



TP2 – Page Object Model (POM)

Objectifs

- Comprendre le concept du Page Object Model
- Structurer les tests Playwright avec des classes représentant les pages
- Réutiliser le code pour simplifier l'écriture et la maintenance des tests
- Séparer la logique des tests de la manipulation des éléments UI

1. Introduction au Page Object Model

1.1 Définition

Le Page Object Model (POM) est un **patron de conception** qui consiste à représenter chaque page de l'application par une classe.

Cette classe contient :

- Les sélecteurs des éléments de la page
- Les méthodes permettant d'interagir avec la page

1.2 Avantages

- Lisibilité améliorée des tests
- Réduction de la duplication de code

- Maintenance facilitée lors de changements UI

2. Structure du projet avec POM

2.1 Organisation recommandée

```
projet/  
├─ tests/  
│   └─ todo.spec.ts  
├─ pages/  
│   └─ TodoPage.ts  
├─ package.json  
└─ playwright.config.ts
```

- `pages/` : contient les classes représentant les pages
- `tests/` : contient les scénarios de test en utilisant les classes
- `playwright.config.ts` : configuration globale des tests

3. Création de la classe Page Object

Créer `pages/TodoPage.ts` :

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

export class TodoPage {
  readonly page: Page;
  readonly todoInput = 'input[placeholder="What needs to be done?"]';
  readonly todoList = '.todo-list li';

  constructor(page: Page) {
    this.page = page;
  }

  async goto() {
    await this.page.goto('https://demo.playwright.dev/todomvc');
  }

  async addTask(task: string) {
    await this.page.fill(this.todoInput, task);
    await this.page.keyboard.press('Enter');
  }

  async deleteTask(task: string) {
    const todoItem = this.page.locator(`xpath=//label[text()="${task}"]/..`);
    await todoItem.hover();
    await todoItem.locator('.destroy').click();
  }

  async completeTask(task: string) {
    const todoItem = this.page.locator(`xpath=//label[text()="${task}"]/../input[@class='toggle']`);
    await todoItem.check();
  }

  async isTaskVisible(task: string) {
    await expect(this.page.getByText(task)).toBeVisible();
  }

  async isTaskCompleted(task: string) {
    const todoItem = this.page.locator(`xpath=//label[text()="${task}"]/../..`);
    await expect(todoItem).toHaveClass(/completed/);
  }
}
```

4. Utilisation de la Page Object dans un test

Créer `tests/todo.spec.ts` :

```
import { test } from '@playwright/test';
import { TodoPage } from '../pages/TodoPage';

test('ajouter et compléter une tâche', async ({ page }) => {
  const todoPage = new TodoPage(page);

  await todoPage.goto();
  await todoPage.addTask('Acheter du café');
  await todoPage.isTaskVisible('Acheter du café');

  await todoPage.completeTask('Acheter du café');
  await todoPage.isTaskCompleted('Acheter du café');
});
```

Exécuter le test :

```
npx playwright test tests/todo.spec.ts --headed --project=chromium
```

5. Refactorisation et bonnes pratiques

5.1 Séparer les responsabilités

- Les méthodes de la classe page **ne doivent jamais contenir de logique métier du test**
- Les assertions doivent idéalement rester dans le test principal ou être exposées via des méthodes explicites (ex: `isTaskVisible`)

5.2 Nommage

- Classes → nom de la page (`TodoPage`)
- Méthodes → actions sur la page (`addTask` , `deleteTask` , `completeTask`)

5.3 Réutilisation

- Les méthodes de la Page Object peuvent être utilisées dans plusieurs tests différents
- Eviter les sélecteurs codés en dur dans le test, toujours passer par la classe

6. Exercice pratique

Reprendre le `tests/multiple-tasks.spec.ts` du TP1, le refactoriser en appliquant le POM

7. Conclusion

À la fin de ce TP, vous devez être capables de :

- Créer des classes représentant les pages (POM)
- Refactoriser vos tests pour utiliser ces classes
- Simplifier la maintenance des tests E2E
- Écrire des tests clairs et réutilisables
- Séparer clairement la logique de test et la manipulation de l'interface