**Session 2: Playwright Automation Core Concepts**

**Slide 1: Introduction to Playwright Core Concepts**

* **Title:** Playwright Core Concepts
* **Content:**
  + Understanding the core components of Playwright
  + Importance of asynchronous programming in Playwright
  + Key features: browser context, page fixtures, and multi-browser support
* **Speaker Notes:**
  + Highlight the importance of these core concepts in building robust and scalable Playwright tests by addressing common challenges like flaky tests, handling multiple browser sessions, and ensuring cross-browser compatibility. Discuss how asynchronous programming allows efficient resource utilisation and prevents blocking operations, while browser contexts simplify multi-user test scenarios, and configuration management centralises settings for consistency across test suites.
  + Explain that Playwright’s design supports modern web automation needs, including multi-tab, multi-browser, and asynchronous operations.

**Slide 2: Async and Await in Playwright**

* **Title:** Async and Await
* **Content:**
  + Playwright operations are asynchronous.
  + **Why async/await?**
    - Handles non-blocking I/O operations.
    - Simplifies working with Promises in JavaScript/TypeScript.
  + Syntax example:
  + const { chromium } = require('playwright');
  + (async () => {
  + const browser = await chromium.launch();
  + const page = await browser.newPage();
  + await page.goto('https://example.com');
  + console.log(await page.title());
  + await browser.close();
  + })();
* **Speaker Notes:**
  + Emphasise that async/await ensures sequential execution of asynchronous tasks.
  + Mention that without async/await, code could become harder to read and debug.
* **Demo:**
  + Show how removing await causes errors or incomplete executions.

**Slide 3: Browser Contexts and Page Fixtures**

* **Title:** Browser Contexts and Page Fixtures
* **Content:**
  + **Browser Contexts:**
    - Simulate multiple independent browser sessions.
    - Useful for testing multi-user workflows.
    - Example:
    - const context = await browser.newContext();
    - const page = await context.newPage();
  + **Page Fixtures:**
    - Represents a single tab or iframe in a browser.
    - Pages are where most interactions happen.
* **Speaker Notes:**
  + Explain how contexts allow isolated sessions (e.g., separate cookies, storage).
  + Highlight that creating a new browser instance for each test is inefficient compared to using contexts.
* **Demo:**
  + Show two contexts accessing the same site but maintaining separate sessions.
  + Code example:
  + const user1 = await browser.newContext();
  + const user2 = await browser.newContext();
  + const page1 = await user1.newPage();
  + const page2 = await user2.newPage();
  + await page1.goto('https://example.com');
  + await page2.goto('https://example.com');
  + // Perform actions as two different users

**Slide 4: Playwright Configuration File**

* **Title:** Playwright Configuration File
* **Content:**
  + **Purpose:** Centralised management of test settings.
  + Key configuration options:
    - Browsers to use (e.g., chromium, firefox, webkit).
    - Test directory and timeout.
    - Reporters and retries.
  + Example playwright.config.ts:
  + import { defineConfig } from '@playwright/test';
  + export default defineConfig({
  + timeout: 30000,
  + use: {
  + browserName: 'chromium',
  + headless: false,
  + },
  + projects: [
  + { name: 'Desktop Chrome', use: { browserName: 'chromium' } },
  + { name: 'Mobile Safari', use: { browserName: 'webkit', viewport: { width: 375, height: 667 } } },
  + ],
  + });
* **Speaker Notes:**
  + Stress how this file simplifies test suite configuration.
  + Mention best practices, like using environment variables for sensitive settings.
* **Demo:**
  + Create and modify a playwright.config.ts file in an example project.
  + Run tests with different configurations.

**Slide 5: Multi-Browser Support**

* **Title:** Multi-Browser Support
* **Content:**
  + Playwright supports Chromium, WebKit, and Firefox.
  + Why multi-browser testing?
    - Ensures compatibility across major browsers.
  + Example:
  + for (const browserType of ['chromium', 'firefox', 'webkit']) {
  + const browser = await playwright[browserType].launch();
  + const page = await browser.newPage();
  + await page.goto('https://example.com');
  + console.log(`Testing on: ${browserType}`);
  + await browser.close();
  + }
* **Speaker Notes:**
  + Explain that Playwright’s browser engines are maintained by the same teams that build the actual browsers, ensuring fidelity.
  + Point out the simplicity of running the same test on multiple browsers using projects in the config file.
* **Demo:**
  + Run a test on all supported browsers and compare behaviours.

**Slide 6: Summary and Q&A**

* **Title:** Key Takeaways
* **Content:**
  + **Async/Await:** Simplifies handling asynchronous operations.
  + **Browser Contexts:** Efficient, isolated test sessions.
  + **Playwright Config:** Centralised test management.
  + **Multi-Browser Testing:** Ensures cross-browser compatibility.
* **Speaker Notes:**
  + Recap the covered topics and their significance.
  + Invite questions from participants.

**Slide 7: Hands-On Exercise**

* **Title:** Practical Exercise
* **Content:**
  + Task: Write a test that:
    - Uses async/await to navigate to a site and check the title.
    - Creates two browser contexts and demonstrates session isolation.
    - Configures and runs tests on multiple browsers.
* **Speaker Notes:**
  + Guide participants through solving the task.
  + Provide support as they encounter issues.

**Slide 1: Introduction to Playwright Core Concepts**

**Points to Talk:**

1. **Core Components of Playwright**:
   * Playwright is built to handle modern web automation needs like dynamic page interactions, multi-tab workflows, and parallel testing.
   * Core features include:
     + **Async and Await**: Ensures smooth handling of asynchronous tasks.
     + **Browser Contexts**: Simulate isolated browser sessions.
     + **Page Fixtures**: Represent browser tabs for user interactions.
     + **Multi-Browser Support**: Enables cross-browser compatibility testing.
2. **Challenges Addressed by Playwright**:
   * **Flaky Tests**: Caused by timing issues or unstable elements. Playwright’s built-in retries and auto-waiting mechanisms help mitigate these.
   * **Multi-User Scenarios**: Browser contexts allow isolated sessions, simulating different users without interference.
   * **Cross-Browser Testing**: Ensures the application behaves consistently on all major browsers.
3. **Why Learn These Concepts?**
   * These core concepts form the foundation of robust and scalable Playwright test automation. They simplify complex testing scenarios and improve test efficiency.

**Slide 2: Async and Await in Playwright**

**Points to Talk:**

1. **Why Async/Await?**:
   * Most Playwright APIs are asynchronous because web operations like loading pages or clicking elements take time.
   * async and await make the code easier to read and follow compared to using Promises directly.
   * Without await, operations can execute out of order, leading to errors or incomplete tests.
2. **Example Code**:

typescript

CopyEdit

const { chromium } = require('playwright');

(async () => {

const browser = await chromium.launch();

const page = await browser.newPage();

await page.goto('https://example.com');

console.log(await page.title()); // Logs the title of the page

await browser.close();

})();

**Walkthrough of the Code:**

* **Line 1**: Import Playwright's Chromium library.
* **Line 3**: Start an asynchronous function.
* **Line 4**: Launch a Chromium browser instance using await.
* **Line 5**: Open a new page in the browser.
* **Line 6**: Navigate to "<https://example.com>" using await page.goto(). Explain how await ensures the navigation completes before moving forward.
* **Line 7**: Log the page title using await page.title(). Without await, the title might not be available yet.
* **Line 8**: Close the browser.

**Demo Commentary:**

* Run the code to show the page loading and the title being printed.
* Remove await in one step to show errors or incomplete operations, reinforcing the importance of await.

**Slide 3: Browser Contexts and Page Fixtures**

**Points to Talk:**

1. **Browser Contexts**:
   * Browser contexts are lightweight, independent browser sessions within a single browser instance.
   * They isolate cookies, cache, and local storage, making them ideal for multi-user workflows.
   * More efficient than launching multiple browser instances.
2. **Page Fixtures**:
   * Pages are individual browser tabs or iframes.
   * Most interactions (e.g., clicks, form submissions) happen on pages.
3. **Example Code**:

typescript

CopyEdit

const browser = await chromium.launch();

// Create two independent browser contexts

const user1Context = await browser.newContext();

const user2Context = await browser.newContext();

// Open pages in the contexts

const user1Page = await user1Context.newPage();

const user2Page = await user2Context.newPage();

await user1Page.goto('https://example.com/login');

await user2Page.goto('https://example.com/login');

console.log(await user1Page.title()); // Logs title for user1

console.log(await user2Page.title()); // Logs title for user2

await browser.close();

**Walkthrough of the Code:**

* **Line 1**: Launch a browser instance.
* **Line 3**: Create two independent browser contexts (user1Context and user2Context).
* **Line 6**: Open new pages (tabs) in each context.
* **Lines 8-9**: Navigate to the login page in both contexts.
* **Lines 11-12**: Log the page title for each context to show isolation.
* **Line 14**: Close the browser.

**Demo Commentary:**

* Run the code and demonstrate:
  + Both contexts navigating to the same site but maintaining separate sessions.
  + How changes in one context (e.g., login details) don’t affect the other.

**Slide 4: Playwright Configuration File**

**Points to Talk:**

1. **Purpose**:
   * The configuration file centralises test settings, making tests easier to manage and run.
2. **Example Config File**:

typescript

CopyEdit

import { defineConfig } from '@playwright/test';

export default defineConfig({

timeout: 30000,

use: {

browserName: 'chromium',

headless: true,

},

projects: [

{ name: 'Desktop Chrome', use: { browserName: 'chromium' } },

{ name: 'Mobile Safari', use: { browserName: 'webkit', viewport: { width: 375, height: 667 } } },

],

});

**Walkthrough of the Code:**

* **timeout**: Sets a default timeout for tests (30 seconds).
* **use**: Specifies browser and test options. In this case, Chromium in headless mode.
* **projects**: Defines multiple configurations (e.g., Desktop Chrome and Mobile Safari).

**Demo Commentary:**

* Create a config file in an example project.
* Modify options (e.g., change timeout, switch to headless mode) and explain their impact.
* Run tests with different configurations using the --project flag.

**Slide 5: Multi-Browser Support**

**Points to Talk:**

1. **Why Multi-Browser Testing?**:
   * Different browsers have unique rendering engines, which can affect the application’s behaviour.
   * Ensures your app works consistently across Chromium, Firefox, and WebKit.
2. **Example Code**:

typescript

CopyEdit

const playwright = require('playwright');

(async () => {

for (const browserType of ['chromium', 'firefox', 'webkit']) {

const browser = await playwright[browserType].launch();

const page = await browser.newPage();

await page.goto('https://example.com');

console.log(`Testing on: ${browserType} - ${await page.title()}`);

await browser.close();

}

})();

**Walkthrough of the Code:**

* **Line 2**: Use a loop to iterate over Playwright’s supported browsers.
* **Line 4**: Launch each browser type (chromium, firefox, and webkit).
* **Line 5**: Open a new page in the browser.
* **Line 6**: Navigate to the website.
* **Line 7**: Log the browser type and page title.
* **Line 8**: Close the browser after the test.

**Demo Commentary:**

* Run the code to show how the same test works across multiple browsers.
* Compare behaviours and discuss how Playwright ensures fidelity by using the browser engines directly.

**Slide 6: Summary and Q&A**

**Points to Recap:**

1. **Async/Await**: Simplifies asynchronous operations.
2. **Browser Contexts**: Efficiently manage isolated test sessions.
3. **Playwright Config**: Centralised test configuration.
4. **Multi-Browser Testing**: Ensures compatibility across browsers.

**Interaction:**

* Invite questions and provide clarifications.

**Slide 7: Hands-On Exercise**

**Task:**

1. Write a test that:
   * Uses async/await to navigate to a website and verify its title.
   * Creates two browser contexts for session isolation.
   * Configures Playwright to run on multiple browsers.

**Support Tips:**

* Help participants debug errors.
* Encourage experimentation with config options (e.g., enabling/disabling headless mode).

This content provides you with everything needed to clearly explain the topics and engage the audience with hands-on examples. Let me know if you'd like adjustments or more depth on any part!

**You said:**

Line 7: Log the page title using await page.title(). Without await, the title might not be available yet. - explain the requirement of await here, how will the code above and below work with and without, please explain the errors or the impact clearly

**ChatGPT said:**

ChatGPT

**Explanation of the Requirement for await**

In JavaScript, many operations like navigating to a webpage or retrieving its title are asynchronous. These operations return a Promise, which represents a value that will eventually be available. Without await, the code moves on to the next line before the Promise resolves, potentially leading to incomplete or incorrect behaviour.

**Code with await**

typescript

CopyEdit

const { chromium } = require('playwright');

(async () => {

const browser = await chromium.launch();

const page = await browser.newPage();

await page.goto('https://example.com');

console.log(await page.title());

await browser.close();

})();

**Step-by-Step Execution with await:**

1. **Launch Browser**: await chromium.launch() ensures the browser instance is fully launched before moving to the next step.
2. **Open a Page**: await browser.newPage() waits for a new tab to open.
3. **Navigate to URL**: await page.goto('https://example.com') waits for the page to load completely.
4. **Retrieve Title**: await page.title() waits until the page's title is fully available before logging it.
5. **Close Browser**: await browser.close() ensures all operations are finished before the browser closes.

**Expected Output:**

* Logs the page title (e.g., "Example Domain").
* Everything works as intended, as each asynchronous operation is properly awaited.

**Code Without await**

typescript

CopyEdit

const { chromium } = require('playwright');

(async () => {

const browser = chromium.launch(); // No await

const page = browser.newPage(); // No await

page.goto('https://example.com'); // No await

console.log(page.title()); // No await

browser.close(); // No await

})();

**What Happens Without await:**

1. **browser.launch()**:
   * Returns a Promise instead of a browser instance.
   * Since await is missing, the browser instance isn't ready, leading to issues in subsequent steps.
2. **browser.newPage()**:
   * Tries to create a new page, but the browser variable is still a Promise rather than a resolved browser object.
   * This results in an error: **“TypeError: browser.newPage is not a function”**.
3. **page.goto()**:
   * Even if the code reaches this step, the navigation request is sent before the browser is ready, resulting in incomplete or failed navigation.
4. **page.title()**:
   * Instead of the actual title, this line logs a Promise, e.g., Promise { <pending> }.
   * Since the title retrieval depends on the navigation completing, the operation fails or returns incorrect data.
5. **browser.close()**:
   * Fails because the browser never fully launched or completed operations.

**Comparison of Behaviour**

| **With await** | **Without await** |
| --- | --- |
| Operations execute sequentially. | Operations execute in parallel and out of order. |
| Properly waits for the browser to launch. | Browser may not fully launch before operations. |
| Ensures the page is fully loaded before interacting. | Page interactions fail or return incomplete data. |
| Logs the correct title. | Logs a Promise object or throws an error. |
| Browser closes after all tasks finish. | Browser may close prematurely, leading to resource leaks. |

**Demo Explanation**

1. **With await**:
   * Run the code and show the correct title being logged.
   * Emphasise how await ensures each step is complete before moving to the next one.
2. **Without await**:
   * Run the code without await and highlight the errors:
     + "TypeError: browser.newPage is not a function."
     + "UnhandledPromiseRejectionWarning."
   * Alternatively, demonstrate logging Promise { <pending> } instead of the title.

**Key Takeaways for await:**

1. **Sequential Execution**: Ensures that asynchronous operations occur in the intended order.
2. **Error Prevention**: Avoids issues caused by unresolved Promise objects.
3. **Data Accuracy**: Ensures data (e.g., page title) is fully available before being accessed.
4. **Code Readability**: Makes asynchronous code easier to write and debug, compared to using .then() or callbacks.