@playwright/test';
2
3 export default defineConfig({
4
5   reporter: [
6     ['html', { outputFolder: 'playwright-report' }],
7     ['json', { outputFile: 'rapport-tests.json' }],
8   ],
9 })
```

Cette configuration est idéale pour avoir des rapports visuels et en même temps des données structurées destinée par exemple à une intégration avec d'autres logiciels (Jira).

ANALYSE DES LOGS ET DES CAPTURES D'ECRAN

Les logs et les captures d'écran sont des outils puissants pour déboguer et analyser les problèmes rencontrés pendant les tests.

Capture automatique des captures d'écran

Playwright permet de capturer des captures d'écran en cas d'échec d'un test ou à des étapes spécifiques de votre scénario de test :

```
test('Test avec capture d'écran', async ({ page }) => {
  await page.goto('https://zenity.fr');
  await page.waitForLoadState('networkidle');
  await page.screenshot({ path: 'captures/page-initiale.png' });
  await page.click('button#inexistant', {timeout:500}); // Ce clic échouera
})
```

Vous pouvez configurer Playwright pour capturer automatiquement les captures d'écran uniquement en cas d'échec du test dans `playwright.config.ts` :

```
use: {
  | screenshot: 'only-on-failure', // Capture d'écran en cas d'échec uniquement
},
```

Analyser les logs de la console

Vous pouvez accéder aux logs de la console du navigateur pour diagnostiquer les erreurs JavaScript ou les comportements inattendus :

```
test('Test avec capture d'écran', async ({ page }) => {
  page.on('console', (msg) => console.log(`Message console : ${msg.text()}`));
  await page.goto('https://zenity.fr');
```

En enregistrant ces logs, vous obtenez un aperçu des erreurs et avertissements générés par le navigateur.

Capture et comparaison de captures d'écran

Pour des tests de régression visuelle, Playwright permet de capturer et comparer des captures d'écran avec des versions de référence :

Déplacer la capture page-initiale.png dans un dossier du même nom que votre fichier de test avec le suffix « -snapshot » et dans le dossier de test, par exemple :

```
test('Test avec comparaison capture d'écran', async ({ page }) => {
  await page.goto('https://zenity.fr');
  await page.waitForLoadState('networkidle');

  // L'utilisation de base et le nom du fichier sont dérivés du nom du test.
  await expect.soft(page).toHaveScreenshot('page-initiale.png');

  // Transmettez les options pour personnaliser la comparaison d'instantanés et avoir un nom généré.
  await expect.soft(page).toHaveScreenshot('page-initiale.png', {
    maxDiffPixels: 27, // n'autorisez pas plus de 27 pixels différents.
  });

  // Configurez le seuil de correspondance d'image,
  await expect.soft(page).toHaveScreenshot('page-initiale.png', {
    threshold: 0.3
  });
});
```

Cette approche permet de s'assurer que l'interface utilisateur ne subit pas de changements indésirables entre les versions de l'application.

GESTION DES ENREGISTREMENTS VIDEO DES SESSIONS DE TEST

Les enregistrements vidéo des sessions de test offrent une vue dynamique du comportement de l'application pendant l'exécution des tests. Ils sont particulièrement utiles pour comprendre ce qui s'est passé lors de l'exécution des tests échoués.

Activer les enregistrements vidéo

Pour activer l'enregistrement vidéo des tests, vous pouvez ajuster la configuration suivante dans `playwright.config.ts` :

```
use: {  
  video: 'on', // Enregistre une vidéo pour chaque test
```

Cette configuration génère des vidéos pour chaque test dans le dossier spécifié. Vous pouvez également choisir d'enregistrer les vidéos uniquement pour les tests échoués :

```
use: {  
  video: 'retain-on-failure', // Enregistre uniquement les vidéos des tests échoués
```

Analyser les enregistrements vidéo

Une fois les vidéos générées, vous pouvez les visionner pour comprendre exactement ce qui s'est passé lors de l'exécution d'un test. Cela est particulièrement utile dans les cas où les logs et les captures d'écran ne donnent pas suffisamment d'informations.

Intégration des vidéos dans les rapports HTML

Playwright intègre automatiquement les enregistrements vidéo dans les rapports HTML, ce qui permet de visionner directement les vidéos depuis le rapport sans avoir à naviguer dans les fichiers. Cela rend l'analyse plus rapide et plus intuitive.

Supprimer les vidéos après analyse

Si vous souhaitez économiser de l'espace disque, vous pouvez configurer Playwright pour supprimer automatiquement les vidéos après analyse, en utilisant un script ou en ajoutant cette logique directement dans vos tests.

CONCLUSION

Les rapports de tests, les logs, les captures d'écran et les enregistrements vidéo sont des outils essentiels pour suivre l'exécution des tests et diagnostiquer les problèmes rencontrés. Playwright propose des fonctionnalités intégrées et des options de configuration pour gérer ces éléments efficacement. Grâce à ces outils, vous pouvez non seulement comprendre le comportement de vos tests, mais également communiquer les résultats de manière claire à votre équipe et à vos parties prenantes, facilitant ainsi la collaboration et la résolution rapide des problèmes.



A TOI DE JOUER !

Répondez aux questions 57 à 66 du QCM de fin de document et exécutez le TP06.

10. Tests d'accessibilité & performance

OBJECTIF

Dans ce chapitre, nous allons explorer l'utilisation de Playwright pour tester l'accessibilité des applications web et évaluer leurs performances.

Ces aspects sont essentiels pour garantir que votre application est utilisable par tous, tout en maintenant un niveau de performance optimal.

INTRODUCTION AUX TESTS D'ACCESSIBILITE AVEC PLAYWRIGHT

L'accessibilité est un aspect fondamental de toute application web moderne. Elle vise à rendre le contenu et les fonctionnalités accessibles à tous les utilisateurs, y compris ceux ayant des handicaps. Playwright propose des outils intégrés qui permettent de tester automatiquement certaines règles d'accessibilité.

Pourquoi l'accessibilité est importante ?

L'accessibilité garantit que les applications sont utilisables par tous, en respectant les standards internationaux comme les Web Content Accessibility Guidelines (WCAG). Elle inclut des aspects comme la navigation clavier, la lisibilité des contenus, et le bon usage des technologies assistives (lecteurs d'écran, etc.).

Tests d'accessibilité avec Playwright :

Playwright permet d'analyser des pages web à l'aide d'outils comme [AXE Core](#), une bibliothèque populaire pour auditer l'accessibilité.

Pour intégrer un audit d'accessibilité dans vos tests Playwright, vous pouvez utiliser l'intégration avec axe-core :

```
TEST RESULTS TERMINAL PORTS GITLENS PROBLEMS 2 OUTPUT DEBUG CONSOLE
PS D:\ZENITY\Projects\formation-playwright> npm install @axe-core/playwright
added 2 packages, and audited 9 packages in 979ms
found 0 vulnerabilities
PS D:\ZENITY\Projects\formation-playwright> 
```

Cet exemple utilise la bibliothèque axe-core pour analyser l'accessibilité de la page chargée et retourne un rapport avec les éventuelles violations.

```
await page.goto('https://zenity.fr'); - 860ms
const accessibilityScanResults = await new AxeBuilder({ page }).analyze();
expect(accessibilityScanResults.violations).toEqual([]); - 7ms
```

Vérification des rôles et des attributs d'accessibilité :

Playwright permet également de vérifier si les rôles [ARIA](#) (Accessible Rich Internet Applications) sont correctement utilisés sur les éléments interactifs de votre application :

```
await page.goto('https://zenity.fr'); - 607ms
const role = await page.getAttribute('button#submit', 'role'); - 29257ms
expect(role).toBe('button'); // Vérifie que l'attribut ARIA est correct
```

Génération de rapports d'accessibilité :

Vous pouvez utiliser des plugins ou des outils comme axe-core pour générer des rapports détaillés sur les problèmes d'accessibilité détectés sur vos pages. Cela permet d'identifier et de corriger des problèmes comme les contrastes de couleurs insuffisants ou l'absence d'alternatives textuelles pour les images.

TEST DES PERFORMANCES DES PAGES

La performance d'une application web est essentielle pour offrir une bonne expérience utilisateur, notamment sur des réseaux lents ou des appareils peu puissants. Playwright permet de mesurer différents aspects de la performance, comme le temps de chargement des pages ou les temps de réponse des interactions utilisateur.

Mesurer le temps de chargement d'une page :

Le temps de chargement d'une page est un indicateur clé de performance. Vous pouvez le mesurer avec Playwright en utilisant des événements comme `load` ou `domcontentloaded` :

```
const startTime = Date.now();
await page.goto('https://zenity.fr'); - 1030ms
await page.waitForLoadState('load'); - 1ms
const loadTime = Date.now() - startTime;
console.log(`Temps de chargement : ${loadTime}ms`);
```

Cet exemple mesure le temps écoulé entre le début de la navigation et le moment où la page est complètement chargée.

Évaluation des performances via l'API Performance Timing :

Playwright permet d'accéder à l'API Performance Timing du navigateur pour obtenir des mesures plus détaillées sur chaque étape du chargement d'une page (DNS, redirection, réponse du serveur, etc.) :

```
const timing = await page.evaluate(() => JSON.stringify(window.performance.timing)); - 8ms
console.log(JSON.parse(timing));
```

Cette méthode fournit des détails sur les différentes phases du chargement de la page, comme le temps de connexion réseau ou le délai avant que le DOM soit prêt.

TEST RESULTS TERMINAL PORTS GITLENS PROBLEMS OUTPUT DEBUG CONSOLE

```
Running 1 test using 1 worker
[chromium] > mep4.test.ts:5:5 > vérifie le titre de la page
{
  connectStart: 0,
  secureConnectionStart: 0,
  unloadEventEnd: 0,
  domainLookupStart: 0,
  domainLookupEnd: 0,
  responseStart: 0,
  connectEnd: 0,
  responseEnd: 1727098028436,
  requestStart: 0,
  domLoading: 1727098028431,
  redirectStart: 0,
  loadEventEnd: 1727098028437,
  domComplete: 1727098028437,
  navigationStart: 1727098028360,
  loadEventStart: 1727098028437,
  domContentLoadedEventEnd: 1727098028436,
  unloadEventStart: 0,
  redirectEnd: 0,
  domInteractive: 1727098028436,
  fetchStart: 0,
  domContentLoadedEventStart: 1727098028436
}
1 passed (1.6s)
```

Suivi du temps de réponse des interactions utilisateur :

Playwright peut aussi être utilisé pour mesurer le temps que prend une interaction utilisateur (comme un clic ou la soumission d'un formulaire) pour être traitée :

```
await page.goto('https://zenity.fr'); - 439ms
const startTime = Date.now();
await page.click('//a[@href="/contact"]'); - 95ms
await page.waitForLoadState(); - 884ms
const responseTime = Date.now() - startTime;
console.log(`Temps de réponse : ${responseTime}ms`);
```

Cela permet de s'assurer que les interactions critiques restent rapides même avec une application lourde.

CONCLUSION

Les tests d'accessibilité et de performance sont essentiels pour garantir une expérience utilisateur inclusive et fluide. Avec Playwright, vous pouvez automatiser ces tests pour identifier les problèmes potentiels avant même qu'ils n'affectent vos utilisateurs. Que ce soit en simulant des scénarios de réseau difficile, en mesurant les temps de réponse, ou en validant l'accessibilité selon les [normes WCAG](#), Playwright fournit les outils nécessaires pour assurer la qualité de vos applications web. Le prochain chapitre abordera l'automatisation des tests à grande échelle et l'intégration avec des pipelines de CI/CD.



Tu as avancé bien !

A TOI DE JOUER !

Répondez aux questions 36 à 42 du QCM de fin de document et exécutez le TP07.

11. Conseils & bonnes pratiques

OBJECTIF

L'automatisation des tests, bien qu'extrêmement puissante, peut devenir un véritable défi lorsqu'il s'agit de maintenir et d'optimiser la suite de tests à long terme. Ce chapitre couvre les meilleures pratiques pour structurer vos tests, améliorer leur résilience, et assurer une maintenance efficace face aux changements fréquents des applications.

ORGANISATION DES TESTS ET STRUCTURATION DU CODE

La structure de votre suite de tests est cruciale pour la lisibilité, la maintenabilité et la performance globale. Voici quelques conseils pour organiser vos tests :

Adopter une architecture modulaire

La création de modules distincts pour chaque composant, fonctionnalité, ou page de votre application est une approche idéale. Cela permet de séparer la logique d'automatisation des tests, facilitant ainsi leur réutilisation et leur modification. Pour cela, vous pouvez créer des **Page Objects** ou des **Helper Functions**.

Par exemple, pour une application e-commerce, vous pourriez avoir les modules suivants :

- pages/: contient les objets de pages (homePage.ts, loginPage.ts).
- tests/: contient les tests propres à chaque fonctionnalité (test-login.spec.ts, test-cart.spec.ts).
- utils/: contient des fonctions utilitaires réutilisables (génération de données aléatoires, gestion des cookies, etc.).

Suivre un schéma de nommage cohérent

Choisissez des noms clairs et cohérents pour vos fichiers, classes, fonctions, et variables. Cela facilite la navigation et la compréhension du code. Par exemple, nommez vos fichiers de test avec le préfixe test- suivi du nom de la fonctionnalité (test-login.spec.ts), et vos pages avec le suffixe Page (homePage.ts).

Utiliser des fichiers de configuration pour les variables globales

Placez les informations globales (URL de l'application, identifiants de test, options de configuration) dans un fichier de configuration (config.ts ou .env). Cela permet d'ajuster ces paramètres facilement sans modifier les tests eux-mêmes.

Exporter une constante dans le fichier playwright.config.ts :

```
playwright.config.ts > [o] default
1 import { defineConfig, devices } from '@playwright/test';
2
3 export const config = {
4   baseUrl: 'https://zenity.fr',
5   defaultTimeout: 5000,
6 };
7
8 export default defineConfig({
9   reporter: [
10 ]
```

Quelque soit la manière utilisée pour créer ses variable de configuration elle permet une utilisation variabilisé dans les tests à suivre :

```
await page.goto(config.baseUrl, { timeout: config.defaultTimeout });
await page.waitForLoadState('networkidle');
```

TESTS ROBUSTES ET RESILIENTS (EVITER LES FAUX POSITIFS/NEGATIFS)

Les tests robustes sont ceux qui ne cassent pas facilement face à des changements mineurs de l'interface utilisateur et qui fournissent des résultats fiables, minimisant ainsi les faux positifs (tests qui réussissent alors qu'ils devraient échouer) et les faux négatifs (tests qui échouent alors qu'ils devraient réussir).

Éviter les sélecteurs fragiles

Privilégiez des sélecteurs robustes et spécifiques pour identifier les éléments. Évitez les sélecteurs basés sur des attributs instables comme id ou class générés dynamiquement. Préférez l'utilisation des attributs data-testid ou aria-label :

```
// Mauvaise pratique (id dynamique)
await page.click('#btn-12345');

// Bonne pratique (attribut stable)
await page.click('[data-testid="login-button"]');
```

Gérer les actions asynchrones

Playwright propose des méthodes `waitForSelector`, `waitForResponse`, et `waitForNavigation` pour s'assurer que les éléments sont présents ou les actions terminées avant de poursuivre le test. Utilisez-les pour synchroniser correctement les actions :

```
await page.waitForSelector('[data-testid="login-success-message"]');
```

Gérer les temps d'attente intelligemment

Utilisez des options de timeout intelligentes pour éviter les longs temps d'attente par défaut, tout en laissant suffisamment de temps pour les actions qui peuvent prendre plus de temps (chargement de page, réponses d'API). Par exemple :

```
await page.waitForResponse((response) => response.url()
  .includes('/api/login')
  && response.status() === 200,
  { timeout: 5000 });
```

Utiliser les tests conditionnels

Implémentez des assertions conditionnelles pour vérifier les états sans interrompre l'exécution des tests lorsque certains éléments ne sont pas présents. Cela permet d'éviter les erreurs dues à des éléments absents :

```
if (await page.isVisible('[data-testid="optional-element"]')) {  
    await page.click('[data-testid="optional-element"]');  
}
```

Tests indépendants et isolés

Assurez-vous que chaque test est indépendant des autres. Évitez d'utiliser des variables ou des états globaux entre les tests, car cela peut entraîner des erreurs difficiles à diagnostiquer et à reproduire.

MAINTENANCE DES TESTS ET GESTION DES CHANGEMENTS DANS LES APPLICATIONS

Les tests d'automatisation doivent être maintenus activement pour suivre les évolutions de l'application. Voici quelques stratégies pour gérer efficacement les changements :

Automatiser les mises à jour des sélecteurs

Utilisez des outils comme Playwright Recorder ou d'autres extensions de navigateur pour capturer automatiquement les sélecteurs et identifier plus facilement les changements dans l'interface utilisateur.

Mise à jour centralisée des sélecteurs

Placez vos sélecteurs dans des fichiers séparés (selectors.ts) ou dans des classes de page pour pouvoir les modifier facilement lorsque l'application évolue :

The screenshot shows a Visual Studio Code interface. On the left is the code editor with a file named `HomePageLocators.ts`. The code defines an export const locators object with two properties: `myProfileButton` and `gameCard`, both using XPath expressions. A red box highlights this code. On the right is the Explorer sidebar, which displays the project structure. A second red box highlights the `pages` folder under `FORMATION-PLAYWRIGHT`, which contains files like `cmd`, `CommonPage`, `dpp`, `DrawGamePages`, `HomePage` (containing `HomePage.ts` and `HomePageLocators.ts`), `LoginPage`, `ValidationPage`, `playwright-report`, `test-results`, `tests`, `tests-examples`, `.gitignore`, `package-lock.json`, `package.json`, `playwright.config.ts`, and `rappor-tests.json`.

```
1  export const locators = {  
2      myProfileButton: "//*[@data-testid='user']",  
3      gameCard: "//*[@data-testid='card']"  
4  }
```

Pour pouvoir par exemple les utiliser dans un Page Object Model de manière dynamique et propre.

```

1 import { expect, Page } from "@playwright/test";
2 import { locators } from "./HomePageLocators";
3 import { drawGameInfoMap, DrawGame } from "./CommonPage/drawGamesConfig";
4 import { menuButtons, menuButtonsInfoMap } from "./CommonPage/menuButtonsConfig";
5
6
7 export class HomePage extends CommonPage {
8   constructor(public page: Page) {
9     super(page);
10   }
11
12   async verifyProfileButtonVisible() {
13     await expect.soft(this.page.locator(locators.myProfileButton)).toBeVisible();
14   }
15
16   async selectLotteryGame(drawGame: DrawGame) {
17     await this.page.click(`#${drawGameInfoMap[drawGame].id}`);
18     await this.page.waitForURL(`**${(drawGameInfoMap[drawGame].endpoint)}`);
19     expect(this.page).toHaveURL(new RegExp(`${(drawGameInfoMap[drawGame].endpoint)$}`));
20     await this.logCurrentPage();
21   }
22
23   async verifyHomePageDisplayed() {
24     console.log('BEFORE --> ${this.page.url()}');
25     await this.page.waitForURL(`**${menuButtonsInfoMap[MenuButtons.HOME].endpoint}`);
26     console.log('AFTER --> ${this.page.url()}');
27   }
28 }
29

```

Mettre en place des tests de régression visuelle

Les tests de régression visuelle comparent l'apparence de l'application entre deux versions. En ajoutant des tests visuels, vous pouvez détecter plus facilement les modifications inattendues de l'interface utilisateur.

Utiliser des tests de contrat d'API

Si votre application repose sur des API, ajoutez des tests de contrat d'API pour valider que les schémas de réponse n'ont pas changé. Cela permet d'identifier les impacts des modifications d'API sur vos tests d'intégration.

Réviser régulièrement les tests

Planifiez des revues de tests périodiques pour élaguer les anciens tests, en ajouter de nouveaux ou les modifier selon les nouvelles fonctionnalités ajoutées à l'application. Supprimez les tests obsolètes et regroupez les tests similaires pour réduire la complexité.

Utilisation d'un CI/CD pour exécuter les tests fréquemment

Intégrez vos tests dans un pipeline de CI/CD pour les exécuter à chaque changement de code. Cela vous permet de détecter rapidement les régressions introduites par les nouvelles modifications de l'application.

CONCLUSION

Suivre ces bonnes pratiques vous aidera à créer une suite de tests Playwright plus stable, fiable et maintenable. L'automatisation des tests est un investissement à long terme qui doit évoluer avec l'application. En adoptant une organisation modulaire, en utilisant des sélecteurs robustes, et en anticipant les changements, vous pouvez réduire le coût de maintenance de vos tests tout en garantissant la qualité et la fiabilité de vos applications web.



A TOI DE JOUER !

Répondez aux questions 67 à 73 du QCM de fin de document.



QCM

QUESTIONS

12. QCM

Attention il ne peut y avoir qu'une bonne réponse par question (sauf quand le contraire est précisé)



1. Qu'est-ce que Playwright ?

- Un outil de gestion de versions
- Un outil d'automatisation des tests web
- Un éditeur de texte pour le développement web
- Un moteur de rendu pour navigateurs

2. Quels navigateurs sont pris en charge par Playwright ?

- Chromium, Firefox, WebKit
- Chrome, Edge, Safari
- Opera, Chrome, Firefox
- Internet Explorer, Firefox, Safari

3. Quelle est un des principaux avantages de Playwright par rapport à Selenium ou Puppeteer ?

- Il supporte Javascript
- Il offre une meilleure gestion des logs
- Il est plus facile et rapide à mettre en place
- Il permet les tests multi-navigateurs

4. Dans quel cas d'utilisation Playwright est-il particulièrement adapté ?



- Automatisation des Tests de Performance
- Automatisation des tests sur plusieurs navigateurs
- Développement de systèmes embarqués
- Gestion des bases de données

5. Quel est un avantage clé de Playwright en matière de gestion des interactions avec une page web ?

- Il nécessite des scripts d'attente explicite pour chaque action
- Il attend automatiquement que les éléments soient prêts avant d'effectuer des actions
- Il ne supporte pas les interactions avec les iframes
- Il se limite à l'automatisation des formulaires simples

6. Quelle commande est utilisée pour installer Playwright avec npm ?

- npm install playwright
- npm init playwright
- npm start playwright
- npm playwright install

7. Quelle commande permet d'initialiser un projet Playwright avec des fichiers de configuration et des exemples de tests ?

- npx playwright install
- npx playwright test
- npx playwright init
- npx playwright setup

8. Quelle option dans le fichier `playwright.config.ts` permet de configurer les tests en mode sans interface graphique (headless) ?

- headless: false
- browser: 'headless'
- use: { headless: true }
- view: headless



9. Quelle méthode Playwright est utilisée pour naviguer vers une URL spécifique ?

- page.click()
- page.goto()
- page.fill()
- page.selectOption()

10. Quelle méthode Playwright est utilisée pour naviguer vers la page précédente ?

- page.goForward()
- page.goto(previous)
- page.goBack()
- page.selectOption(backward)

11. Quelle méthode est utilisée pour entrer du texte dans un champ de saisie ?

- page.click()
- page.type()
- page.fill()
- page.selectOption()

12. Comment vérifier qu'un élément est visible sur la page avec Playwright ?

- await expect(page.locator('selector')).toBe('visible');
- await expect(page.locator('selector')).toBeVisible();
- await expect(page.locator('selector')).state('visible');
- await expect(page.locator('selector')).toBe(visible);

13. Quelle méthode Playwright permet de sélectionner une option dans un menu déroulant ?

- page.click()
- page.goto()
- page.fill()
- page.selectOption()



14. Quelle méthode Playwright permet de rafraîchir la page actuelle ?

- page.refresh()
- page.goto(current)
- page.return()
- page.reload()

15. Quel est le rôle principal de l'outil Codegen dans Playwright ?

- Générer des rapports de test
- Créer des scripts de test à partir d'interactions utilisateur
- Exécuter des tests en parallèle
- Optimiser les performances des tests

16. Quelle commande permet de lancer Playwright avec Codegen ?

- npx playwright codegen <URL>
- npm install playwright-codegen
- npx playwright generate <URL>
- npm run codegen

17. **Quel avantage principal apporte l'utilisation de Pick Locator dans Codegen ?**

- Identifier rapidement les éléments interactifs à tester
- Générer des rapports de performance
- Configurer les tests pour plusieurs navigateurs
- Gérer les erreurs dans les scripts générés

18. Quel type de sélecteurs peut-on générer automatiquement avec Codegen ?

- XPath uniquement
- CSS uniquement
- data-testid, aria-label, CSS, XPath
- ID uniquement

19. **Est-il possible de démarrer Codegen sans spécifier d'URL au lancement ?**



- Oui, on peut naviguer manuellement vers le site de son choix
- Non, une URL est obligatoire au lancement
- Oui, mais uniquement en mode headless
- Non, sauf si le site est en local

20. **Que permet l'option `--save-storage` dans Codegen ?**

- Sauvegarder les interactions dans un fichier CSV
- Enregistrer l'état de stockage local pour une utilisation future
- Exporter les rapports de test
- Partager les scripts générés entre plusieurs projets

21. **Quelle est la bonne pratique recommandée lors de l'utilisation de Codegen ?**

- N'utiliser que des sélecteurs XPath
- Toujours réviser le code généré pour améliorer la maintenance
- Ne jamais modifier le code généré
- Lancer Codegen en mode headless

22. **Quelle méthode Playwright permet de manipuler les frames et iframes ?**

- page.evaluateHandle()
- page.navigateTo()
- page.frames()
- page.frame()

23. **Quelle méthode permet de gérer les temps d'attente (timeouts) dans Playwright ?**

- page.waitForSelector()
- page.setDefaultTimeout()
- page.waitForRequest()
- page.expectTimeout()

24. **Quelle commande est utilisée pour interagir avec un dialogue, une alerte ou un prompt dans Playwright ?**

- page.on('prompt', callback)
- page.on('dialog', callback)

- page.on('alert', callback)
- page.on('alertDismiss', callback)

25. Pour manipuler plusieurs fenêtres dans Playwright, quelle structure est utilisée ?

- MultiWindowContext
- BrowserContext
- page.context()
- windowHandler()

26. Comment capturer une exception de navigation lors de l'utilisation de frames ?

- Utiliser try...catch
- Utiliser frame.waitForNavigation()
- Utiliser frame.onError()
- Utiliser page.navigateCatch()

27. Quelle méthode est recommandée pour interagir avec un pop-up dans Playwright ?

- popup.handle()
- page.waitForPopup()
- page.popupIntercept()
- popup.dismiss()

28. Quelle option de configuration permet de réduire les temps d'attente pour les interactions ?

- slowMo
- timeout
- retry
- delay

29. Quelle méthode Playwright permet d'intercepter les requêtes réseau ?

- page.waitForResponse()
- page.interceptRequest()
- page.on('request', callback)





page.route()

30. Quelle méthode est utilisée pour simuler des réponses réseau dans Playwright ?

- page.route.fulfill()
- page.mockResponse()
- page.route.mock()
- page.setResponse()

31. Comment tester les scénarios où une API retourne une réponse avec une erreur 500 ?

- Utiliser page.exposeFunction() pour injecter un comportement de réponse
- Utiliser page.waitForRequest() pour attendre une requête avec erreur
- Utiliser page.route() et rediriger la requête avec route.fulfill()
- Utiliser page.mockError() pour générer une réponse 500

32. **Quel est l'avantage principal de l'interception des requêtes réseau lors des tests Playwright ?**

- Réduire le temps d'exécution des tests
- Simuler des réponses réseau sans dépendre du serveur
- Rendre les pages plus rapides à charger
- Capturer les logs de performance du réseau

33. **Comment vérifier le contenu d'une réponse réseau dans Playwright ?**

- Utiliser response.body() et comparer avec un JSON attendu
- Utiliser page.evaluate() pour extraire les données
- Utiliser request.content() pour vérifier le contenu
- Utiliser route.fulfill() et tester les logs

34. Quelle méthode permet de simuler des réponses réseau avec des données mockées dans Playwright ?

- page.route.mockResponse()
- route.fulfill()
- page.simulateResponse()



- route.override()

35. Quel type d'interception est possible avec page.route() ?

- Modifier la méthode HTTP (GET, POST, etc.) d'une requête
- Modifier uniquement le header de la requête
- Modifier la structure du DOM
- Modifier le contenu de la page actuelle

36. Quel package est recommandé pour effectuer des tests d'accessibilité avec Playwright ?

- @playwright/test-axe
- @playwright/accessibility
- @axe-core/playwright
- @accessibility/playwright

37. Quelle norme d'accessibilité web est mentionnée dans le chapitre ?

- WAI-ARIA
- WCAG
- ISO 9241
- Section 508

38. Quel événement permet de mesurer le moment où le contenu HTML est complètement chargé dans Playwright ?

- domcomplete
- load
- documentloaded
- DOMContentLoaded

39. Quelle méthode Playwright permet de tester les performances en évaluant le temps de chargement des ressources ?

- page.waitForTimeout()
- page.waitForLoadState()
- page.evaluatePerformance()
- page.waitForResponse()



40. Quel outil tiers est mentionné pour évaluer l'accessibilité dans les applications web ?

- Lighthouse
- Jest
- axe-core
- Puppeteer

41. Quelle est la principale utilisation du package axe-core dans les tests d'accessibilité ?

- Évaluer le score de performance de la page
- Générer un rapport d'accessibilité
- Simuler des interactions utilisateur
- Calculer le temps de réponse du serveur

42. Quelle directive est recommandée pour améliorer l'accessibilité des pages web ?

- Ajouter des animations dynamiques
- Utiliser des attributs ARIA pour les éléments interactifs
- Réduire la taille des fichiers JavaScript
- Ajouter des commentaires HTML explicatifs

43. Quelle méthode Playwright est utilisée pour gérer le téléchargement de fichiers ?

- page.downloadFile()
- page.on('download', callback)
- page.expectDownload()
- page.waitForDownload()

44. Comment vérifier le chemin d'un fichier téléchargé dans Playwright ?

- Utiliser download.path()
- Utiliser page.filePath()
- Utiliser download.verify()
- Utiliser page.getDownloadPath()



45. **Quelle méthode permet de gérer l'upload de fichiers dans Playwright ?**

- page.uploadFile()
- page.setInputFiles()
- page.upload()
- page.sendFile()

46. **Quel sélecteur est utilisé pour cibler les éléments permettant l'upload de fichiers ?**

- input[type="file"]
- button[type="file"]
- div.upload
- file-selector

47. **Comment s'assurer que le fichier téléchargé a été supprimé après le test ?**

- Utiliser fs.unlinkSync() pour supprimer le fichier
- Utiliser download.delete() pour le supprimer directement
- Utiliser page.removeFile()
- Utiliser fs.rmdirSync() pour effacer le répertoire des téléchargements

48. **Comment Playwright peut-il être utilisé pour tester le contenu d'un fichier téléchargé ?**

- Lire le contenu du fichier avec fs.readFileSync()
- Utiliser page.readFile()
- Utiliser download.read()
- Utiliser fs.verifyFileContent()

49. **Comment automatiser le téléchargement de fichiers tout en vérifiant le chemin et le nom du fichier ?**

- page.on('download', callback)
- page.waitForDownload().then(download => download.path())
- page.route('download').then(path => path)
- page.checkDownload()



50. Quelle option de configuration Playwright permet de définir le **nombre maximal de workers pour l'exécution en parallèle ?**
- parallelWorkers
 - maxThreads
 - workers
 - parallelExecution
51. Quel service cloud est mentionné dans le chapitre pour exécuter des tests Playwright sur différents navigateurs et **systèmes d'exploitation ?**
- AWS Lambda
 - BrowserStack
 - Jenkins
 - GitHub Actions
52. Quelle variable d'**environnement est nécessaire pour l'intégration de Playwright avec BrowserStack ?** plusieurs réponses possibles.
- BROWSERSTACK_URL
 - BROWSERSTACK_PROJECT_ID
 - BROWSERSTACK_USERNAME
 - BROWSERSTACK_ACCESS_KEY
53. Quelle méthode permet d'exécuter des tests Playwright en parallèle sur plusieurs navigateurs ?
- test.parallel()
 - test.run()
 - test.project()
 - test.describe.parallel()
54. Comment spécifier un navigateur particulier (ex : Firefox) pour un test dans Playwright ?
- test.use({ browserName: 'firefox' })
 - test.selectBrowser('firefox')
 - test.withBrowser('firefox')
 - test.browser('firefox')



55. Quel attribut de configuration permet de spécifier les navigateurs à utiliser pour les tests parallèles ?
- projects
 - parallelBrowsers
 - multiBrowser
 - browsers
56. Comment exécuter un test Playwright sur une infrastructure cloud tierce comme Sauce Labs ?
- Utiliser test.run() avec --cloud
 - Configurer les variables d'environnement nécessaires et exécuter avec npx playwright test
 - Déployer le projet Playwright sur le cloud
 - Lancer l'application Playwright avec un script custom
57. Quelle option de configuration permet de générer des rapports de tests dans Playwright ?
- reporters
 - generateReports
 - testReporter
 - outputReports
58. Quelle extension de fichier est couramment utilisée pour les rapports JSON de Playwright ?
- .json
 - .log
 - .txt
 - .yaml
59. **Comment capturer des captures d'écran lors des échecs de test dans Playwright ?**
- page.screenshot()
 - test.screenshotOnFailure = true
 - use: { screenshot: 'only-on-failure' }
 - test.on('fail', callback)



60. Quel outil est mentionné dans le chapitre pour visualiser les rapports générés par Playwright ?

- report-visualizer
- allure-playwright
- playwright-report
- junit-reporter

61. Quelle option permet de capturer des vidéos des sessions de test dans Playwright ?

- use: { video: 'on-first-retry' }
- page.recordVideo()
- test.captureVideo()
- videoRecorder.start()

62. Comment analyser les logs de requêtes réseau dans Playwright ?

- Utiliser page.on('request') et page.on('response')
- Utiliser networkMonitor()
- Utiliser test.logRequests()
- Utiliser captureNetworkLogs()

63. Quelle méthode permet d'ajouter un message personnalisé dans les logs de Playwright ?

- console.logMessage()
- page.logInfo()
- console.message()
- console.log()

64. Quelle pratique permet d'améliorer la résilience des tests dans Playwright ?

- Utiliser des wait explicites pour chaque action
- Utiliser des sélecteurs robustes et stables
- Réécrire le code des tests à chaque changement de l'application
- Éviter d'utiliser des assertions dans les tests



65. Quel est un des principaux avantages d'utiliser `waitForSelector` par rapport à un `wait` explicite ?

- `waitForSelector` attend de façon dynamique l'apparition d'un élément sur la page
- `waitForSelector` réduit la vitesse d'exécution du test
- `waitForSelector` simule mieux le comportement humain
- `waitForSelector` interrompt l'exécution du script en cas de timeout

66. Quelle méthode est recommandée pour structurer les tests dans Playwright ?

- Utiliser des `test.describe` pour regrouper les tests par fonctionnalité
- Tout placer dans un seul fichier `test.ts`
- Utiliser des classes pour chaque cas de test
- Écrire un script unique avec plusieurs assertions

67. Quel est l'intérêt d'utiliser `test.beforeEach` et `test.afterEach` ?

- Gérer les états de préparation et de nettoyage avant et après chaque test
- Réduire la complexité du code
- Déclarer des variables globales pour tous les tests
- Empêcher l'exécution de tests redondants

68. Comment éviter les sélecteurs instables dans Playwright ?

- Utiliser des sélecteurs basés sur les attributs `data-testid` ou `aria-label`
- Utiliser des sélecteurs CSS avec des classes dynamiques
- Utiliser des XPath pour tous les éléments
- Utiliser `page.locator('div')` pour chaque élément

69. Quel est le rôle du timeout dans Playwright ?

- Définir le temps maximal pour une action avant qu'elle ne soit considérée comme échouée
- Augmenter la durée de vie d'un test
- Spécifier la fréquence d'exécution des actions
- Ajouter un délai après chaque action



70. Quelle est une bonne pratique pour gérer les tests en cas de **modifications fréquentes de l'interface utilisateur** ?

- Mettre à jour les sélecteurs au niveau centralisé, par exemple dans un fichier de page object
- Éviter les tests d'interface utilisateur jusqu'à la stabilisation
- Recréer les tests à partir de zéro à chaque changement
- Utiliser des ID générés aléatoirement pour tous les sélecteurs

71. Comment améliorer la maintenance des tests Playwright ?

- Organiser les tests en modules et composants réutilisables
- Écrire tous les tests dans un seul fichier
- Ignorer les tests qui échouent souvent
- Supprimer les assertions trop strictes

72. Quelle méthode permet d'attendre qu'une navigation soit terminée dans Playwright ?

- page.waitForNavigation()
- page.waitForResponse()
- page.waitForSelector()
- page.waitForLoad()

73. Quelle pratique peut éviter les erreurs liées aux changements asynchrones dans une page ?

- Utiliser waitForSelector ou waitForLoadState pour attendre les états stables de la page
- Ajouter des setTimeout avant chaque action
- Mettre des sleep explicites dans chaque test
- Ignorer les erreurs pour éviter les faux positifs



QCM

RÉPONSES

CORRECTION QCM

1. Qu'est-ce que Playwright ?

- Un outil de gestion de versions
- Un outil d'automatisation des tests web
- Un éditeur de texte pour le développement web
- Un moteur de rendu pour navigateurs

Analyse : Playwright est bien un outil d'automatisation de tests end-to-end, permettant d'interagir avec différents navigateurs pour tester des applications web.

2. Quels navigateurs sont pris en charge par Playwright ?

- Chromium, Firefox, WebKit
- Chrome, Edge, Safari
- Opera, Chrome, Firefox
- Internet Explorer, Firefox, Safari

Analyse : Playwright supporte les moteurs Chromium (pour Chrome, Edge), Firefox et WebKit (pour Safari). Il ne supporte pas des navigateurs comme Opera ou Internet Explorer directement.

3. Quelle est un des principaux avantages de Playwright par rapport à Selenium ou Puppeteer ?

- Il supporte Javascript
- Il offre une meilleure gestion des logs
- Il est plus facile et rapide à mettre en place
- Il permet les tests multi-navigateurs

Analyse : Playwright se distingue par sa configuration simple et ses fonctionnalités intégrées (multi-navigateurs, captures d'écran, vidéos), ce qui le rend plus facile à mettre en œuvre que Selenium.

4. Dans quel cas d'utilisation Playwright est-il particulièrement adapté ?

- Automatisation des Tests de Performance
- Automatisation des tests sur plusieurs navigateurs
- Développement de systèmes embarqués
- Gestion des bases de données



Analyse : Playwright est conçu pour automatiser les tests sur plusieurs navigateurs et s'assure que le comportement de l'application est le même sur différents environnements.

5. Quel est un avantage clé de Playwright en matière de gestion des interactions avec une page web ?

- Il nécessite des scripts d'attente explicite pour chaque action
- Il attend automatiquement que les éléments soient prêts avant d'effectuer des actions
- Il ne supporte pas les interactions avec les iframes
- Il se limite à l'automatisation des formulaires simples

Analyse : Playwright attend automatiquement que les éléments de la page soient stables avant de réaliser une action, ce qui réduit le besoin d'ajouter des attentes explicites dans le code de test.

6. Quelle commande est utilisée pour installer Playwright avec npm ?

- npm install playwright
- npm init playwright
- npm start playwright
- npm playwright install

Analyse : `npm install playwright` installe Playwright à partir du registre npm.

7. Quelle commande permet d'initialiser un projet Playwright avec des fichiers de configuration et des exemples de tests ?

- npx playwright install
- npx playwright test
- npx playwright init
- npx playwright setup

Analyse : `npx playwright init` initialise un projet Playwright avec un modèle de configuration et des exemples de tests.

8. Quelle option dans le fichier `playwright.config.ts` permet de configurer les tests en mode sans interface graphique (headless) ?

- headless: false

- browser: 'headless'
- use: { headless: true }
- view: headless

Analyse : `use: { headless: true }` dans `playwright.config.ts` permet de configurer Playwright pour exécuter les tests en mode sans interface graphique (headless).

9. Quelle méthode Playwright est utilisée pour naviguer vers une URL spécifique ?

- page.click()
- page.goto()
- page.fill()
- page.selectOption()

Analyse : `page.goto()` permet de naviguer vers une URL spécifique dans Playwright.

10. Quelle méthode Playwright est utilisée pour naviguer vers la page précédente ?

- page.goForward()
- page.goto(previous)
- page.goBack()
- page.selectOption(backward)

Analyse : `page.goBack()` permet de naviguer vers la précédente page de l'historique dans Playwright.

11. Quelle méthode est utilisée pour entrer du texte dans un champ de saisie ?

- page.click()
- page.type()
- page.fill()
- page.selectOption()

Analyse : `page.fill()` est utilisé pour remplir un champ de saisie avec un texte donné.

12. Comment vérifier qu'un élément est visible sur la page avec Playwright ?

- await expect(page.locator('selector')).toBe('visible');
- await expect(page.locator('selector')).toBeVisible();



- await expect(page.locator('selector')).state('visible');
- await expect(page.locator('selector')).toBe(visible);

Analyse : La méthode `toBeVisible()` est utilisée pour vérifier la visibilité d'un élément avec les assertions de Playwright.

13. Quelle méthode Playwright permet de sélectionner une option dans un menu déroulant ?

- page.click('option')
- page.getOption()
- page.select('option')
- page.selectOption()

Analyse : `page.selectOption()` est utilisé pour sélectionner une option dans un `select` ou un menu déroulant.

14. Quelle méthode Playwright permet de rafraîchir la page actuelle ?

- page.refresh()
- page.goto(current)
- page.return()
- page.reload()

Analyse : `page.reload()` est utilisé rafraîchir la page actuelle.

15. Quel est le rôle principal de l'outil Codegen dans Playwright ?

- Générer des rapports de test
- Créer des scripts de test à partir d'interactions utilisateur
- Exécuter des tests en parallèle
- Optimiser les performances des tests

Analyse : Codegen est un générateur de scripts automatisé qui capture les interactions des utilisateurs avec une application web. Ces interactions sont converties en code Playwright, permettant de créer rapidement des scripts de tests end-to-end sans avoir à coder manuellement chaque action. C'est particulièrement utile pour les débutants en automatisation ou les utilisateurs expérimentés souhaitant accélérer le développement de scripts.



16. Quelle commande permet de lancer Playwright avec Codegen ?

- npx playwright codegen <URL>
- npm install playwright-codegen
- npx playwright generate <URL>
- npm run codegen

Analyse : La commande `npx playwright codegen <URL>` est celle utilisée pour lancer l'outil Codegen et commencer à enregistrer les actions de l'utilisateur sur la page spécifiée par l'URL. Elle démarre Playwright avec l'outil d'enregistrement actif, ce qui permet de capturer et de visualiser le script généré en temps réel.

17. Quel avantage principal apporte l'utilisation de `Pick Locator` dans Codegen ?

- Identifier rapidement les éléments interactifs à tester
- Générer des rapports de performance
- Configurer les tests pour plusieurs navigateurs
- Gérer les erreurs dans les scripts générés

Analyse : `Pick Locator` est une fonctionnalité de Codegen qui permet de sélectionner les éléments du DOM de manière interactive en cliquant sur eux. Cela aide à identifier et à générer des sélecteurs uniques, simplifiant ainsi le processus d'interaction avec les éléments de la page dans les scripts Playwright. C'est un gain de temps par rapport à la recherche manuelle des sélecteurs.

18. Quel type de sélecteurs peut-on générer automatiquement avec Codegen ?

- XPath uniquement
- CSS uniquement
- data-testid, aria-label, CSS, XPath
- ID uniquement

Analyse : Codegen permet de générer plusieurs types de sélecteurs, y compris `data-testid` (souvent utilisé dans les tests pour cibler des éléments), `aria-label` (pour l'accessibilité), CSS, et XPath. Il sélectionne automatiquement le type de sélecteur le plus approprié en fonction de la structure de la page pour garantir la robustesse du script généré.

19. Est-il possible de démarrer Codegen sans spécifier d'URL au lancement ?

- Oui, on peut naviguer manuellement vers le site de son choix

- Non, une URL est obligatoire au lancement
- Oui, mais uniquement en mode headless
- Non, sauf si le site est en local

Analyse : Il est possible de démarrer Codegen sans URL spécifique. Si aucune URL n'est fournie, un navigateur vide s'ouvre, permettant à l'utilisateur de naviguer manuellement vers le site de son choix. Cela peut être utile lorsque l'utilisateur veut se connecter ou parcourir différentes sections du site avant de commencer l'enregistrement des actions.

20. Que permet l'option `--save-storage` dans Codegen ?

- Sauvegarder les interactions dans un fichier CSV
- Enregistrer l'état de stockage local pour une utilisation future
- Exporter les rapports de test
- Partager les scripts générés entre plusieurs projets

Analyse : L'option `--save-storage` permet de sauvegarder l'état actuel du stockage local (`localStorage` et `cookies`) dans un fichier. Cela est particulièrement utile pour conserver l'état de la session (par exemple, lorsque l'utilisateur est connecté) et réutiliser cet état pour des tests ultérieurs sans devoir se reconnecter à chaque fois.

21. Quelle est la bonne pratique recommandée lors de l'utilisation de Codegen ?

- N'utiliser que des sélecteurs XPath
- Toujours réviser le code généré pour améliorer la maintenance
- Ne jamais modifier le code généré
- Lancer Codegen en mode headless

Analyse : Bien que Codegen génère du code rapidement et de manière efficace, il est important de toujours réviser le code généré. Les sélecteurs choisis par Codegen peuvent ne pas toujours être les plus robustes ou les plus lisibles. Réviser et ajuster ces sélecteurs améliore la maintenabilité et la résilience des tests, surtout lorsque la structure de la page est susceptible de changer.

22. Quelle méthode Playwright permet de manipuler les frames et iframes ?

- `page.evaluateHandle()`
- `page.navigateTo()`
- `page.frames()`
- `page.frame()`

Analyse : `page.frame()` permet d'accéder à un frame ou iframe particulier sur une page.

23. Quelle méthode permet de gérer les temps d'attente (timeouts) dans Playwright ?

- page.waitForSelector()
- page.setDefaultTimeout()
- page.waitForRequest()
- page.expectTimeout()

Analyse : `page.setDefaultTimeout()` permet de configurer un timeout par défaut pour toutes les actions sur une page.

24. Quelle commande est utilisée pour interagir avec un dialogue, une alerte ou un prompt dans Playwright ?

- page.on('prompt', callback)
- page.on('dialog', callback)
- page.on('alert', callback)
- page.on('alertDismiss', callback)

Analyse : `page.on('dialog', callback)` est utilisé pour capturer et interagir avec les dialogues, alertes ou prompts dans Playwright.

25. Pour manipuler plusieurs fenêtres dans Playwright, quelle structure est utilisée ?

- MultiWindowContext
- BrowserContext
- page.context()
- windowHandler()

Analyse : `BrowserContext` permet de gérer plusieurs fenêtres ou contextes de navigation indépendamment.

26. Comment capturer une exception de navigation lors de l'utilisation de frames ?

- Utiliser try...catch
- Utiliser frame.waitForNavigation()
- Utiliser frame.onError()
- Utiliser page.navigateCatch()

Analyse : Pour capturer une exception, il faut utiliser un `try...catch`.

27. Quelle méthode est recommandée pour interagir avec un pop-up dans Playwright ?

- `popup.handle()`
- `page.waitForPopup()`
- `page.popupIntercept()`
- `popup.dismiss()`

Analyse : `page.waitForPopup()` est utilisé pour attendre et capturer un nouvel onglet ou fenêtre pop-up ouvert.

28. Quelle option de configuration permet de réduire les temps d'attente pour les interactions ?

- `slowMo`
- `timeout`
- `retry`
- `delay`

Analyse : `timeout` dans Playwright permet de configurer le délai d'attente global pour les interactions.

29. Quelle méthode Playwright permet d'intercepter les requêtes réseau ?

- `page.waitForResponse()`
- `page.interceptRequest()`
- `page.on('request', callback)`
- `page.route()`

Analyse : La méthode `page.route()` est celle utilisée pour intercepter, modifier, ou mocker les requêtes réseau avant qu'elles ne soient envoyées au serveur. Les autres réponses ne sont pas pertinentes pour cette fonctionnalité.

30. Quelle méthode est utilisée pour simuler des réponses réseau dans Playwright ?

- `page.route.fulfill()`
- `page.mockResponse()`
- `page.route.mock()`
- `page.setResponse()`

Analyse : `route.fulfill()` est utilisé pour fournir une réponse personnalisée à une requête interceptée avec `page.route()`.

31. Comment tester les scénarios où une API retourne une réponse avec une erreur 500 ?

- Utiliser `page.exposeFunction()` pour injecter un comportement de réponse
- Utiliser `page.waitForRequest()` pour attendre une requête avec erreur
- Utiliser `page.route()` et rediriger la requête avec `route.fulfill()`
- Utiliser `page.mockError()` pour générer une réponse 500

Analyse : En interceptant les requêtes avec `page.route()`, on peut utiliser `route.fulfill()` pour simuler une réponse avec un statut d'erreur, comme 500.

32. Quel est l'avantage principal de l'interception des requêtes réseau lors des tests Playwright ?

- Réduire le temps d'exécution des tests
- Simuler des réponses réseau sans dépendre du serveur
- Rendre les pages plus rapides à charger
- Capturer les logs de performance du réseau

Analyse : L'interception des requêtes réseau permet d'effectuer des tests indépendamment du backend, en simulant des réponses personnalisées ou des scénarios d'erreur.

33. Comment vérifier le contenu d'une réponse réseau dans Playwright ?

- Utiliser `response.body()` et comparer avec un JSON attendu
- Utiliser `page.evaluate()` pour extraire les données
- Utiliser `request.content()` pour vérifier le contenu
- Utiliser `route.fulfill()` et tester les logs

Analyse : `response.body()` est utilisé pour récupérer le corps de la réponse et le vérifier. `page.evaluate()` ou `request.content()` ne permettent pas de valider directement le corps de la réponse réseau.

34. Quelle méthode permet de simuler des réponses réseau avec des données mockées dans Playwright ?

- `page.route.mockResponse()`
- `route.fulfill()`
- `page.simulateResponse()`
- `route.override()`

Analyse : `route.fill()` permet de simuler des réponses réseau avec des données personnalisées, ce qui est essentiel pour les tests basés sur des données mockées.

35. Quel type d'interception est possible avec `page.route()` ?

- Modifier la méthode HTTP (GET, POST, etc.) d'une requête
- Modifier uniquement le header de la requête
- Modifier la structure du DOM
- Modifier le contenu de la page actuelle

Analyse : `page.route()` permet de modifier la méthode, les headers, ou même le corps des requêtes interceptées.

36. Quel package est recommandé pour effectuer des tests d'accessibilité avec Playwright ?

- `@playwright/test-axe`
- `@playwright/accessibility`
- `@axe-core/playwright`
- `@accessibility/playwright`

Analyse : Cette réponse est exacte. Le package `@axe-core/playwright` est bien celui recommandé pour effectuer des tests d'accessibilité avec Playwright, car il permet d'analyser le DOM d'une page selon les règles de l'outil d'évaluation d'accessibilité `axe-core`.

37. Quelle norme d'accessibilité web est mentionnée dans le chapitre ?

- WAI-ARIA
- WCAG
- ISO 9241
- Section 508

Analyse : Cette réponse est correcte. La norme WCAG (Web Content Accessibility Guidelines) est la norme de référence pour l'accessibilité des contenus web et est souvent mentionnée dans le contexte de l'automatisation des tests d'accessibilité.

38. Quel événement permet de mesurer le moment où le contenu HTML est complètement chargé dans Playwright ?

- `domcomplete`
- `load`
- `documentloaded`

DOMContentLoaded

Analyse : L'événement `DOMContentLoaded` se déclenche lorsque le document HTML a été entièrement chargé et analysé, sans attendre que les feuilles de style, images et sous-documents soient complètement chargés.

39. Quelle méthode Playwright permet de tester les performances en évaluant le temps de chargement des ressources ?

- `page.waitForTimeout()`
- `page.waitForLoadState()`
- `page.evaluatePerformance()`
- `page.waitForResponse()`

Analyse : Exact. `page.waitForLoadState()` permet d'attendre différents états de chargement comme `load`, `domcontentloaded`, `networkidle`, ce qui est essentiel pour évaluer la performance de chargement des ressources d'une page.

40. Quel outil tiers est mentionné pour évaluer l'accessibilité dans les applications web ?

- Lighthouse
- Jest
- axe-core
- Puppeteer

Analyse : `axe-core` est l'outil tiers le plus couramment utilisé pour l'évaluation de l'accessibilité dans les applications web, et il est intégré directement dans Playwright via `@axe-core/playwright`.

41. Quelle est la principale utilisation du package axe-core dans les tests d'accessibilité ?

- Évaluer le score de performance de la page
- Générer un rapport d'accessibilité
- Simuler des interactions utilisateur
- Calculer le temps de réponse du serveur

Analyse : `axe-core` génère des rapports d'accessibilité détaillant les violations de normes d'accessibilité, les erreurs et les recommandations.

42. Quelle directive est recommandée pour améliorer l'accessibilité des pages web ?

- Ajouter des animations dynamiques

- Utiliser des attributs ARIA pour les éléments interactifs
- Réduire la taille des fichiers JavaScript
- Ajouter des commentaires HTML explicatifs

Analyse : L'utilisation des attributs ARIA est essentielle pour améliorer l'accessibilité des éléments interactifs sur les pages web, surtout pour les utilisateurs ayant recours à des technologies d'assistance.

43. Quelle méthode Playwright est utilisée pour gérer le téléchargement de fichiers ?

- page.downloadFile()
- page.on('download', callback)
- page.expectDownload()
- page.waitForDownload()

Analyse : La méthode `page.waitForDownload()` est utilisée pour attendre qu'un téléchargement démarre et capturer un objet `Download` qui contient des informations sur le fichier téléchargé.

44. Comment vérifier le chemin d'un fichier téléchargé dans Playwright ?

- Utiliser `download.path()`
- Utiliser `page.filePath()`
- Utiliser `download.verify()`
- Utiliser `page.getDownloadPath()`

Analyse : `download.path()` permet d'obtenir le chemin local du fichier téléchargé, ce qui est crucial pour vérifier la réussite du téléchargement et interagir avec le fichier.

45. Quelle méthode permet de gérer l'upload de fichiers dans Playwright ?

- page.uploadFile()
- page.setInputFiles()
- page.upload()
- page.sendFile()

Analyse : `page.setInputFiles()` est utilisé pour simuler l'upload d'un fichier en définissant les fichiers d'entrée sur un élément de type `input[type="file"]`.

46. Quel sélecteur est utilisé pour cibler les éléments permettant l'upload de fichiers ?



- input[type="file"]
- button[type="file"]
- div.upload
- file-selector

Analyse : `input[type="file"]` est le sélecteur standard pour cibler les éléments de formulaire permettant l'upload de fichiers. Les autres réponses ne sont pas valides.

47. Comment s'assurer que le fichier téléchargé a été supprimé après le test ?

- Utiliser `fs.unlinkSync()` pour supprimer le fichier
- Utiliser `download.delete()` pour le supprimer directement
- Utiliser `page.removeFile()`
- Utiliser `fs.rmdirSync()` pour effacer le répertoire des téléchargements

Analyse : `fs.unlinkSync()` (de Node.js) est la méthode appropriée pour supprimer un fichier de façon synchrone.

48. Comment Playwright peut-il être utilisé pour tester le contenu d'un fichier téléchargé ?

- Lire le contenu du fichier avec `fs.readFileSync()`
- Utiliser `page.readFile()`
- Utiliser `download.read()`
- Utiliser `fs.verifyFileContent()`

Analyse : `fs.readFileSync()` est couramment utilisé pour lire le contenu d'un fichier téléchargé et vérifier son intégrité.

49. Comment automatiser le téléchargement de fichiers tout en vérifiant le chemin et le nom du fichier ?

- `page.on('download', callback)`
- `page.waitForDownload().then(download => download.path())`
- `page.route('download').then(path => path)`
- `page.checkDownload()`

Analyse : `page.waitForDownload()` retourne une promesse résolue avec un objet `Download`, et `download.path()` permet de récupérer le chemin complet du fichier téléchargé.

50. Quelle option de configuration Playwright permet de définir le nombre maximal de workers pour l'exécution en parallèle ?



- parallelWorkers
- maxThreads
- workers
- parallelExecution

Analyse : L'option `workers` dans le fichier de configuration `playwright.config.ts` permet de définir le nombre maximal de workers pour les tests en parallèle.

51. Quel service cloud est mentionné dans le chapitre pour exécuter des tests Playwright sur différents navigateurs et systèmes d'exploitation ?

- AWS Lambda
- BrowserStack
- Jenkins
- GitHub Actions

Analyse : BrowserStack est un service cloud populaire pour l'exécution des tests automatisés Playwright sur différents navigateurs et systèmes d'exploitation. Les autres options (AWS Lambda, Jenkins, GitHub Actions) ne sont pas spécifiquement orientées vers l'exécution de tests cross-browser comme BrowserStack.

52. Quelle variable d'environnement est nécessaire pour l'intégration de Playwright avec BrowserStack ?

- BROWSERSTACK_URL
- BROWSERSTACK_PROJECT_ID
- BROWSERSTACK_USERNAME
- BROWSERSTACK_ACCESS_KEY

Analyse : `BROWSERSTACK_USERNAME` et `BROWSERSTACK_ACCESS_KEY` sont les deux variables d'environnement principales pour l'intégration avec BrowserStack.

53. Quelle méthode permet d'exécuter des tests Playwright en parallèle sur plusieurs navigateurs ?

- test.parallel()
- test.run()
- test.project()
- test.describe.parallel()

Analyse : `test.describe.parallel()` permet d'exécuter les tests au sein d'une `describe` block en parallèle. C'est une fonctionnalité qui s'applique lorsque l'on souhaite exécuter des tests sur différents navigateurs en parallèle, tout en gardant une cohérence dans les suites de tests.



54. Comment spécifier un navigateur particulier (ex : Firefox) pour un test dans Playwright ?

- test.use({ browserName: 'firefox' })
- test.selectBrowser('firefox')
- test.withBrowser('firefox')
- test.browser('firefox')

Analyse : `test.use()` permet de spécifier des options de configuration pour un test particulier, y compris le navigateur utilisé.

55. Quel attribut de configuration permet de spécifier les navigateurs à utiliser pour les tests parallèles ?

- projects
- parallelBrowsers
- multiBrowser
- browsers

Analyse : L'attribut `projects` dans le fichier de configuration `playwright.config.ts` est utilisé pour définir plusieurs configurations de tests, y compris les navigateurs, et exécuter les tests en parallèle sur ces navigateurs.

56. Comment exécuter un test Playwright sur une infrastructure cloud tierce comme Sauce Labs ?

- Utiliser `test.run()` avec `--cloud`
- Configurer les variables d'environnement nécessaires et exécuter avec `npx playwright test`
- Déployer le projet Playwright sur le cloud
- Lancer l'application Playwright avec un script custom

Analyse : Pour exécuter un test sur un cloud tiers, il faut généralement configurer les variables d'environnement telles que `SAUCE_USERNAME` et `SAUCE_ACCESS_KEY` et lancer les tests avec `npx playwright test`.

57. Quelle option de configuration permet de générer des rapports de tests dans Playwright ?

- reporters
- generateReports
- testReporter
- outputReports

Analyse : L'option `reporters` dans le fichier de configuration `playwright.config.ts` permet de spécifier les reporters pour générer des rapports de tests. Cela inclut des formats comme `list`, `json`, `html`, etc.

58. Quelle extension de fichier est couramment utilisée pour les rapports JSON de Playwright ?

- .json
- .log
- .txt
- .yaml

Analyse : Les rapports JSON de Playwright sont souvent générés avec l'extension `.json`, car cela permet une exploitation facile des données de test pour d'autres outils.

59. Comment capturer des captures d'écran lors des échecs de test dans Playwright ?

- `page.screenshot()`
- `test.screenshotOnFailure = true`
- `use: { screenshot: 'only-on-failure' }`
- `test.on('fail', callback)`

Analyse : Cette option de configuration dans `playwright.config.ts` permet de capturer automatiquement une capture d'écran uniquement lors d'un échec de test, ce qui est très utile pour diagnostiquer les erreurs.

60. Quel outil est mentionné dans le chapitre pour visualiser les rapports générés par Playwright ?

- `report-visualizer`
- `allure-playwright`
- `playwright-report`
- `junit-reporter`

Analyse : `playwright-report` est un outil intégré à Playwright pour visualiser les rapports de test. D'autres outils tiers comme `allure-playwright` OU `junit-reporter` Sont aussi utilisés pour des intégrations spécifiques.

61. Quelle option permet de capturer des vidéos des sessions de test dans Playwright ?

- `use: { video: 'on-first-retry' }`
- `page.recordVideo()`
- `test.captureVideo()`

- videoRecorder.start()

Analyse : Cette option dans `playwright.config.ts` permet de capturer une vidéo de la session de test lors de la première tentative de réexécution du test (en cas d'échec). Il existe d'autres options comme `on` ou `retain-on-failure`.

62. Comment analyser les logs de requêtes réseau dans Playwright ?

- Utiliser `page.on('request')` et `page.on('response')`
- Utiliser `networkMonitor()`
- Utiliser `test.logRequests()`
- Utiliser `captureNetworkLogs()`

Analyse : `page.on('request')` et `page.on('response')` permettent de capturer et d'analyser les requêtes réseau et les réponses pendant l'exécution du test. Cela permet d'accéder aux informations comme le contenu de la requête, les headers, le statut, etc.

63. Quelle méthode permet d'ajouter un message personnalisé dans les logs de Playwright ?

- `console.logMessage()`
- `page.logInfo()`
- `console.message()`
- `console.log()`

Analyse : `console.log()` est utilisé dans le code Playwright pour ajouter des messages personnalisés dans les logs. Il s'agit de la même méthode JavaScript utilisée pour enregistrer des informations de debug dans la console.

64. Quelle pratique permet d'améliorer la résilience des tests dans Playwright ?

- Utiliser des wait explicites pour chaque action
- Utiliser des sélecteurs robustes et stables
- Réécrire le code des tests à chaque changement de l'application
- Éviter d'utiliser des assertions dans les tests

Analyse : Utiliser des sélecteurs robustes (par exemple, basés sur des attributs comme `data-testid` ou `aria-label`) permet d'améliorer la résilience des tests face aux changements mineurs dans l'interface utilisateur.

65. Quel est un des principaux avantages d'utiliser `waitForSelector` par rapport à un wait explicite ?

- waitForSelector attend de façon dynamique l'apparition d'un élément sur la page
- waitForSelector réduit la vitesse d'exécution du test
- waitForSelector simule mieux le comportement humain
- waitForSelector interrompt l'exécution du script en cas de timeout

Analyse : waitForSelector est plus flexible car il attend l'apparition d'un élément jusqu'à ce qu'un timeout soit atteint, ce qui le rend plus adapté que les Wait explicites.

66. Quelle méthode est recommandée pour structurer les tests dans Playwright ?

- Utiliser des test.describe pour regrouper les tests par fonctionnalité
- Tout placer dans un seul fichier test.ts
- Utiliser des classes pour chaque cas de test
- Écrire un script unique avec plusieurs assertions

Analyse : test.describe est couramment utilisé pour regrouper les tests par fonctionnalité ou module, ce qui facilite la maintenance et la compréhension des tests.

67. Quel est l'intérêt d'utiliser test.beforeEach et test.afterEach ?

- Gérer les états de préparation et de nettoyage avant et après chaque test
- Réduire la complexité du code
- Déclarer des variables globales pour tous les tests
- Empêcher l'exécution de tests redondants

Analyse : test.beforeEach et test.afterEach sont utilisés pour initialiser ou nettoyer l'environnement de test (par exemple, initialiser les sessions, supprimer les cookies).

68. Comment éviter les sélecteurs instables dans Playwright ?

- Utiliser des sélecteurs basés sur les attributs data-testid ou aria-label
- Utiliser des sélecteurs CSS avec des classes dynamiques
- Utiliser des XPath pour tous les éléments
- Utiliser page.locator('div') pour chaque élément

Analyse : Les sélecteurs basés sur data-testid ou aria-label sont moins susceptibles de changer par rapport aux sélecteurs CSS basés sur les classes ou les XPath.

69. Quel est le rôle du timeout dans Playwright ?

- Définir le temps maximal pour une action avant qu'elle ne soit considérée comme échouée
- Augmenter la durée de vie d'un test
- Spécifier la fréquence d'exécution des actions
- Ajouter un délai après chaque action

Analyse : Les timeout dans Playwright définissent le temps maximal d'attente pour une action ou une assertion avant qu'elle ne soit interrompue.

70. Quelle est une bonne pratique pour gérer les tests en cas de modifications fréquentes de l'interface utilisateur ?

- Mettre à jour les sélecteurs au niveau centralisé, par exemple dans un fichier de page object
- Éviter les tests d'interface utilisateur jusqu'à la stabilisation
- Recréer les tests à partir de zéro à chaque changement
- Utiliser des ID générés aléatoirement pour tous les sélecteurs

Analyse : Centraliser les sélecteurs dans des fichiers de type *Page Object* permet de ne modifier qu'un seul endroit en cas de changements, ce qui réduit la maintenance.

71. Comment améliorer la maintenance des tests Playwright ?

- Organiser les tests en modules et composants réutilisables
- Écrire tous les tests dans un seul fichier
- Ignorer les tests qui échouent souvent
- Supprimer les assertions trop strictes

Analyse : Modulariser les tests en composants réutilisables (e.g., Page Objects, fonctions utilitaires) facilite la maintenance et réduit les répétitions de code.

72. Quelle méthode permet d'attendre qu'une navigation soit terminée dans Playwright ?

- page.waitForNavigation()
- page.waitForResponse()
- page.waitForSelector()
- page.waitForLoad()

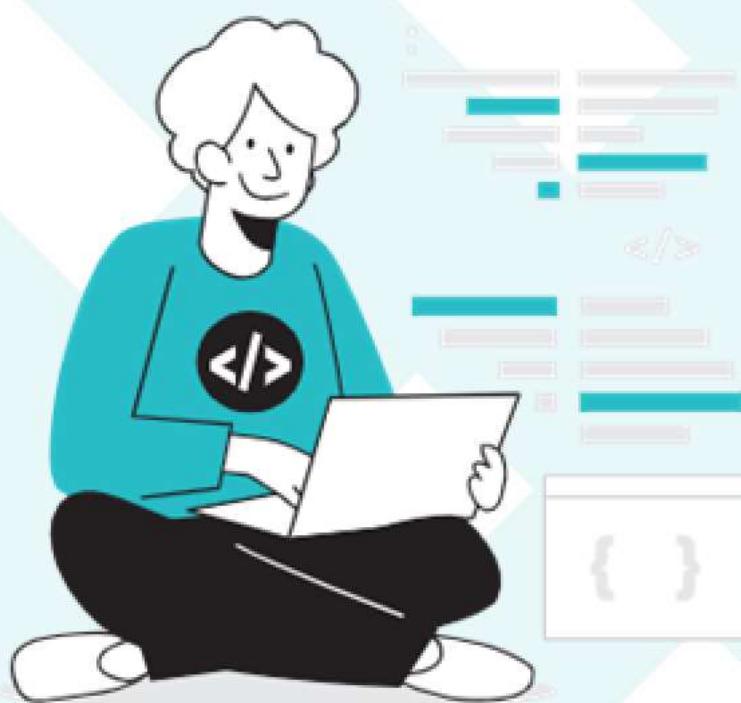
Analyse : page.waitForNavigation() attend la complétion de la navigation, ce qui est essentiel pour s'assurer que la page est prête avant de réaliser des actions.

73. Quelle pratique peut éviter les erreurs liées aux changements asynchrones dans une page ?

- Utiliser waitForSelector ou waitForLoadState pour attendre les états stables de la page
- Ajouter des setTimeout avant chaque action
- Mettre des sleep explicites dans chaque test
- Ignorer les erreurs pour éviter les faux positifs

Analyse : Utiliser waitForSelector ou waitForLoadState permet de gérer les transitions et états asynchrones, garantissant que la page est dans un

TP



TP

13. TP01

Créer un environnement de test Playwright

MISE EN SITUATION

Créer un environnement de test Playwright.

1. Installez Node.js sur votre machine.
2. Créez un nouveau projet Playwright (TypeScript) dans un nouveau répertoire `formation-playwright-tp`
3. Créez dans un fichier de `test_tp01.test.ts` un script de test qui navigue vers une URL de votre choix, puis vérifie que le titre de la page est correct.
4. Créez 3 scripts de test supplémentaires qui navigue vers **d'autres pages de ce même site, puis vérifie que le titre de la page est correct.**

5. Configurez le fichier `playwright.config.ts` de base en vous aidant de la [spécification](#)
 - a. url global
 - b. timeout global et timeout par test
 - c. test uniquement sur Google Chrome
 - d. rejet de test
 - e. tests en parallèle
 - f. tests ralenti à 1s par étape

CORRECTION TP01

playwright.config.ts

```
import { defineConfig, devices } from '@playwright/test';

export default defineConfig({

  testDir: './tests',
  fullyParallel: true,
  retries: 2,
  workers: 4,
  reporter: 'html',
  timeout: 30000, // Délai d'attente pour chaque test individuel (30 secondes)
  globalTimeout: 60 * 60 * 1000, // Délai d'attente global pour l'ensemble des tests (1 heure)

  use: [
    baseURL: 'https://zenity.fr',
    trace: 'on-first-retry',
    headless: false,
    launchOptions: {
      slowMo: 1000
    }
  ],

  projects: [
    {
      name: 'chromium',
      use: { ...devices['Desktop Chrome'] },
    }
  ]
})
```

tp01.test.ts

```
import { test, expect, } from '@playwright/test';

test(`Le titre de la page d'accueil s'affiche correctement`, async ({ page }) => {
    // Naviguer vers la page d'accueil
    await page.goto('./');

    // Afficher l'URL de la page dans la console
    console.log(`L'URL de la page est : ${page.url()}`);

    // Afficher le titre de la page dans la console
    console.log(`Le titre de la page affichée est : ${await page.title()}`);

    // Vérifier que le titre de la page correspond à l'expression régulière spécifiée
    await expect(page).toHaveTitle(/Pure Player du Test Logiciel \| Zenity/);
});

test(`Le titre de la page Identité s'affiche correctement`, async ({ page }) => {
    // Naviguer vers la page d'accueil
    await page.goto('./identite');

    // Afficher l'URL de la page dans la console
    console.log(`L'URL de la page est : ${page.url()}`);

    // Afficher le titre de la page dans la console
    console.log(`Le titre de la page affichée est : ${await page.title()}`);

    // Vérifier que le titre de la page correspond à l'expression régulière spécifiée
    await expect(page).toHaveTitle(/Identité \| Zenity/);
});

test(`Le titre de la page inside Zenity s'affiche correctement`, async ({ page }) => {
    // Naviguer vers la page d'accueil
    await page.goto('./inside-zenity');

    // Afficher l'URL de la page dans la console
    console.log(`L'URL de la page est : ${page.url()}`);

    // Afficher le titre de la page dans la console
    console.log(`Le titre de la page affichée est : ${await page.title()}`);

    // Vérifier que le titre de la page correspond à l'expression régulière spécifiée
    await expect(page).toHaveTitle(/Inside Zenity \| Zenity/);
});

... ainsi de suite
```

TP

14. TP02

Interaction avec les éléments d'une page
web

MISE EN SITUATION

1. Créez un script de test qui ouvre la page de Ztrain
<https://ztrain-web.vercel.app/home>
2. **Effectuez un test d'inscription.**

CORRECTION TP02

Tp02.test.ts



```
import { test, expect, } from '@playwright/test';

const password = '123456789';
const email = generateRandomEmail();

test(`Vérification de l'inscription d'un nouvel utilisateur avec des informations valides`, async ({ page }) => {
    // Naviguer vers la page d'accueil de l'application
    await page.goto('./home');

    // Cliquer sur le deuxième élément du conteneur d'avatars (par exemple, pour ouvrir un menu utilisateur ou une option de connexion)
    await page.locator('#style_avatar_wrapper__pEGIQ span').nth(1).click();

    // Sélectionner l'onglet "Inscription" pour accéder au formulaire d'inscription
    await page.getByRole('tab', { name: 'Inscription' }).click();

    // Cliquer sur le champ Email pour le sélectionner
    await page.getByPlaceholder('Email').click();

    // Remplir le champ Email avec la variable `email`
    await page.getByPlaceholder('Email').fill(email);

    // Remplir le champ Mot de passe avec la variable `password`
    await page.getByPlaceholder('Mot de passe', { exact: true }).fill(password);

    // Remplir le champ "Confirmer votre mot de passe" avec la variable `password`
```

TP

15. TP03

Utiliser l'outil Codegen

MISE EN SITUATION

1. **Lancez l'outil Codegen à l'url de Ztrain**
<https://ztrain-web.vercel.app/home>
2. Naviguez sur le site
 - a. Ajoutez 3 articles au panier
 - b. Vérifiez que les articles apparaissent bien dans le panier (prix et nom)
3. Analysez le script généré et exécutez-le pour vérifier qu'il reproduit bien vos actions.

CORRECTION TP03

Tp03.test.ts

```
import { test, expect, } from '@playwright/test';

const products = [
  {
    title: `PC Portable 15.6" FHD IPS Argent naturel (Intel core i5, RAM 8 Go, SSD 512 Go, AZERTY, Windows 10)`,
    price: `1325.80 €`
  },
  {
    title: `Machine à café rouge de la marque nespresso`,
    price: `50.00 €`
  },
  {
    title: `T-shirt en coton biologique`,
    price: `8.99 €`
  }
];

test(`Vérification de l'ajout de produits au panier depuis la page d'accueil`, async ({ page }) =>
{
  // Naviguer vers la page d'accueil de l'application
  await page.goto(`./home`);

  // Faire défiler la page vers le bas
  await page.mouse.wheel(0, 300);
}
```

```
// Boucle pour chaque produit
let i = 0;

for (const product of products) {
    i +=1;

    // Survoler le produit pour afficher le bouton d'ajout
    await page.getText(product.title).hover();

    // Cliquer sur le bouton d'ajout au panier
    await page.locator(`//h5[contains(text(), "${product.title.substring(0, 10)}")]/following-sibling::button`).click();

    // Cliquer sur le bouton Panier
    await page.locator(`//div[@id="style_content_cart_wrapper__mqNbf"]`).click();

    // Vérifier que les éléments sont bien ajoutés au panier
    await expect.soft(page.locator(`//div[@id='style_card_wrapper__hrc1I']/div[${i}]/p[1]`)).toContainText(product.title.substring(0, 10));
    await expect.soft(page.locator(`//div[@id='style_card_wrapper__hrc1I']/div[${i}]/p[2]`)).toContainText(product.price);

    // Fermer le panier
    await page.locator(`//h3[contains(text(), "Mon panier")]/preceding-sibling::div[1}`).click();
}
});
```

TP

16. TP04

Gérer des fenêtres et contextes multiples

MISE EN SITUATION

Écrivez un test qui a 3 contextes :

1. Dans cette fenêtre initiale (1) :
 - a. naviguez vers : ztrain-web.vercel.app/home
 - b. connectez-vous
 - c. vérifiez si vous êtes connecté
2. Dans un nouvel onglet (2) :
 - a. naviguez vers : ztrain-web.vercel.app/home
 - b. vérifiez si vous êtes connecté
 - c. ajoutez un article au panier
 - d. vérifiez si l'article est ajouté au panier
3. Dans cette nouvelle fenêtre (3) :

- a. naviguez vers : ztrain-web.vercel.app/home
- b. vérifiez si vous êtes connecté
- c. vérifiez si l'article est ajouté au panier

CORRECTION TP04

Oui le test est en échec c'est normal.

tp04.test.ts

```
import { test, expect, Page } from '@playwright/test';

const email = 'test.test@test.com';
const password = 'test1234';
const product = {
  title: `Machine à café rouge de la marque nespresso`,
  price: `50.00 €`
};
```

```
test(`Vérification de la persistance des données entre onglets et fenêtres`, async ({ browser }) =>
{
  // Créer un nouveau contexte de navigateur (une nouvelle fenêtre)
  const context = await browser.newContext();
  const page = await context.newPage();

  // Naviguer vers la page d'accueil de l'application
  await page.goto(`./home`);

  // Effectuer la connexion
  await page.locator('#style_avatar_wrapper span').nth(1).click();
  await page.getByPlaceholder('Email').fill(email);
  await page.getByPlaceholder('Mot de passe').fill(password);
  await page.getByRole('button', { name: 'Connexion', exact: true }).click();

  // Vérifier que l'email est affiché après connexion
  await verifyLogin(page);
}
```

```
// Ouvrir un nouvel onglet dans le même contexte
const newTab = await context.newPage();
await newTab.goto(`./home`);

// Vérifier que l'email est affiché après connexion dans le nouvel onglet
await verifyLogin(newTab);

// Faire défiler la page vers le bas pour afficher le produit
await newTab.mouse.wheel(0, 300);

// Survoler le produit pour afficher le bouton d'ajout au panier
await newTab.getText(product.title).hover();

// Cliquer sur le bouton d'ajout au panier
await newTab.locator(`//h5[contains(text(), "${product.title.substring(0, 10)}")]`).click();

// Ouvrir le Panier
await newTab.locator(`#style_content_cart_wrapper_mqNbf`).click();

// Vérifier que l'article est bien ajouté au panier
await verifyBasket(newTab, product.title.substring(0, 10), product.price);

// Ouvrir une nouvelle fenêtre (nouveau contexte de navigateur)
const newWindowContext = await browser.newContext();
const newWindow = await newWindowContext.newPage();
await newWindow.goto(`./home`);

// Vérifier que l'email est affiché après connexion dans la nouvelle fenêtre
await verifyLogin(newWindow);

// Cliquer sur le bouton Panier
await newWindow.locator(`#style_content_cart_wrapper_mqNbf`).click();

// Vérifier que les éléments sont bien ajoutés au panier dans la nouvelle fenêtre
await verifyBasket(newWindow, product.title.substring(0, 10), product.price);
});

async function verifyLogin(page: Page) {
  await expect.soft(page.locator('#style_avatar_wrapper_pEGIQ')).toContainText(email);
}

async function verifyBasket(page: Page, pageTitle: string, productPrice: string) {
  await expect.soft(page.locator(`div[@id='style_card_wrapper_hrc1I']/div[1]/p[1]`)).toContainText(pageTitle);
  await expect.soft(page.locator(`div[@id='style_card_wrapper_hrc1I']/div[1]/p[2]`)).toContainText(productPrice);
}
```

TP

17. TP05

Interception et modification des réponses
http. Téléchargement et vérification d'un
fichier.

MISE EN SITUATION

1. Ecrivez un script qui intercepte toutes les transactions API **déclenchées par l'accès à l'url** : ztrain-web.vercel.app/home
2. Sur la requête qui récupère la liste des produits :
 - a. créez un .json à partir de la réponse à cette requête

- b. dupliquez manuellement ce *.json* et dupliquez un produit afin qu'il apparaisse deux fois
- c. simulez la réponse à cette requête avec le *.json* créé à l'étape précédente
- d. vérifiez que l'article apparaît bien deux fois sur l'application
- e. prenez une capture d'écran de la page

CORRECTION TPO5

tp05.test.ts

```
test(`Remplacement de la réponse API avec un fichier JSON local`, async ({ page }) => {
  const jsonMockPath = './tp05_test-data/product_response_mock.json';

  // Lecture le fichier JSON local
  const data = fs.readFileSync(jsonMockPath, 'utf8'); // Spécifier 'utf8' pour obtenir un string

  // Analyse les données JSON
  const responseBodyMock = JSON.parse(data);
  console.log(responseBodyMock);

  // Configuration l'interception de l'appel API et remplacer les données du JSON
  await page.route('https://api-ztrain.onrender.com/product/', async route => {
    await route.fulfill({
      contentType: 'application/json', // Définir le type de contenu
      body: JSON.stringify(responseBodyMock) // Envoyer le corps de réponse fictif
    });
  });

  // Navigation vers la page d'accueil
  await page.goto('./home', { waitUntil: 'networkidle' });

  // Scroll de la page vers le bas pour afficher le produit
  await page.mouse.wheel(0, 300);
  const productElements = page.getByText("PC Portable 15.6\" FHD IPS Argent naturel (Intel core i5, RAM 8 Go, SSD 512 Go, AZERTY, Windows 10)");
}
```

TP

18. TP06

Exécuter des tests en parallèle et
Exploiter le rapport d'exécution

MISE EN SITUATION

1. Configurez le fichier `playwright.config.ts`
 - a. pour lancer les tests en headless
 - b. pour permettre l'exécution en parallèle sur les navigateurs Chrome, Firefox et Opera.
 - c. pour lancer 4 tests en parallèle
 - d. pour rejouer les tests en échec
 - e. pour enregistrer des captures d'écran
 - f. pour enregistrer des vidéos en cas d'échec
2. Configurez Playwright pour générer des rapports sous forme de fichiers HTML.

3. Exécutez l'ensemble des tests et générez les rapports.
4. Analysez les rapports pour identifier les tests réussis et échoués.

CORRECTION TP06

Playwright.config.ts

```
import { defineConfig, devices } from '@playwright/test';

export default defineConfig({
  testDir: './tests',
  fullyParallel: true,
  retries: 1,
  workers: 4,
  reporter: 'html',
  timeout: 120000, // Délai d'attente pour chaque test individuel (120 secondes)
  globalTimeout: 60 * 60 * 1000, // Délai d'attente global pour l'ensemble des tests (1 heure)
  use: [
    {
      baseURL: 'https://ztrain-web.vercel.app',
      trace: 'on-first-retry',
      headless: true,
      screenshot: "on",
      video: "on-first-retry",
      launchOptions: {
        slowMo: 2000
      }
    },
    projects: [
      {
        name: 'chromium',
        use: { ...devices['Desktop Chrome'] },
      },
      {
        name: "firefox",
        use: { ...devices["Desktop Firefox"] },
      },
      {
        name: "opera",
        use: { ...devices["Desktop Opera"] },
      },
    ],
  });
});
```

TP

19. TP07

Analyse d'accessibilité avec axe-core

MISE EN SITUATION

1. Installez le module `@axe-core/playwright`
2. Créez un script de test qui évalue l'accessibilité d'une page de votre choix.
3. Affichez les résultats et identifiez les éléments non conformes.

CORRECTION TP07

Dans le terminal :

```
PS D:\ZENITY\Projects\formation-playwright-tp> npm install @axe-core/playwright
added 2 packages, and audited 8 packages in 2s
found 0 vulnerabilities
PS D:\ZENITY\Projects\formation-playwright-tp>
```

```
import { test, expect } from '@playwright/test';
import { AxeBuilder } from '@axe-core/playwright';

test(`Évaluation de l'accessibilité d'une page`, async ({ page }) => {
    // Naviguer vers la page à tester
    await page.goto('https://youtube.com');

    // Créer une instance d'AxeBuilder
    const axe = new AxeBuilder({ page });

    // Exécuter l'évaluation de l'accessibilité
    const results = await axe.analyze();

    // Afficher les résultats dans la console
    console.log(`Résultats de l'évaluation de l'accessibilité :`, results);

    // Vérifier le nombre d'erreurs d'accessibilité
    expect.soft(results.violations.length).toBe(0); // Vérifie qu'il n'y a pas de violations

    // Affichage des violations
    if (results.violations.length > 0) {
        console.log('Éléments non conformes :');
        results.violations.forEach(violation => {
            console.log(`\nViolation : ${violation.description}`);
            violation.nodes.forEach(node => {
                console.log(` - Élément : ${node.html}`);
            });
        });
    }
});
```

