



THE ULTIMATE GUIDE TO PLAYWRIGHT (JAVA) INTERVIEW SUCCESS



# CRACK PLAYWRIGHT (JAVA) INTERVIEWS

## SAMPLE BOOK

**100+**  
QUESTIONS

**1800+** Interview Questions, Automation Scenarios,  
Mobile Testing, Framework Design, and  
Architect-Level **Playwright Java** Q&A



**1800+**  
QUESTIONS



**250+**  
AUTOMATION SCENARIOS



**280+**  
MOBILE



**150+**  
FRAMEWORK/TEST DESIGN



**250+**  
ARCHITECT

```
@Test
public void testLogin() {
    page.navigate("https://app.com");
    page.locator("input[\"!\"{\"admin123\"}");
    page.locator("password").fill("admin123");
    page.locator("button[type='submit']").click();
    page.locator(page.locator("#dashboard").isVisible)
    assertTrue(page.locator("#dashboard").isVisible)
}
```



**R. RAJAMANTHIRAM**



[www.crackplaywrightinterviews.com](http://www.crackplaywrightinterviews.com)



---

# Crack Playwright (Java) Interviews

**1800+ Interview Questions, Automation Scenarios, Mobile Testing, Framework Design, and Architect-Level Playwright Java Q&A**

Author: R.Rajamanthiram

Website: [www.crackplaywrightinterviews.com](http://www.crackplaywrightinterviews.com)

LinkedIn: <https://www.linkedin.com/in/rajamanthiram/>

---

---

## Ebook Highlights

- 1800+ Playwright Java interview questions and answers
  - 100% structured coverage of major Playwright Java interview concepts
  - 250+ real automation scenario questions
  - 280+ mobile automation and emulation-focused questions
  - 150+ framework design and test design questions
  - 250+ automation architect, lead, and strategy-level questions
  - Structured into 5 focused books for foundation, scenarios, framework design, architect thinking, and advanced topics
  - Covers beginner, project-level, senior, and architect-level interviews
  - Includes Java-focused examples, common mistakes, and practical explanations
-

---

## Objective

Master every important Playwright Java concept and interview question through structured Q&A, real project scenarios, mobile automation, framework design, debugging, and architect-level thinking.

This ebook is designed to help automation engineers prepare for Playwright Java interviews with confidence by learning not only answers, but also the reasoning, mistakes, trade-offs, and real project situations behind each question.

---

---

## About This Ebook

Crack Playwright (Java) Interviews is a structured interview preparation ebook for QA engineers, automation engineers, SDETs, test leads, and automation architects who want to master Playwright Java.

The book is organized into five focused books:

1. Book 1: Playwright

Covers the core Playwright Java concepts, setup, browser execution, browser contexts, pages, locators, actions, assertions, frames, network, screenshots, tracing, and extensibility foundations.

2. Book 2: Automation Scenarios & Debugging

Focuses on real automation scenarios, practical project workflows, UI/API situations, troubleshooting patterns, debugging evidence, and failure investigation.

3. Book 3: Framework & Test Design

Covers test design, Page Object Model, framework design, maintainability, reusable components, and parallel-safe test data strategy.

4. Book 4: Architect/Leadership

Builds senior, lead, and architect-level thinking around automation strategy, governance, reliability, ROI, release readiness, ownership, and leadership decisions.

5. Book 5: Advanced Playwright Topics

Covers advanced Playwright topics such as Selenium Grid, snapshot testing, accessibility, cross-browser and visual testing, emulation, CI, AI, touch events, and WebView2.

---

---

## Who This Book Is For

This ebook is useful for:

- Freshers preparing for Playwright Java automation interviews
  - Manual testers moving into automation
  - Selenium Java automation engineers moving to Playwright Java
  - QA engineers preparing for SDET roles
  - Automation engineers working on modern web applications
  - Test leads preparing for framework and strategy interviews
  - Automation architects preparing for senior-level discussions
  - Trainers and mentors teaching Playwright Java interview preparation
-

---

# How to Use This Book

Use this ebook in five passes:

## **Pass 1: Learn Playwright Java Foundations**

Start with Book 1 - Playwright. Understand the core Playwright Java concepts, setup, browser execution, browser contexts, pages, locators, assertions, actions, downloads, uploads, frames, network, screenshots, traces, and related fundamentals.

## **Pass 2: Practice Automation Scenarios and Debugging**

Move to Book 2 - Automation Scenarios & Debugging. Read each scenario as if an interviewer is asking how you would solve it in a real project. Focus on the reasoning, test flow, validation strategy, failure evidence, and debugging approach.

## **Pass 3: Build Framework and Test Design Thinking**

Study Book 3 - Framework & Test Design. Focus on how to design maintainable tests, Page Objects, reusable components, test data strategy, and framework structure for real automation projects.

## **Pass 4: Prepare for Architect and Leadership Questions**

Use Book 4 - Architect/Leadership to prepare for senior-level discussions. Practice explaining automation strategy, governance, ROI, release readiness, reliability, ownership, quality gates, and leadership trade-offs.

## **Pass 5: Revise Advanced Playwright Topics**

Complete Book 5 - Advanced Playwright Topics. Revise advanced areas such as Selenium Grid integration, snapshot testing, accessibility, cross-browser testing, visual testing, emulation, CI, AI, touch events, and WebView2.

---

## **Final Interview Strategy**

Before the interview:

1. Revise Book 1 for conceptual clarity.
2. Practice Book 2 to answer real project scenario and debugging questions confidently.



- 
3. Study Book 3 to explain framework design, Page Object Model, test design, and test data strategy.
  4. Use Book 4 to sound senior, practical, architecture-aware, and leadership-oriented.
  5. Revise Book 5 for advanced Playwright Java topics and modern automation expectations.
  6. Focus on Java examples, not only theory.
  7. Always explain the why, not only the API name.
  8. Mention common mistakes and how you avoid them.
-

---

## Author Profile

R. Rajamanthiram is a Quality Engineering leader and Automation Test Architect with 22+ years of experience in software quality engineering, test automation, framework design, and automation strategy.

He has worked extensively on automation architecture, enterprise test frameworks, UI automation, API testing, CI/CD quality integration, and interview preparation for QA and automation engineers.

He has taken hundreds of interviews and has worked in premium product companies, bringing practical hiring and real-project automation experience into this ebook.

LinkedIn: <https://www.linkedin.com/in/rajamanthiram/>

Website: [www.crackplaywrightinterviews.com](http://www.crackplaywrightinterviews.com)

---

---

# Book Summary

Metric	Count
Total Books	5
Total Questions	1876
Total Topic Sections	50
Unique Topic Labels	50
Book 1 Questions	605
Book 2 Questions	349
Book 3 Questions	152
Book 4 Questions	299
Book 5 Questions	471
Book 1 Topics	31
Book 2 Topics	2
Book 3 Topics	4
Book 4 Topics	2
Book 5 Topics	11

---

This sample book contains 100+ Playwright interview questions.

For the complete book with 1800+ questions, visit:

**[www.crackplaywrightinterviews.com](http://www.crackplaywrightinterviews.com)**

---

# Table of Contents

## **Book 1 - Playwright 14**

1. Introduction - page 15
2. Get Started - page 24
3. Installation & Browser - page 32
4. BrowserContext - page 41
5. Page - page 48
6. Autowait - page 54
7. Assertions - page 61
8. Locators - page 71
9. Actions - page 79
10. Authentication - page 85
11. Download - page 91
12. File Upload - page 95
13. Handles - page 99
14. Evaluating Javascript - page 103
15. Navigation - page 107
16. Dialogs - page 111
17. Events - page 115
18. Test Isolation - page 119
19. Frames - page 124
20. Mock APIs - page 128
21. Network - page 133
22. Timeout - page 137
23. Clock - page 141
24. Screenshots - page 145
25. Videos - page 149
26. Multithreading - page 153
27. Flakiness - page 158
28. Performance - page 163
29. CodeGen - page 168
30. Trace viewer - page 173
31. Extensibility - page 177

## **Book 2 - Automation Scenarios & Debugging 181**

1. Automation scenarios - page 182
2. Debugging - page 223

## **Book 3 - Framework & Test Design 236**

1. Test Design - page 237
2. Page Object Model - page 249
3. Framework Design - page 257

---

## **Book 4 - Architect/Leadership 272**

1. Architect - page 273
2. Test Strategy - page 303

## **Book 5 - Advanced Playwright Topics 312**

1. Selenium Grid - page 313
2. Snapshot testing - page 319
3. Accessibility Testing - page 327
4. Cross browser testing - page 334
5. Visual testing - page 340
6. Playwright Vs Selenium - page 344
7. Emulation (Mobile) - page 350
8. CI - page 357
9. AI - page 365

---

# **Book 1 - Playwright**

---

# **1. Introduction**



---

## **Part I - Core Questions**

## Question 1.1

### What is Playwright, and why is it used in test automation?

#### Interview-Style Answer

Playwright is a modern end-to-end testing and browser automation tool used to test modern web applications reliably.

It is used because modern applications are dynamic and need stable automation. Playwright makes tests more reliable with features like auto-waiting, web-first assertions, browser context isolation, tracing which help reduce flaky test failures.

Playwright provides one API to test across different browsers, platforms, and languages. It supports Chromium, Firefox, WebKit, Windows, Linux, macOS, and languages like Java, JavaScript, TypeScript, Python, and .NET.

It also supports important real-project testing needs such as mobile web emulation, multiple tabs, multiple users, iframes, Shadow DOM, API testing, mock APIs, and network interception.

For faster test creation and debugging, Playwright provides useful tools like Codegen, Playwright Inspector, and Trace Viewer. Codegen helps generate test scripts by recording user actions, Inspector helps debug step by step, and Trace Viewer helps analyze failures with screenshots, DOM snapshots, actions, and network details.

#### Detailed Explanation

Playwright is useful because modern web applications are not simple static pages. They are usually dynamic, asynchronous, and heavily dependent on JavaScript, APIs, animations, validations, and real-time UI updates. Because of this, automation tools must wait correctly, interact like real users, and provide strong debugging support. Playwright solves these needs by providing reliable browser automation with built-in waiting, strong assertions, and powerful test execution features.

One major reason Playwright is reliable is auto-waiting. Before performing actions like `click()`, `fill()`, or `selectOption()`, Playwright automatically waits until the element is ready for action. This reduces the need for hard waits like `Thread.sleep()`, which often make tests slow and flaky. Its web-first assertions also retry validations until

---

the expected condition is met, making checks more stable for dynamic pages.

Playwright also gives better test isolation using browser contexts. Each test can run in a fresh browser context, similar to a new browser profile, with separate cookies, local storage, sessions, and permissions. This prevents one test from affecting another. At the same time, authentication state can be saved and reused, so teams can avoid repeated login steps while still keeping tests isolated.

It is also suitable for real project scenarios because it supports cross-browser testing, mobile web emulation, multiple tabs, multiple users, iframes, Shadow DOM, API testing, mock APIs, and network interception. This means the same framework can validate both UI behavior and backend/API behavior, and it can also mock responses to test success, failure, empty data, or server-error scenarios.

For faster development and debugging, Playwright provides tools like Codegen, Playwright Inspector, and Trace Viewer. Codegen helps create test scripts by recording user actions. Inspector helps debug tests step by step. Trace Viewer helps analyze failures using screenshots, DOM snapshots, actions, source code, and network details.

## **Common Mistake**

treating Playwright only as a browser automation tool. In real projects, it is a complete end-to-end testing framework for reliable, isolated, cross-browser, API-enabled, mock-supported, and debuggable automation.

---

## Question 1.2

**Why is Playwright considered suitable for testing modern web applications like React, Angular, and Vue apps?**

### Interview-Style Answer

Playwright is suitable for modern web applications because it is designed to handle dynamic UI behavior, asynchronous DOM updates, auto-waiting, reliable locators, and stable assertions. This makes it effective for testing applications built with frameworks like React, Angular, and Vue, where elements can appear, disappear, or change state without a full page reload.

### Detailed Explanation

Modern web applications are usually dynamic. In frameworks like React, Angular, and Vue, the page does not always reload fully after every action. Instead, the application updates only parts of the DOM.

For example:

- A button may become enabled after an API response.
- A list may update after filtering.
- A modal may appear after a user action.
- A component may re-render after state change.
- A loading spinner may disappear after data is loaded.

Traditional automation tools may struggle with these asynchronous changes because they often try to interact with elements before the UI is ready.

Playwright helps solve this problem through auto-waiting. Before performing actions, Playwright waits for the element to be ready for interaction. It checks conditions such as:

- Element is visible.
- Element is stable.
- Element is enabled.
- Element is ready to receive user action.

This reduces the need for hard waits like `Thread.sleep()` and makes tests more reliable.

---

Playwright also provides web-first assertions. These assertions automatically retry until the expected condition becomes true or the timeout is reached. This is very useful in SPAs because UI updates may happen after a short delay.

Another major advantage is its locator strategy. Playwright encourages user-facing locators such as:

- `getByRole()`
- `getByText()`
- `getByLabel()`

These locators are based on how users see and interact with the application, rather than depending heavily on fragile DOM structures. This is especially useful in React, Angular, and Vue applications where DOM structure may change because of component re-rendering.

### Example Scenario

Suppose a React application loads a “Submit” button only after form validation is complete.

With Playwright, we can write:

```
page.getByRole(AriaRole.BUTTON,  
  new Page.GetByRoleOptions().setName("Submit")).click();
```

Playwright will wait until the button is ready for interaction before clicking it.

## Common Mistake

A common mistake is treating modern web apps like static HTML pages.

In SPAs, the UI changes dynamically. So, using fixed waits, fragile XPath, or immediate assertions can create flaky tests. Playwright is preferred because it works naturally with dynamic UI behavior through auto-waiting, retrying assertions, and reliable locators.

---

## Question 1.11

### What are the top 10 reasons to use Playwright for modern web automation?

#### Interview-Style Answer

The top 10 reasons to use Playwright are:

- It handles modern web applications smoothly.
- It provides auto-waiting for stable execution.
- It supports smart web-first assertions.
- It offers reliable user-centric locators.
- It supports real mobile device emulation.
- It provides strong test isolation using browser contexts.
- It handles multiple tabs and windows efficiently.
- It supports reusable authentication.
- It provides advanced network interception and API mocking.
- It supports both UI and API testing in the same framework.

#### Detailed Explanation

Playwright is a strong choice for modern web automation because it is designed for today's dynamic web applications.

Modern applications built using frameworks like React, Angular, and Vue update the page dynamically without full page reloads. Elements may appear, disappear, or change state asynchronously. Playwright handles these changes well because it waits for elements to be ready before performing actions. This makes it suitable for testing SPAs and highly interactive web applications.

The first major reason is auto-waiting. Playwright automatically waits for elements to become:

- Visible
- Stable
- Enabled
- Ready for interaction

Because of this, we do not need to depend heavily on `Thread.sleep()` or unnecessary manual waits. This directly improves test stability.

---

The second major advantage is web-first assertions. Playwright assertions automatically retry until the expected condition becomes true or the timeout is reached. This is very useful for dynamic UIs where the expected text, element, or state may take a short time to appear.

Another important reason is its robust locator strategy. Playwright promotes user-facing locators such as:

- `getByRole()`
- `getByText()`
- `getByLabel()`

These locators are closer to how real users interact with the application, so tests become more readable and maintainable.

Playwright also supports real mobile device emulation. We can emulate devices like iPhone, Pixel, and iPad, along with screen size and touch behavior. This helps test mobile responsiveness without always needing physical devices.

A very important feature is test isolation using browser contexts. Each test can run in a separate browser context with its own:

- Cookies
- Sessions
- Local storage
- Clean browser state

This prevents one test from affecting another test and makes execution more reliable.

Playwright also makes it easier to handle real-world browser scenarios such as:

- Multiple tabs
- New windows
- Popups
- Payment gateway redirects
- Report windows

Another useful reason is reusable authentication. We can log in once, save the authenticated state, and reuse it across multiple tests. This saves execution time, especially in large regression suites.

Playwright also provides network interception and API mocking. This allows us to:

- Intercept network requests

- 
- Mock API responses
  - Simulate backend failures
  - Test UI even when backend services are unstable

Finally, Playwright supports both UI testing and API testing. This makes it useful for end-to-end testing because we can prepare backend data, test the UI flow, and validate server-side results within the same automation framework.

## Common Mistake

A common mistake is thinking Playwright is only another browser automation tool.

In reality, Playwright is useful because it combines many modern automation needs in one framework:

- Stable UI automation
- Cross-browser testing
- Mobile emulation
- Test isolation
- Network control
- Debugging support
- API testing

This makes it more suitable for modern end-to-end automation than tools that focus only on simple browser actions.

---



---

## 2. Get Started

---

## **Part I - Core Questions**

## Question 2.1

### How would you get started with Playwright Java in a new automation project?

#### Interview-Style Answer

To get started with Playwright Java in a new automation project, I would first create a Maven or Gradle project, add the Playwright Java dependency, and write a simple test to launch a browser, open a page, perform a basic action, and validate the result.

After confirming that the basic setup works, I would introduce a test framework such as JUnit or TestNG. Then I would organize the project with proper setup and teardown methods, browser/context management, reusable page objects, configuration handling, assertions, and reporting.

In real projects, I would also make sure the framework supports browser selection, environment selection, headless/headed execution, CI execution, screenshots, traces, and proper cleanup of Playwright resources.

#### Detailed Explanation

Getting started with Playwright Java should be done step by step. The first goal is to verify that Playwright is installed correctly and can control a browser successfully. So I would begin with a simple Maven or Gradle project and add the Playwright Java dependency.

A basic starting flow would be:

1. Create a Maven or Gradle project.
2. Add the Playwright Java dependency.
3. Install or ensure Playwright browser binaries are available.
4. Create a simple Java test.
5. Create a Playwright instance.
6. Launch Chromium, Firefox, or WebKit.
7. Create a BrowserContext and Page.
8. Navigate to the application.
9. Perform a basic action or assertion.
10. Close resources properly.

The basic Playwright object flow is:

```
Playwright → Browser → BrowserContext → Page → Locator → Actions /  
Assertions
```

Example Maven dependency:

```
<dependency>
  <groupId>com.microsoft.playwright</groupId>
  <artifactId>playwright</artifactId>
  <version>${playwright.version}</version>
</dependency>
```

Using a property for the version is better in real projects because it keeps dependency upgrades easier.

Example Java code:

```
import com.microsoft.playwright.*;
import static
    com.microsoft.playwright.assertions.PlaywrightAssertions.assertThat;

public class FirstPlaywrightTest {
    public static void main(String[] args) {
        try {Playwright playwright = Playwright.create(); {
            Browser browser = playwright.chromium().launch(
                new BrowserType.LaunchOptions().setHeadless(false)
            );

            BrowserContext context = browser.newContext();
            Page page = context.newPage();

            page.navigate("https://example.com");
            assertThat(page.locator("h1")).hasText("Example Domain");

            context.close();
            browser.close();
        }
    }
}
```

Once this basic script works, I would move from a simple `main` method to a proper test framework like JUnit or TestNG. The framework should manage setup and teardown, create a fresh `BrowserContext` for each test, close resources after execution, and support command-line execution using Maven or Gradle.

For a real automation project, I would gradually add:

- Base test setup
- Browser and environment configuration
- Page Object Model
- Reusable locators and actions
- Playwright assertions
- Test data handling
- Screenshots on failure
- Trace and video capture
- CI/CD execution support
- Reporting integration

This approach keeps the project simple at the beginning and scalable later. The important point is not just to launch a browser, but to build the foundation correctly so that tests are reliable, maintainable, and easy to run in local and CI environments.

---

## Common Mistake

starting with a complex framework before verifying the basic Playwright Java setup. In real projects, first confirm that Playwright can launch the browser, open the application, perform actions, validate results, and close resources properly; then build the framework layer step by step.

---

## Question 2.6

### How do you create a `BrowserContext` and `Page` in Playwright Java?

#### Interview-Style Answer

Create a browser context using `browser.newContext()`, then create a page using `context.newPage()`. The context gives test isolation, and the page represents the browser tab used for automation.

In Playwright Java, this matters because a `BrowserContext` provides an isolated browser session with its own cookies, storage, permissions, and session data, while a `Page` represents the tab where navigation, actions, and validations are performed.

#### Detailed Explanation

A `BrowserContext` acts like an isolated browser profile. It has its own cookies, local storage, session storage, permissions, and other browser state. This helps keep tests independent and prevents one test from affecting another.

A `Page` is created inside a browser context. It represents a browser tab or window and is used to perform actions such as navigation, clicking, typing, uploading files, and validating UI behavior.

Example:

```
Browser browser = playwright.chromium().launch();
BrowserContext context = browser.newContext();
Page page = context.newPage();
page.navigate("https://example.com");
context.close();
browser.close();
```

For reliable test automation frameworks, each test should usually get a fresh browser context. This avoids state leakage between tests and makes execution more predictable, especially in CI or parallel runs.

#### Common Mistake

using one shared page or browser context for many unrelated tests. This can cause test dependency, session leakage, unstable failures, and incorrect results.

## Question 2.12

### Why is it important to close Playwright, browser, and context resources properly?

#### Interview-Style Answer

It is important to close Playwright, browser, and context resources properly because unclosed resources can leave browser processes running, consume memory, lock files, slow down execution, and make CI pipelines unstable.

In Playwright Java, tests communicate with real browser processes. If these resources are not closed after execution, they may remain active in the background and affect later tests. Proper cleanup keeps the framework stable, prevents memory leaks, and ensures predictable test execution.

#### Detailed Explanation

Playwright Java creates and manages real automation resources during test execution. `Playwright` starts the Playwright engine, `Browser` launches the browser process, `BrowserContext` creates an isolated browser profile, and `Page` opens a browser tab.

If these resources are not closed correctly, the browser process or context may continue running even after the test has finished. Over time, especially in large test suites or CI/CD pipelines, this can lead to memory usage issues, hanging test runs, locked files, port/resource conflicts, and unstable execution.

A good practice is to use `try-with-resources` for `Playwright` and explicitly close the `BrowserContext` and `Browser`.

```
try (Playwright playwright = Playwright.create()) {
    Browser browser = playwright.chromium().launch();
    BrowserContext context = browser.newContext();
    Page page = context.newPage();

    page.navigate("https://example.com");

    context.close();
    browser.close();
}
```

In real frameworks, cleanup is usually handled using JUnit or TestNG lifecycle methods such as `@AfterEach`, `@AfterMethod`, `@AfterAll`, or `@AfterSuite`. This ensures that resources are closed even when a test

---

fails.

### **Common Mistake**

closing only the page and forgetting to close the browser context, browser, or Playwright instance. In real projects, proper cleanup is part of framework stability, not just code hygiene.

---



---

## **3. Installation & Browser**

---

## **Part I - Core Questions**

### Question 3.8

**What is the difference between `install`, `install-deps`, and `install --with-deps`?**

### Interview-Style Answer

In Playwright Java, these commands are related to browser setup, but they solve different setup problems.

```
install          -> installs Playwright browser binaries
install-deps     -> installs required OS/system dependencies
install --with-deps -> installs both browser binaries and OS/system
                    dependencies
```

`install` is mainly for downloading Playwright-supported browser binaries such as Chromium, Firefox, and WebKit. `install-deps` is mainly for Linux, Docker, or CI machines where required system libraries may be missing. `install --with-deps` combines both and is commonly useful for clean CI or container setup.

### Detailed Explanation

Playwright tests run against real browser engines. For those browsers to run correctly, two things may be needed:

1. Playwright-managed browser binaries
2. Operating system libraries required by those browsers

The `install` command downloads browser binaries:

```
mvn exec:java -e -D exec.mainClass=com.microsoft.playwright.CLI -D
    exec.args="install"
```

This is useful during local setup, fresh machine setup, or after upgrading the Playwright version. Without the browser binary, Playwright may not be able to launch Chromium, Firefox, or WebKit.

Example test that needs the browser binary:

```
try (Playwright playwright = Playwright.create()) {
    Browser browser = playwright.chromium().launch();
    Page page = browser.newPage();

    page.navigate("https://example.com");

    PlaywrightAssertions.assertThat(page)
        .hasTitle(Pattern.compile("Example"));

    browser.close();
}
```

---

The `install-deps` command installs OS-level dependencies:

```
mvn exec:java -e -D exec.mainClass=com.microsoft.playwright.CLI -D
  exec.args="install-deps"
```

These dependencies include system libraries needed for graphics, fonts, audio, sandboxing, rendering, and shared Linux libraries. This matters most in minimal environments such as Docker images, Linux build agents, and CI runners.

You can also install dependencies for a specific browser:

```
mvn exec:java -e -D exec.mainClass=com.microsoft.playwright.CLI -D
  exec.args="install-deps chromium"
```

This is useful when the framework runs only Chromium and does not need Firefox or WebKit dependencies.

The `install --with-deps` command installs both browser binaries and system dependencies:

```
mvn exec:java -e -D exec.mainClass=com.microsoft.playwright.CLI -D
  exec.args="install --with-deps chromium"
```

This is often the safest option for clean CI or Docker setup because it reduces the chance of installing the browser but forgetting the required OS packages.

Simple comparison:

```
install
- Installs browser binaries
- Useful for local setup and Playwright upgrades

install-deps
- Installs OS/system dependencies
- Useful for Linux, Docker, and CI environments

install --with-deps
- Installs browser binaries and OS dependencies together
- Useful for clean CI/Docker setup
```

A good Playwright Java framework should document these setup steps clearly in the README, Dockerfile, or CI pipeline. If tests fail with missing shared libraries, sandbox errors, browser launch failures, or dependency-related Linux errors, the setup may be missing `install-deps` or `install --with-deps`.

## Common Mistake

---

Running only `install` in a Linux or CI environment and assuming setup is complete. `install` downloads browser binaries, but it does not always guarantee that the operating system has all libraries required to run those browsers.

---

### Question 3.14

## Does Playwright install Google Chrome and Microsoft Edge by default?

### Interview-Style Answer

No. Playwright does not install branded Google Chrome or Microsoft Edge by default.

By default, Playwright installs and uses Playwright-managed browser builds such as Chromium, Firefox, and WebKit. Playwright-managed Chromium is not the same as branded Google Chrome or Microsoft Edge.

If a project needs to run tests on branded Chrome or Edge, those browsers must be available on the machine, and the test should launch them using the `channel` option.

### Detailed Explanation

Playwright can run tests using both Playwright-managed browsers and supported branded Chromium-based browser channels.

Default Chromium execution:

```
Browser browser = playwright.chromium().launch();
```

This launches Playwright-managed Chromium. It does not automatically launch Google Chrome installed on the machine.

To launch branded Google Chrome:

```
Browser browser = playwright.chromium().launch(  
    new BrowserType.LaunchOptions().setChannel("chrome")  
);
```

To launch branded Microsoft Edge:

```
Browser browser = playwright.chromium().launch(  
    new BrowserType.LaunchOptions().setChannel("msedge")  
);
```

Common branded browser channels include:

1. chrome
2. msedge
3. chrome-beta
4. msedge-beta
5. chrome-dev
6. msedge-dev
7. chrome-canary
8. msedge-canary

This is useful when the test strategy requires validation on the same branded browser used by customers, enterprise users, or production support teams. For example, an organization may run most regression tests on Playwright-managed Chromium but add a smaller smoke suite on branded Chrome Stable or Edge Stable.

The normal browser installation command:

```
mvn exec:java -e -D exec.mainClass=com.microsoft.playwright.CLI -D  
exec.args="install"
```

installs Playwright-managed browsers. It does not install branded Google Chrome or Microsoft Edge.

If the framework uses:

```
new BrowserType.LaunchOptions().setChannel("chrome")
```

or:

```
new BrowserType.LaunchOptions().setChannel("msedge")
```

then the CI agent, Docker image, or developer machine must already have that branded browser installed and accessible.

## Common Mistake

assuming `mvn exec:java -D exec.mainClass=com.microsoft.playwright.CLI -D exec.args="install"` installs Google Chrome or Microsoft Edge. It installs Playwright-managed browser binaries; branded Chrome or Edge must be installed separately before launching them with `setChannel("chrome")` or `setChannel("msedge")`.

### Question 3.17

**How do you launch Microsoft Edge using the `channel` option in Playwright Java?**

### Interview-Style Answer

In Playwright Java, Microsoft Edge is launched through `playwright.chromium()` because Edge is a Chromium-based browser. To use the installed Microsoft Edge browser instead of Playwright-managed Chromium, set the browser `channel` to `msedge`.

```
Browser browser = playwright.chromium().launch(  
    new BrowserType.LaunchOptions().setChannel("msedge")  
);
```

This is useful when the project needs Edge-specific validation, customer-browser certification, enterprise policy testing, or reproduction of an issue that appears only in branded Microsoft Edge.

### Detailed Explanation

Playwright-managed Chromium is used by default when we call:

```
Browser browser = playwright.chromium().launch();
```

But Microsoft Edge is a branded Chromium-based browser installed separately on the machine. To launch it, we still use the Chromium browser type, but specify the Edge channel.

Complete example:



```
import com.microsoft.playwright.*;
import java.util.regex.Pattern;

import static
    com.microsoft.playwright.assertions.PlaywrightAssertions.assertThat;

public class EdgeLaunchExample {
    public static void main(String[] args) {
        try (Playwright playwright = Playwright.create()) {
            Browser browser = playwright.chromium().launch(
                new BrowserType.LaunchOptions()
                    .setChannel("msedge")
                    .setHeadless(true)
            );

            Page page = browser.newPage();
            page.navigate("https://example.com");

            assertThat(page).hasTitle(Pattern.compile("Example"));

            browser.close();
        }
    }
}
```

Common Edge channels include:

msedge	-> Microsoft Edge Stable
msedge-beta	-> Microsoft Edge Beta
msedge-dev	-> Microsoft Edge Dev
msedge-canary	-> Microsoft Edge Canary

Use Edge channel testing when:

1. Customers officially use Microsoft Edge.
2. Release certification requires Edge validation.
3. A defect reproduces only in Edge.
4. Enterprise policies affect browser behavior.
5. The team needs to compare Edge behavior with Playwright-managed Chromium.

This should be configured carefully in CI. Playwright does not install branded Microsoft Edge by default, so the CI image or execution machine must already have Edge installed. The framework should also document which browser channels are supported locally and in CI to avoid browser launch failures.

## Common Mistake

Assuming Playwright automatically installs Microsoft Edge when using `setChannel("msedge")`. Playwright-managed Chromium is installed by Playwright, but branded Edge must already be available on the machine or CI image.

---

## 4. BrowserContext

---

## **Part I - Core Questions**

### Question 4.3

## Why should each test usually create a fresh `BrowserContext`?

### Interview-Style Answer

Each test should usually create a fresh `BrowserContext` because the context is the main unit of browser-session isolation in Playwright Java. It keeps cookies, local storage, session storage, permissions, cache, viewport settings, and authentication state separate between tests.

This prevents session leakage, order-dependent failures, false positives, and unstable parallel execution. A common framework pattern is to reuse `Playwright` and `Browser` where appropriate, but create a new `BrowserContext` and `Page` for each test.

### Detailed Explanation

A `BrowserContext` behaves like an independent browser profile. If multiple tests share the same context, they can accidentally share login state, cart data, permissions, cached data, local storage, feature flags, or application state.

A good lifecycle is:

```
@BeforeEach
void setup() {
    context = browser.newContext();
    page = context.newPage();
}

@AfterEach
void cleanup() {
    context.close();
}
```

This pattern gives each test a clean browser session. One test does not depend on whether another test logged in, accepted a cookie banner, changed a language setting, granted a permission, added an item to a cart, or stored data in local storage.

Benefits of a fresh context include:

- 
1. No login-state leakage.
  2. No cookie or token leakage.
  3. No local storage or session storage leakage.
  4. No permission leakage.
  5. No cart, order, or workflow-state leakage.
  6. Better parallel execution.
  7. Easier debugging because each test starts from a known state.
  8. Safer role-based testing.
  9. Cleaner artifact capture and teardown.
  10. More reliable CI execution.

For example, if one test logs in as an admin and another test expects a guest user, sharing the same context can produce a false result. The second test may pass or fail because it inherited the admin session instead of starting clean.

For different users, separate contexts are also required:

```
BrowserContext adminContext = browser.newContext();
Page adminPage = adminContext.newPage();

BrowserContext customerContext = browser.newContext();
Page customerPage = customerContext.newPage();
```

This keeps admin and customer cookies, tokens, permissions, and storage separate.

Fresh contexts are especially important in CI/CD and parallel execution because tests may run in a different order or at the same time. Without proper context isolation, failures can become difficult to reproduce because they depend on hidden browser state from another test.

## Common Mistake

Reusing one `BrowserContext` across many tests to save setup time. This can leak cookies, storage, permissions, and application state between tests, creating false positives, false failures, and order-dependent behavior that becomes worse in parallel CI execution.

---

## Question 4.9

### What are common mistakes with `BrowserContext` in Playwright Java?

#### Interview-Style Answer

Common mistakes with `BrowserContext` include reusing one context for all tests, using the same context for different users, sharing a static `Page`, not closing contexts after tests, misusing storage state, and configuring permissions, viewport, locale, or timezone at the wrong level.

In Playwright Java, `BrowserContext` is the main unit of browser-session isolation. A good framework should create a fresh context per test or per independent user session, configure context-level options before creating the page, and close the context during teardown to avoid state leakage and resource issues.

#### Detailed Explanation

`BrowserContext` controls browser-session state such as cookies, local storage, permissions, viewport, locale, timezone, geolocation, extra headers, downloads, and storage state. Many flaky tests come from misunderstanding this responsibility.

Common mistakes include:

1. Reusing one `BrowserContext` for all tests.
2. Sharing a static `Page` across tests.
3. Using the same context for admin, customer, and other users.
4. Not closing the context after each test.
5. Loading the wrong storage-state file for a role.
6. Reusing expired or stale authentication state.
7. Expecting cookies or local storage to automatically cross contexts.
8. Setting permissions after the page has already started the workflow.
9. Forgetting viewport, locale, timezone, and geolocation are context-level settings.
10. Running parallel tests without isolated users, data, downloads, and storage state.

A better framework pattern is:

```

@BeforeEach
void setUp() {
    context = browser.newContext(
        new Browser.NewContextOptions()
            .setViewportSize(1366, 768)
            .setLocale("en-IN")
            .setTimezoneId("Asia/Kolkata")
    );

    page = context.newPage();
}

@AfterEach
void tearDown() {
    context.close();
}

```

For multiple users, create separate contexts instead of multiple pages in the same context:

```

BrowserContext adminContext = browser.newContext(
    new Browser.NewContextOptions()
        .setStorageStatePath(Paths.get("auth/admin.json"))
);
Page adminPage = adminContext.newPage();

BrowserContext customerContext = browser.newContext(
    new Browser.NewContextOptions()
        .setStorageStatePath(Paths.get("auth/customer.json"))
);
Page customerPage = customerContext.newPage();

```

This keeps cookies, tokens, local storage, permissions, and session data separate. It is especially important for role-based tests, parallel execution, CI stability, and tests that depend on clean user state.

Storage state should also be used carefully. It can speed up authenticated tests, but each role should have its own storage-state file, and those files should not expose sensitive tokens in Git, reports, traces, screenshots, or logs. If authentication expires often, the framework should refresh storage state in a controlled way.

A clean approach is:

- Reuse Playwright and Browser where appropriate.
- Create a fresh `BrowserContext` per test.
- Use a separate `BrowserContext` per user.
- Configure context options before creating the `Page`.
- Keep storage-state files role-specific.
- Isolate test data, files, downloads, and users for parallel runs.
- Close the `BrowserContext` after each test.

## Common Mistake

Treating `BrowserContext` as just a browser container and then sharing it across tests or users. This causes hidden state leakage through

---

cookies, local storage, permissions, cached data, and storage-state files, leading to flaky, order-dependent, and unsafe parallel execution.

---



---

## 5. Page

---

## **Part I - Core Questions**

## Question 5.5

**What is the difference between `page.waitForPopup()` and `context.waitForPage()`?**

### Interview-Style Answer

`page.waitForPopup()` waits for a popup opened by a specific `Page`. It is best when a known action on the current page opens a new tab or window.

`context.waitForPage()` waits for any new `Page` created inside the `BrowserContext`. It is useful when the new page may be created from anywhere in the context, or when the opener page is not the main focus.

### Detailed Explanation

In Playwright Java, both methods return a `Page`, but they listen at different levels.

`page.waitForPopup()` is tied to a specific opener page. Use it when the current page action clearly opens the popup. This keeps the relationship between the original page and the popup easy to understand.

```
Page popup = page.waitForPopup(() -> {
    page.getByRole(AriaRole.LINK,
        new Page.GetByRoleOptions().setName("Open Report")
    ).click();
});

popup.waitForLoadState();

PlaywrightAssertions.assertThat(
    popup.getByText("Report Details")
).isVisible();
```

Here, the popup is expected to be opened by `page`. This is common for links with `target="_blank"`, report links, payment windows, invoice pages, or external help pages.

`context.waitForPage()` listens at the `BrowserContext` level. It captures a new page created anywhere inside that context.

---

```
Page docsPage = context.waitForPage(() => {
  page.getByText("Open documentation").click();
});

docsPage.waitForLoadState();

PlaywrightAssertions.assertThat(
  docsPage.getByText("Documentation")
).isVisible();
```

This is useful when the test cares about any newly created page in the context, or when the source of the new page is less direct. For example, a popup may be triggered indirectly by application code, another page, a redirect flow, or a shared context-level workflow.

The practical rule is simple: use `page.waitForPopup()` when one known page action opens the popup. Use `context.waitForPage()` when you want to capture any new page created in the context.

Both should be started before the action that creates the new page. If the click happens first and the wait starts later, Playwright may miss the page creation event.

## Common Mistake

using `context.waitForPage()` everywhere even when `page.waitForPopup()` would make the opener relationship clearer and reduce confusion in multi-page tests.

---

## Question 5.10

### How do you get all pages from a `BrowserContext` in Playwright Java?

#### Interview-Style Answer

In Playwright Java, you can get all currently open pages in a `BrowserContext` using `context.pages()`.

```
List<Page> allPages = context.pages();
```

This returns the current list of `Page` objects, including normal tabs and popup pages inside that context. It is useful for inspecting, logging, debugging, or cleaning up multiple pages, but it should not be used as the primary way to capture a page opened by a known action.

#### Detailed Explanation

A single `BrowserContext` can contain multiple `Page` objects. Each `Page` represents a tab or popup-like browser page that belongs to that isolated context.

Example:

```
BrowserContext context = browser.newContext();

Page productsPage = context.newPage();
Page cartPage = context.newPage();

productsPage.navigate("https://example.com/products");
cartPage.navigate("https://example.com/cart");

List<Page> allPages = context.pages();

System.out.println("Total pages: " + allPages.size());
```

You can iterate through the current pages when debugging or managing multiple tabs:

```
for (Page p : context.pages()) {
    System.out.println("Page URL: " + p.url());
    System.out.println("Page title: " + p.title());
}
```

This is useful for scenarios such as:

- 
- Checking how many pages are open in a context
  - Debugging unexpected tabs or popups
  - Logging page URLs during failure analysis
  - Closing extra pages after validation
  - Inspecting multi-tab workflows

However, `context.pages()` only gives the current snapshot of pages. It does not wait for a new page to open. If a specific user action should open a new tab, use `context.waitForPage()` instead:

```
Page reportPage = context.waitForPage(() -> {
    page.getByRole(
        AriaRole.LINK,
        new Page.GetByRoleOptions().setName("Open report")
    ).click();
});

reportPage.waitForLoadState();

PlaywrightAssertions.assertThat(
    reportPage.getByText("Report Summary")
).isVisible();
```

For a popup opened from a specific page, `page.waitForPopup()` is often more precise:

```
Page popup = page.waitForPopup(() -> {
    page.getByText("Open invoice").click();
});
```

So, use `context.pages()` when you need the current list of pages. Use `context.waitForPage()` or `page.waitForPopup()` when the test must reliably capture a newly opened page from a known action.

## Common Mistake

using `context.pages()` and assuming the last page in the list is always the newly opened tab. In stable automation, capture known new pages with `context.waitForPage()` or `page.waitForPopup()` instead of guessing from the page list.

---

---

## **6. Autowait**

---

## **Part I - Core Questions**



## Question 6.6

### Why is visibility not enough before clicking an element in Playwright?

#### Interview-Style Answer

Visibility only means the element is present and can be seen on the page. It does not guarantee that the element is ready for a real user click.

Before clicking, Playwright checks more than visibility. The element should be stable, enabled, attached to the DOM, and able to receive pointer events. If the element is covered by a modal, sticky header, loader, animation, badge, or overlay, it may be visible but still not practically clickable.

#### Detailed Explanation

In real web applications, an element can be visible but still not safe to click. For example, a button may appear on the screen while the page is still loading, an animation may still be moving it, or another invisible overlay may be sitting above it. A user would not be able to click the element correctly in that state, so Playwright should not click it blindly.

Example:

```
Locator accountCard = page.locator("#account-card");  
accountCard.click();
```

Before performing the click, Playwright checks whether the locator resolves to an actionable element. It verifies conditions such as visibility, stability, enabled state, and whether the element can receive pointer events. This prevents the test from passing by doing something a real user could not do.

A common case is an overlapping element:

```
Visible card:           #account-card  
Overlapping element:  status badge / loader / sticky header / modal overlay
```

Even though `#account-card` is visible, the actual click point may be blocked. In that situation, Playwright may fail the click instead of incorrectly pretending that the user can interact with the card.

A better test should wait for the real clickable state or remove the blocking condition through a valid user flow:

---

```
Locator accountCard = page.locator("#account-card");  
  
PlaywrightAssertions.assertThat(accountCard).isVisible();  
accountCard.click();
```

If a loader or overlay is expected, wait for it to disappear:

```
Locator loader = page.locator(".loading-overlay");  
PlaywrightAssertions.assertThat(loader).isHidden();  
  
page.locator("#account-card").click();
```

Using `force: true` should be rare because it bypasses Playwright's actionability checks. It may hide a real product issue where the UI looks visible but is not actually usable.

## Common Mistake

forcing the click as soon as the element is visible instead of checking why the element is not receiving pointer events.

---

## Question 6.15

### How do you remove hard waits from an existing Playwright Java suite?

#### Interview-Style Answer

To remove hard waits from an existing Playwright Java suite, first identify what each wait was trying to protect, then replace it with a real synchronization condition. That condition could be a locator assertion, URL assertion, API response wait, download wait, popup wait, navigation wait, or business-state validation.

The goal is not just to delete `waitForTimeout()`. The goal is to replace time-based waiting with condition-based waiting so the test proceeds when the application is actually ready.

#### Detailed Explanation

Hard waits like `page.waitForTimeout(3000)` make tests slower and still unreliable. If the application is ready in 500 milliseconds, the test wastes time. If the application takes 5 seconds, the test may still fail. So the correct approach is to understand why the hard wait was added and replace it with a meaningful signal.

Common replacements:

```
waitForTimeout after login
-> assert dashboard heading or user profile is visible

waitForTimeout after search
-> assert result row, result count, or empty state is visible

waitForTimeout after save
-> assert success message or updated row status is visible

waitForTimeout after export
-> use waitForDownload()

waitForTimeout after popup click
-> use waitForPopup()

waitForTimeout after API-triggering action
-> use waitForResponse() and then assert the visible UI result
```

Weak approach:

```

page.getByRole(AriaRole.BUTTON,
  new Page.GetByRoleOptions().setName("Save")
).click();

page.waitForTimeout(3000);

PlaywrightAssertions.assertThat(
  page.getByText("Saved successfully")
).isVisible();

```

**Better approach:**

```

page.getByRole(AriaRole.BUTTON,
  new Page.GetByRoleOptions().setName("Save")
).click();

PlaywrightAssertions.assertThat(
  page.getByText("Saved successfully")
).isVisible();

```

Here, the `web-first` assertion automatically retries until the success message appears or the timeout is reached.

For API-based synchronization, wait for the relevant response and then verify the user-visible result:

```

Response response = page.waitForResponse(
  res -> res.url().contains("/api/orders")
  && res.status() == 200,
  () -> {
    page.getByRole(AriaRole.BUTTON,
      new Page.GetByRoleOptions().setName("Submit Order")
    ).click();
  }
);

PlaywrightAssertions.assertThat(
  page.getByText("Order submitted successfully")
).isVisible();

```

**For downloads:**

```

Download download = page.waitForDownload(() -> {
  page.getByRole(AriaRole.BUTTON,
    new Page.GetByRoleOptions().setName("Export")
  ).click();
});

download.saveAs(Paths.get("downloads/report.xlsx"));

```

**For popups:**

---

```
Page popup = page.waitForPopup(() -> {
    page.getByRole(AriaRole.LINK,
        new Page.GetByRoleOptions().setName("Open Report")
    ).click();
});

popup.waitForLoadState();
```

In real projects, removing hard waits should be done carefully. Each wait should be replaced with the exact condition the test depends on. This improves speed, stability, and debugging because failures now point to the missing condition instead of an arbitrary timeout.

## Common Mistake

removing `waitForTimeout()` without replacing it with the actual readiness condition, which makes the test faster but more flaky.

---

---

## **7. Assertions**

---

## **Part I - Core Questions**

## Question 7.1

### What are web-first assertions in Playwright Java?

#### Interview-Style Answer

Web-first assertions are Playwright assertions that automatically retry until the expected UI condition becomes true or the assertion timeout is reached.

In Playwright Java, they are used through `PlaywrightAssertions.assertThat()`. They are useful for modern web applications because UI changes often happen asynchronously after clicks, API responses, animations, or page updates. Instead of checking the DOM only once, web-first assertions keep checking the expected condition and make tests more stable.

#### Detailed Explanation

Modern web pages do not always update immediately. After a user action, the application may call an API, update state, re-render components, show a spinner, display a success message, or update a table. Web-first assertions help handle this by waiting for the expected UI state.

Example:

```
PlaywrightAssertions.assertThat(
    page.getByText("Order submitted")
).isVisible();
```

This does not check visibility only once. Playwright keeps checking until the text becomes visible or the assertion timeout expires.

Common web-first assertions include:

```
1. isVisible()
2. isHidden()
3. hasText()
4. containsText()
5. hasURL()
6. hasCount()
7. isEnabled()
8. isDisabled()
9. isChecked()
10. hasValue()
```

For example, after submitting an order, a strong test should assert the visible business result:



---

```
page.getByRole(
    AriaRole.BUTTON,
    new Page.GetByRoleOptions().setName("Submit Order")
).click();

PlaywrightAssertions.assertThat(
    page.getByText("Order submitted successfully")
).isVisible();

PlaywrightAssertions.assertThat(
    page.getByRole(AriaRole.ROW)
        .filter(new Locator.FilterOptions().setHasText(orderId))
).containsText("Submitted");
```

Here, the test validates not only that the button was clicked, but also that the user can see the expected final result. This is more reliable than using fixed waits or checking raw DOM state at one instant.

Web-first assertions should be preferred over `Thread.sleep()` or `waitForTimeout()` because they wait for meaningful conditions instead of waiting for an arbitrary number of seconds.

## Common Mistake

Adding `Thread.sleep()` before assertions instead of using retry-friendly Playwright assertions. Fixed waits slow down the suite and still do not prove that the expected user-visible state has been reached.

---

---

## **Part II - Additional Questions**

## Question 7.25

**Can you list all commonly used Playwright Java assertions with examples?**

### Interview-Style Answer

Playwright Java provides web-first assertions through `PlaywrightAssertions.assertThat()`. These assertions are designed for dynamic web applications because they automatically wait until the expected condition becomes true or the timeout is reached.

The commonly used assertion groups are:

1. Locator assertions
2. Page assertions
3. Response assertions

Locator assertions validate element state, text, attributes, CSS, count, form values, screenshots, and visibility. Page assertions validate page-level behavior such as title, URL, and screenshots. Response assertions validate whether an API or network response succeeded.

### Detailed Explanation

Playwright assertions are preferred for UI validation because modern applications update asynchronously. A success message, table row, route change, or enabled button may not appear immediately after an action. Web-first assertions retry automatically, which makes tests more stable than checking the condition only once.

#### 1. Visibility and state assertions

```
Locator submitButton = page.getByRole(
    AriaRole.BUTTON,
    new Page.GetByRoleOptions().setName("Submit")
);

PlaywrightAssertions.assertThat(submitButton).isVisible();
PlaywrightAssertions.assertThat(submitButton).isEnabled();
```

Common state assertions include:

```
isVisible()  
isHidden()  
isEnabled()  
isDisabled()  
isEditable()  
isChecked()  
isFocused()  
isAttached()  
isInViewport()  
isEmpty()
```

## Examples:

```
PlaywrightAssertions.assertThat(page.getByTestId("loading-spinner")).isHidden();  
  
PlaywrightAssertions.assertThat(page.getByLabel("Accept terms")).isChecked();  
  
PlaywrightAssertions.assertThat(page.getByLabel("Email")).isEditable();  
  
PlaywrightAssertions.assertThat(page.getByTestId("promo-banner")).isAttached();
```

These assertions are useful for buttons, inputs, checkboxes, modals, loaders, banners, menus, and form validation.

## 2. Text assertions

```
Locator successMessage = page.getByTestId("success-message");  
  
PlaywrightAssertions.assertThat(successMessage)  
    .hasText("Order submitted successfully");
```

Use `hasText()` when the exact text matters.

Use `containsText()` when the element may contain extra text but must include a specific value:

```
PlaywrightAssertions.assertThat(successMessage)  
    .containsText("submitted");
```

For lists, `hasText()` and `containsText()` can also validate multiple visible values:

```
Locator rows = page.getByTestId("order-row");  
  
PlaywrightAssertions.assertThat(rows)  
    .containsText(Arrays.asList("Order 1001", "Pending", "₹500"));
```

## 3. Count assertions

```
Locator rows = page.getByRole(AriaRole.ROW);  
  
PlaywrightAssertions.assertThat(rows).hasCount(5);
```

---

`hasCount()` is useful for search results, table rows, cards, dropdown options, menu items, and list validation.

#### 4. Attribute, class, CSS, and property assertions

```
Locator getStarted = page.getByRole(
    AriaRole.LINK,
    new Page.GetByRoleOptions().setName("Get Started")
);

PlaywrightAssertions.assertThat(getStarted)
    .hasAttribute("href", "/docs/intro");
```

Common examples:

```
PlaywrightAssertions.assertThat(page.getByText("Dashboard"))
    .hasClass(Pattern.compile(".*active.*"));

PlaywrightAssertions.assertThat(page.getByText("Invalid password"))
    .hasCSS("color", "rgb(255, 0, 0)");

PlaywrightAssertions.assertThat(page.locator("form"))
    .hasId("login-form");

PlaywrightAssertions.assertThat(page.getByLabel("Email"))
    .hasJSPROPERTY("value", "raj@example.com");
```

These assertions should be used when the attribute, class, CSS, ID, or DOM property is part of the actual requirement. For normal business validation, visible text or role-based assertions are usually easier to maintain.

#### 5. Form value assertions

```
Locator email = page.getByLabel("Email");

email.fill("raj@example.com");

PlaywrightAssertions.assertThat(email)
    .hasValue("raj@example.com");
```

For multi-select fields:

```
Locator colors = page.getByLabel("Choose multiple colors");

colors.selectOption(new String[] {"red", "green"});

PlaywrightAssertions.assertThat(colors)
    .hasValues(new String[] {"red", "green"});
```

These are useful for textboxes, textareas, date fields, search filters, dropdowns, and multi-select controls.

#### 6. Page assertions

```
PlaywrightAssertions.assertThat(page)
    .hasTitle(Pattern.compile("Dashboard"));

PlaywrightAssertions.assertThat(page)
    .hasURL(Pattern.compile(".*orders$"));
```

Page assertions are useful after navigation, login, redirects, route changes, and SPA transitions.

Example:

```
page.getByRole(
    AriaRole.LINK,
    new Page.GetByRoleOptions().setName("Orders")
).click();

PlaywrightAssertions.assertThat(page).hasURL(Pattern.compile(".*orders$"));
PlaywrightAssertions.assertThat(
    page.getByRole(AriaRole.HEADING)
).hasText("Orders");
```

This validates both the browser route and the visible page content.

## 7. Screenshot assertions

```
PlaywrightAssertions.assertThat(page)
    .hasScreenshot("dashboard-page.png");
```

For component-level visual validation:

```
Locator invoicePreview = page.getByTestId("invoice-preview");

PlaywrightAssertions.assertThat(invoicePreview)
    .hasScreenshot("invoice-preview.png");
```

Screenshot assertions should be used when visual rendering is the requirement, such as layout, theme, spacing, chart rendering, invoice preview, or responsive UI. They should not replace normal functional assertions.

## 8. Response assertions

```
Response response = page.waitForResponse("**/api/orders", () -> {
    page.getByRole(
        AriaRole.BUTTON,
        new Page.GetByRoleOptions().setName("Submit")
    ).click();
});

PlaywrightAssertions.assertThat(response).isOK();
```

`isOK()` verifies that the response status is successful. For UI tests, the response assertion should normally be followed by a visible user outcome:

---

```
PlaywrightAssertions.assertThat(response).isOk();

PlaywrightAssertions.assertThat(
    page.getByText("Order submitted successfully")
).isVisible();
```

This proves that the backend call succeeded and the UI reflected the result.

### Practical rule

```
Use locator assertions for element-level UI validation.
Use page assertions for URL, title, and page-level checks.
Use response assertions for network/API success.
Use normal Java/JUnit/TestNG assertions for plain Java values.
```

Avoid this for dynamic UI state:

```
Assertions.assertTrue(page.getByText("Saved").isVisible());
```

This checks immediately and can fail before the UI updates.

Prefer:

```
PlaywrightAssertions.assertThat(
    page.getByText("Saved")
).isVisible();
```

This waits until the expected condition is met or the assertion timeout is reached.

## Common Mistake

using normal JUnit or TestNG assertions for dynamic UI conditions. Plain Java assertions check the current state immediately, while Playwright web-first assertions wait for the UI to reach the expected state, making them more reliable for real web applications.

---

---

## 8. Locators



---

## **Part I - Core Questions**

### Question 8.3

## How is a Playwright `Locator` different from a one-time element lookup?

### Interview-Style Answer

A Playwright `Locator` is a reusable query that resolves the element at the time of action or assertion. It does not permanently store one fixed DOM element when it is created.

A one-time element lookup captures the page state at a specific moment. If the page later re-renders, replaces the element, or updates the DOM, that old reference may no longer represent the current UI. Playwright `Locator` avoids many stale-element style problems because it re-evaluates the target when operations like `click()`, `fill()`, or web-first assertions run.

### Detailed Explanation

Modern web applications often update the DOM after initial load. Frameworks like React, Angular, and Vue may replace elements during rendering, refresh lists after API calls, or rebuild sections of the page after user actions.

A one-time element lookup depends on what existed at that exact moment. If the element is later replaced, the stored reference can become outdated. A `Locator` works differently. It stores the selector logic, not a fixed element instance.

Example:

```
Locator getStarted = page.locator("text=Get Started");

PlaywrightAssertions.assertThat(getStarted)
    .hasAttribute("href", "/docs/intro");

getStarted.click();
```

Here, `getStarted` can be declared before the element is ready. When `hasAttribute()` runs, Playwright resolves the current matching element and retries the assertion until the expected attribute appears or timeout happens. When `click()` runs, Playwright again resolves the locator and waits for actionability conditions such as visibility, stability, enabled state, and ability to receive events.

---

This makes locators more reliable for dynamic pages. They also support strictness, meaning Playwright expects action locators to identify a clear target. If the locator matches multiple elements during an action, Playwright can fail instead of clicking an unintended element.

A better locator also improves maintainability:

```
Locator getStarted = page.getByRole(  
    AriaRole.LINK,  
    new Page.GetByRoleOptions().setName("Get Started")  
);
```

This expresses the user-facing intent more clearly than a fragile CSS path or XPath tied to DOM structure.

## Common Mistake

assuming a [Locator](#) must find the element immediately when it is declared. A locator is resolved when it is used for an action or assertion, not when the variable is created.

---

## Question 8.19

### What is a good locator priority order in Playwright Java?

#### Interview-Style Answer

A good locator priority order in Playwright Java starts with user-facing locators such as `getByRole()`, `getByLabel()`, `getByPlaceholder()`, and `getByText()`, then uses `getByTestId()` for stable automation hooks, followed by stable CSS selectors when needed. XPath should usually be the last option.

The goal is to choose locators that express user intent, remain unique under Playwright strict mode, work well with auto-waiting and web-first assertions, and stay maintainable after UI refactoring.

#### Detailed Explanation

A practical locator priority order can be:

1. `getByRole()`
2. `getByLabel()`
3. `getByPlaceholder()`
4. `getByText()`
5. `getByTestId()`
6. Stable CSS selector
7. XPath only as a last option

For buttons, links, headings, checkboxes, radio buttons, dialogs, tabs, and menu items, `getByRole()` with an accessible name is usually the best option:

```
page.getByRole(
    AriaRole.BUTTON,
    new Page.GetByRoleOptions().setName("Submit")
).click();
```

This is better than a styling-based XPath or CSS selector:

```
page.locator("//button[@class='btn primary']").click();
```

The role-based locator is more readable because it says the user clicks the Submit button. It is also less dependent on DOM structure or styling classes.

For form fields, `getByLabel()` is usually strong:

```
page.getByLabel("Email").fill("user@example.com");
```

---

For repeated rows or cards, combine locator priority with scoping and filtering:

```
Locator orderRow = page.getByRole(AriaRole.ROW)
    .filter(new Locator.FilterOptions().setHasText("ORD-1001"));

orderRow.getByRole(
    AriaRole.BUTTON,
    new Locator.GetByRoleOptions().setName("Edit")
).click();
```

This avoids fragile index-based locators and clearly identifies the correct row before clicking the button inside it.

`getByTestId()` is useful when the UI has no stable accessible name or when the element is difficult to identify reliably through user-facing locators. CSS is acceptable for stable automation-safe attributes, but styling classes should be avoided. XPath can be used when no better option exists, but it should be readable, scoped, and not dependent on fragile DOM positions.

## Common Mistake

Starting with XPath or CSS class selectors for every element even when Playwright provides stronger user-facing locators. This makes tests harder to read, more fragile during UI refactoring, and more likely to fail strictness or maintenance reviews.

---

## Question 8.34

### How would you debug a locator that unexpectedly matches multiple elements?

#### Interview-Style Answer

I would first confirm why the locator is matching multiple elements instead of immediately using `.first()` or `.nth(0)`. Multiple matches usually mean the locator is too broad or missing business context.

In Playwright Java, I would check the match count, inspect the matching candidates using Trace Viewer or Inspector, and then narrow the locator using accessible role and name, parent scoping, `filter()`, `hasText`, `has`, or a stable test id.

#### Detailed Explanation

Playwright locators are strict for actions. If a locator used for `click()`, `fill()`, or similar actions matches more than one element, Playwright may fail instead of guessing. This is useful because it prevents the test from accidentally interacting with the wrong element.

A broad locator like this may fail if there are many Approve buttons on the page:

```
page.getByRole(  
    AriaRole.BUTTON,  
    new Page.GetByRoleOptions().setName("Approve")  
)  
.click();
```

The first debugging step is to understand what matched. You can check the count and inspect the candidates:

```
Locator approveButtons = page.getByRole(  
    AriaRole.BUTTON,  
    new Page.GetByRoleOptions().setName("Approve")  
);  
  
System.out.println("Matching buttons: " + approveButtons.count());
```

Then identify the real business target. For example, if the test needs to approve order ORD-1001, scope the button to that row:

```
Locator row = page.getByRole(AriaRole.ROW)
    .filter(new Locator.FilterOptions().setHasText("ORD-1001"));

row.getByRole(
    AriaRole.BUTTON,
    new Locator.GetByRoleOptions().setName("Approve")
).click();
```

This is better because the locator now expresses the real intent: approve the row for ORD-1001, not just any Approve button.

For card-based layouts, use the same idea:

```
Locator userCard = page.getByTestId("user-card")
    .filter(new Locator.FilterOptions().setHasText("Raj Kumar"));

userCard.getByRole(
    AriaRole.BUTTON,
    new Locator.GetByRoleOptions().setName("Edit")
).click();
```

Good debugging steps are:

1. Check how many elements match.
2. Inspect each matching candidate in Trace Viewer or Playwright Inspector.
3. Identify the exact business target.
4. Scope the locator to a row, card, dialog, form, or section.
5. Use role/name, filter, hasText, has, or test id to make the locator unique.

Using `.first()` or `.nth(0)` is acceptable only when position is part of the requirement. Otherwise, it hides ambiguity and may click the wrong element when table order, sorting, filtering, or test data changes.

## Common Mistake

fixing a strictness error by adding `.first()` without checking why multiple elements matched and without narrowing the locator to the correct business context.

---

## 9. Actions



---

## **Part I - Core Questions**

## Question 9.8

### Why is it important to use logical key names such as ArrowRight, Backspace, or Enter?

#### Interview-Style Answer

It is important to use logical key names because Playwright needs to know whether the test is sending an actual keyboard key or typing normal text. Keys such as `ArrowRight`, `Backspace`, `Enter`, `Tab`, and shortcuts like `Control+A` represent real keyboard actions.

In Playwright Java, `press("Enter")` sends the Enter key, but `fill("Enter")` enters the word Enter into the field. Similarly, `pressSequentially("Enter")` types the characters `E`, `n`, `t`, `e`, and `r`.

Logical key names are important for testing keyboard navigation, shortcuts, form submission, text editing, accessibility behavior, and command-style UIs.

#### Detailed Explanation

Keyboard-based automation often needs actual keyboard events, not visible text. Playwright uses logical key names to represent these real keyboard actions.

Example:

```
page.getByRole(AriaRole.TEXTBOX).press("Backspace");
page.getByRole(AriaRole.TEXTBOX).press("Enter");
page.getByRole(AriaRole.TEXTBOX).press("Control+ArrowRight");
```

Here, `Backspace` deletes text, `Enter` submits or confirms, and `Control+ArrowRight` moves the cursor by word depending on the application and operating system behavior.

This is different from filling or typing text:

```
page.getByRole(AriaRole.TEXTBOX).fill("Enter");
```

This does not press the Enter key. It sets the textbox value to the word Enter.

Similarly:

```
page.getByRole(AriaRole.TEXTBOX).pressSequentially("Enter");
```

This types each character of the word Enter, not the Enter key.

---

Logical key names are especially useful when validating:

- Keyboard shortcuts
- Search submission using Enter
- Focus movement using Tab
- Text deletion using Backspace
- Cursor movement using ArrowLeft or ArrowRight
- Accessibility keyboard navigation
- Dropdowns, menus, and command palettes

For example, a search box may show results only after pressing Enter:

```
Locator searchBox = page.getByRole(  
    AriaRole.TEXTBOX,  
    new Page.GetByRoleOptions().setName("Search")  
);  
  
searchBox.fill("Playwright Java");  
searchBox.press("Enter");  
  
PlaywrightAssertions.assertThat(  
    page.getByText("Search results")  
).isVisible();
```

Using the correct logical key makes the test match real user keyboard behavior and avoids confusing text input with keyboard commands.

### Common Mistake

using `fill("Enter")` or `pressSequentially("Enter")` when the application needs the actual Enter key. This changes the input text instead of triggering the keyboard event.

---

## Question 9.19

**What would you do if `dragTo()` does not trigger drag-and-drop correctly in all browsers?**

### Interview-Style Answer

If `dragTo()` does not trigger drag-and-drop correctly in all browsers, I would use Playwright's lower-level mouse actions to manually perform the drag operation.

Some pages depend on the `dragover` event being dispatched before the drop is accepted. Playwright documentation recommends using `hover()`, `page.mouse().down()`, repeated movement or hover over the drop target, and then `page.mouse().up()`. The repeated hover or mouse move is important because some browsers need at least two mouse movements to reliably trigger `dragover`.

### Detailed Explanation

`dragTo()` is convenient, but some custom drag-and-drop implementations need more precise mouse control. This is common in boards, calendars, custom lists, workflow tools, and drag-and-drop UI libraries.

A reliable manual sequence is:

1. Hover the draggable element.
2. Press the mouse down.
3. Hover the drop target.
4. Hover the drop target a second time.
5. Release the mouse.
6. Verify the final UI result.

Example:

```
Locator source = page.getByText("Task A");
Locator target = page.getTestId("done-column");

source.hover();
page.mouse().down();

target.hover();
target.hover();

page.mouse().up();

PlaywrightAssertions.assertThat(
    target.getByText("Task A")
).isVisible();
```

---

The second `target.hover()` is not random. It is useful because if the page relies on the `dragover` event, Playwright documentation says at least two mouse moves may be needed to trigger it reliably in all browsers.

If more control is needed, I can also use explicit mouse coordinates:

```
BoundingBox sourceBox = source.boundingBox();
BoundingBox targetBox = target.boundingBox();

page.mouse().move(
    sourceBox.x + sourceBox.width / 2,
    sourceBox.y + sourceBox.height / 2
);

page.mouse().down();

page.mouse().move(
    targetBox.x + targetBox.width / 2,
    targetBox.y + targetBox.height / 2
);

page.mouse().move(
    targetBox.x + targetBox.width / 2,
    targetBox.y + targetBox.height / 2
);

page.mouse().up();

PlaywrightAssertions.assertThat(
    target.getByText("Task A")
).isVisible();
```

Important checks:

1. Source locator should point to the actual draggable element.
2. Target locator should point to the real drop area.
3. The target should be visible and not covered by another element.
4. The drag movement should trigger the expected dragover behavior.
5. The final UI state should confirm that the item was dropped correctly.

## Common Mistake

Using only `dragTo()` and assuming the drag succeeded because no exception was thrown. A good test should verify the final user-visible result after the drag-and-drop operation.

---

---

## **10. Authentication**

---

## **Part I - Core Questions**

## Question 10.7

**How would you create storage state for different user roles?**

### Interview-Style Answer

I would log in once per user role, save a separate storage-state file for each role, and load the correct file based on the test scenario.

Each role should have its own cookies, local storage, permissions, and access level. For example, admin, manager, employee, and viewer should not share the same authenticated state because role-specific defects can be missed if every test runs with an admin session.

### Detailed Explanation

Different roles should have separate storage-state files:

```
auth/admin.json
auth/manager.json
auth/employee.json
auth/viewer.json
```

A setup step can log in as each role and save its state:

```
BrowserContext context = browser.newContext();
Page page = context.newPage();

page.navigate(baseUrl + "/login");

page.getByLabel("Email").fill("manager@example.com");
page.getByLabel("Password").fill("managerPassword");

page.getByRole(
    AriaRole.BUTTON,
    new Page.GetByRoleOptions().setName("Login")
).click();

PlaywrightAssertions.assertThat(
    page.getByText("Dashboard")
).isVisible();

context.storageState(
    new BrowserContext.StorageStateOptions()
        .setPath(Paths.get("auth/manager.json"))
);

context.close();
```

Then the test can load the role-specific storage state:



```
BrowserContext context = browser.newContext(  
    new Browser.NewContextOptions()  
        .setStorageStatePath(Paths.get("auth/manager.json"))  
);  
  
Page page = context.newPage();  
page.navigate(baseUrl + "/dashboard");
```

This is useful for:

1. Admin permission tests
2. Approval workflows
3. Viewer-only access checks
4. Maker-checker scenarios
5. Role-specific dashboard validation
6. Negative authorization tests
7. Menu and navigation visibility checks

For maintainability, storage-state files should be clearly named by role and environment. Sensitive files containing cookies or tokens should not be committed to Git. They should be protected through [.gitignore](#), regenerated when credentials or permissions change, and handled carefully in CI/CD using secrets or secure setup jobs.

After loading a role's storage state, the test should still validate that the expected role is active:

```
PlaywrightAssertions.assertThat(  
    page.getByText("Manager Dashboard")  
).isVisible();  
  
PlaywrightAssertions.assertThat(  
    page.getByRole(  
        AriaRole.BUTTON,  
        new Page.GetByRoleOptions().setName("Approve")  
    )  
).isVisible();
```

## Common Mistake

using one admin storage-state file for all tests. This makes tests faster but hides permission, access-control, menu-visibility, and role-specific workflow defects.

## Question 10.11

### Why should authentication state files not be committed to Git?

#### Interview-Style Answer

Authentication state files should not be committed to Git because they can contain sensitive session data such as cookies, local storage values, tokens, and other browser authentication information. Anyone who gets access to that file may be able to reuse the session and impersonate the test user.

In Playwright Java, storage-state files should be treated like secrets, not normal test data. They should be generated locally or securely in CI/CD, stored in a dedicated auth folder, and excluded from source control using `.gitignore`.

This is especially important for shared repositories, private company repositories, and public Git hosting because leaked auth state can create security and compliance risks.

#### Detailed Explanation

When Playwright saves authenticated state, it writes the browser context's logged-in state into a JSON file. That file may include session cookies, authentication tokens, local storage values, and other data required by the application to recognize the user as signed in.

Example:

```
context.storageState(  
    new BrowserContext.StorageStateOptions()  
        .setPath(Paths.get("playwright/.auth/state.json"))  
);
```

Although the file may look like a normal JSON file, it is not ordinary test data. If the session is still valid, someone with access to this file may be able to load it into a new `BrowserContext` and access the application as that user.

Example reuse:

```
BrowserContext context = browser.newContext(  
    new Browser.NewContextOptions()  
        .setStorageStatePath(Paths.get("playwright/.auth/state.json"))  
);
```

---

That is useful for automation, but risky if the file is exposed.

A safer project structure is:

```
playwright/.auth/state.json  
playwright/.auth/admin.json  
playwright/.auth/user.json
```

Then exclude the folder in `.gitignore`:

```
playwright/.auth
```

In CI/CD, the state file should be generated during setup or fetched from a secure secret-management process, not stored permanently in the repository. It should also be regenerated or rotated when sessions expire, passwords change, permissions change, or environments are refreshed.

For role-based testing, separate state files should be used for different users, but all of them should still be protected:

```
playwright/.auth/admin.json  
playwright/.auth/buyer.json  
playwright/.auth/approver.json
```

This prevents accidental exposure of privileged sessions and avoids mixing authentication state across roles.

## Common Mistake

treating `state.json` as a harmless test fixture and committing it with the automation framework. A storage-state file may contain valid authentication data, so it should be ignored by Git, regenerated securely, and protected like any other secret.

---

---

## **11. Download**

---

## **Part I - Core Questions**

## Question 11.1

### How do you handle file downloads in Playwright Java?

#### Interview-Style Answer

In Playwright Java, file downloads should be handled using `page.waitForDownload()` around the action that triggers the download. The wait must start before clicking the download button because download is a browser event.

After the download starts, Playwright returns a `Download` object. From that object, we can get the suggested filename, access the temporary path, or save the file to a project-controlled location using `saveAs()`.

A reliable download test should not depend on the operating system's default downloads folder. It should save the file to a known path, verify that the file exists, and validate the file name, size, extension, or content based on the business requirement.

#### Detailed Explanation

Downloads are event-driven. If the test clicks the download button first and only then starts looking for the downloaded file, it may miss the download event. That is why the correct pattern is to start `waitForDownload()` before the action.

Example:

```
Download download = page.waitForDownload(() -> {
    page.getByRole(
        AriaRole.BUTTON,
        new Page.GetByRoleOptions().setName("Export CSV")
    ).click();
});
```

Once the download is captured, save it to a controlled location:

```
String suggestedName = download.suggestedFilename();

Path downloadDir = Paths.get("target/downloads");
Files.createDirectories(downloadDir);

Path savedPath = downloadDir.resolve(suggestedName);
download.saveAs(savedPath);

Assertions.assertTrue(Files.exists(savedPath));
Assertions.assertTrue(Files.size(savedPath) > 0);
```

This is better than reading from the default downloads folder because the test controls where the file is stored. It also makes the test more

---

reliable in local runs, CI/CD pipelines, Docker containers, and parallel execution.

For parallel tests, use a unique folder or filename to avoid collisions:

```
Path downloadDir = Paths.get(
    "target/downloads",
    UUID.randomUUID().toString()
);
Files.createDirectories(downloadDir);

Path savedPath = downloadDir.resolve(download.suggestedFilename());
download.saveAs(savedPath);
```

After saving the file, validate the actual business expectation. For example, for a CSV export, check the filename, extension, file size, headers, or expected row content. For a PDF, check that the file exists and contains expected report information if the framework supports PDF validation.

## Common Mistake

clicking the download button and then searching the operating system's default downloads folder. This is unreliable because the file location may differ by machine, browser settings, CI environment, or parallel test execution.

---

---

## **12. File Upload**



---

## **Part I - Core Questions**

## Question 12.6

### How do you remove all selected files from a file input in Playwright Java?

#### Interview-Style Answer

In Playwright Java, you can remove all selected files from a file input by calling `setInputFiles()` with an empty `Path` array.

```
page.getByLabel("Upload file").setInputFiles(new Path[0]);
```

This clears the file selection from the `<input type="file">`. It is useful when testing flows where a user selects the wrong file, removes an uploaded file before submitting, or resets the upload field before choosing another file.

After clearing the input, the test should verify the visible application behavior, such as the file name disappearing, preview being removed, validation message appearing, or submit button becoming disabled.

#### Detailed Explanation

`setInputFiles()` is normally used to select one or more files in a file input.

For a single file:

```
page.getByLabel("Upload file")
    .setInputFiles(Paths.get("myfile.pdf"));
```

For multiple files:

```
page.getByLabel("Upload files")
    .setInputFiles(new Path[] {
        Paths.get("file1.txt"),
        Paths.get("file2.txt")
    });
```

To remove all selected files, pass an empty `Path` array:

```
page.getByLabel("Upload file")
    .setInputFiles(new Path[0]);
```

This resets the file input selection. But the test should not stop at the action. A reliable test should also confirm how the application reacts after the file is cleared.

Example:

```
Locator uploadInput = page.getByLabel("Upload file");
uploadInput.setInputFiles(Paths.get("invoice.pdf"));

PlaywrightAssertions.assertThat(
    page.getByText("invoice.pdf")
).isVisible();

uploadInput.setInputFiles(new Path[0]);

PlaywrightAssertions.assertThat(
    page.getByText("invoice.pdf")
).isHidden();

PlaywrightAssertions.assertThat(
    page.getByRole(
        AriaRole.BUTTON,
        new Page.GetByRoleOptions().setName("Submit")
    )
).isDisabled();
```

This verifies both the technical action and the user-visible result. The selected file is removed, the file name is no longer shown, and the application returns to a state where submission is not allowed without a required file.

This pattern is also useful when testing upload validation scenarios, such as clearing an invalid file type, removing an oversized file, replacing a file, or confirming that upload-related error messages are reset correctly.

## Common Mistake

clearing the file input but not validating the UI result. The important test assertion is not only that `setInputFiles(new Path[0])` was called, but that the application correctly removes the preview, file name, validation state, or submit readiness after the file is cleared.

---

## **13. Handles**

---

## **Part I - Core Questions**

## Question 13.1

### What are handles in Playwright Java?

#### Interview-Style Answer

Handles in Playwright Java are references to objects that exist inside the browser page context. The real object stays in the browser, while the Java test code receives a handle that can be used to inspect or pass that browser-side object.

There are two common handle types:

1. `JSHandle`  
-> Reference to any JavaScript object in the page
2. `ElementHandle`  
-> Reference to a DOM element in the page

An `ElementHandle` is a specialized kind of `JSHandle` because every DOM element is also a JavaScript object.

#### Detailed Explanation

Playwright Java tests and browser-side JavaScript run in separate environments:

Java test process	-> Playwright test code
Browser page context	-> DOM, window, document, JavaScript objects

Because of this separation, Java code cannot directly own the real browser-side object. Instead, Playwright creates a handle that points to that object.

Example of a `JSHandle`:

```
JSHandle windowHandle = page.evaluateHandle("() => window");
```

Here, `window` is a JavaScript object inside the page, and `windowHandle` is the Java-side reference to it.

For DOM elements, Playwright provides `ElementHandle`:

```
ElementHandle box = page.querySelector("#box");  
String classValue = box.getAttribute("class");  
BoundingBox boundingBox = box.boundingBox();
```

Handles can also be passed back into `page.evaluate()`:

---

```
ElementHandle button = page.querySelector("button");

Object text = page.evaluate(
    "button => button.textContent",
    button
);
```

This is useful when JavaScript needs to operate on a specific browser-side object.

Handles are helpful for advanced use cases such as:

- Passing DOM elements or JavaScript objects into `page.evaluate()`
- Reading low-level DOM properties
- Getting bounding box information
- Working with browser-side objects that cannot be fully serialized
- Debugging advanced page behavior

However, handles should not be the default choice for normal UI automation. For clicking, filling, waiting, and assertions, `Locator` is usually better because it is re-resolved when used and works well with auto-waiting and web-first assertions.

Preferred normal automation style:

```
Locator box = page.locator("#box");

PlaywrightAssertions.assertThat(box)
    .hasAttribute("class", Pattern.compile("highlighted"));
```

## Common Mistake

treating `ElementHandle` like Selenium `WebElement` and using it everywhere for normal automation. In Playwright Java, prefer `Locator` for most UI actions and assertions, and use `JSHandle` or `ElementHandle` only when a low-level reference to a browser-side object is truly needed.

---

---

## **14. Evaluating Javascript**



---

## **Part I - Core Questions**

### Question 14.3

**What kind of values can be passed as the optional argument to `Page.evaluate()`?**

### Interview-Style Answer

`Page.evaluate()` can accept one optional argument from Java and pass it into the browser-side JavaScript function.

That argument can be:

1. Primitive values
2. Arrays / Lists
3. Objects / Maps
4. `JSHandle` or `ElementHandle` instances
5. A combination of serializable values and handles

The key rule is that Java variables are not directly available inside the browser page. Anything needed by the evaluated JavaScript must be passed explicitly through this single optional argument. If multiple values are needed, wrap them in a Java `Map` or object-like structure.

### Detailed Explanation

`Page.evaluate()` runs JavaScript in the browser page context, not in the Java test context. Because these are two separate runtimes, Playwright must serialize the Java value or pass a browser-side handle into the evaluated function.

Primitive values can be passed directly:

```
Object result = page.evaluate("num => num * 2", 42);
```

Lists can also be passed:

```
Object count = page.evaluate(
    "items => items.length",
    Arrays.asList("A", "B", "C")
);
```

When multiple values are needed, use a `Map`:

```
Map<String, Object> user = new HashMap<>();
user.put("name", "Raj");
user.put("age", 25);

Object result = page.evaluate(
    "user => user.name + ' - ' + user.age",
    user
);
```

---

Playwright converts the Java `Map` into a JavaScript object, so the browser-side function can access properties such as `user.name` and `user.age`.

Handles can also be passed. For example, an `ElementHandle` represents a real DOM element in the browser:

```
ElementHandle button = page.querySelector("button");

Object text = page.evaluate(
    "button => button.textContent",
    button
);
```

A `JSHandle` can be used when the value represents a browser-side JavaScript object:

```
JSHandle userHandle = page.evaluateHandle(
    "() => ({ name: 'Raj', role: 'admin' })"
);

Object name = page.evaluate(
    "user => user.name",
    userHandle
);
```

You can also combine handles and normal serializable values inside one argument:

```
ElementHandle button = page.querySelector("button");

Map<String, Object> arg = new HashMap<>();
arg.put("button", button);
arg.put("prefix", "Button text: ");

Object result = page.evaluate(
    "arg => arg.prefix + arg.button.textContent",
    arg
);
```

This pattern is useful when browser-side JavaScript needs both DOM references and simple Java-side values. However, only one optional argument is allowed, so multiple inputs should be grouped into a single `Map`.

## Common Mistake

trying to pass multiple separate Java arguments to `Page.evaluate()` as if it were a normal Java method call. Playwright accepts only one optional argument, so multiple values should be wrapped inside a `Map` and then accessed by property name inside the browser-side function.

---

## **15. Navigation**

---

## **Part I - Core Questions**

## Question 15.10

### What is the difference between navigation and loading in Playwright?

#### Interview-Style Answer

In Playwright, navigation and loading are related but not the same.

Navigation is the process of moving the page to a new document or URL. It can start when the test calls `page.navigate()`, clicks a link, submits a form, or triggers a JavaScript navigation. Navigation is considered committed only after the response headers are received and the browser session history is updated.

Loading starts after navigation is committed. It includes downloading the document body, parsing HTML, firing `DOMContentLoaded`, executing scripts, loading resources such as CSS and images, firing the `load` event, and then possibly executing more dynamically loaded scripts.

So, navigation proves that the browser accepted and committed the new document, while loading is the process of making that document available and functional in the page.

#### Detailed Explanation

Playwright separates navigation from loading because showing a new document in the browser happens in stages.

Navigation can start in different ways:

1. Calling `page.navigate()`
2. Clicking a normal link
3. Submitting a form
4. Triggering JavaScript navigation
5. Redirecting from application code

However, every navigation intent does not always become a successful page load.

For example:

1. Navigation may fail because the domain cannot be resolved.
2. Navigation may be blocked or canceled.
3. Navigation may be redirected.
4. Navigation may be transformed into a file download.

A navigation is committed when the browser has received and parsed the response headers and updated session history. At that point,

---

`page.url()` is updated to the new URL, and the browser starts loading the new document.

After navigation is committed, loading continues in stages:

1. Document content is downloaded.
2. HTML is parsed.
3. `DOMContentLoaded` event is fired.
4. Scripts execute.
5. Stylesheets, images, and other resources load.
6. Load event is fired.
7. Dynamically loaded scripts may continue running.

Example:

```
Response response = page.navigate("https://example.com/orders");

System.out.println("Current URL: " + page.url());

page.waitForLoadState(LoadState.DOMCONTENTLOADED);

PlaywrightAssertions.assertThat(
    page.getByRole(
        AriaRole.HEADING,
        new Page.GetByRoleOptions().setName("Orders")
    )
).isVisible();
```

The important point is that `DOMContentLoaded` or `load` does not always mean the application is fully ready from a business perspective. Modern applications may still load data, execute dynamic scripts, hydrate components, or update UI after these browser events.

For that reason, a good Playwright test should usually validate the final user-visible result:

```
page.navigate("https://example.com/orders");

PlaywrightAssertions.assertThat(
    page.getByRole(
        AriaRole.HEADING,
        new Page.GetByRoleOptions().setName("Orders")
    )
).isVisible();

PlaywrightAssertions.assertThat(
    page.locator(".order-row")
).hasCount(10);
```

## Common Mistake

Assuming navigation, `DOMContentLoaded`, and `load` all mean the same thing. Navigation only means the new document was reached or committed, while loading and application readiness may continue after that.

---

## **16. Dialogs**



---

## **Part I - Core Questions**

## Question 16.5

### How would you handle a JavaScript alert in Playwright Java?

#### Interview-Style Answer

I would handle a JavaScript alert by registering a `page.onDialog()` handler before performing the action that triggers the alert. Inside the handler, I would verify the alert message and accept the dialog.

After accepting the alert, I would validate the final user-visible result, such as a success message, updated record, changed status, or absence of an error.

#### Detailed Explanation

JavaScript alerts are browser-level dialogs. They are not normal HTML elements, so they cannot be handled with `Locator` APIs. They should be handled using Playwright's dialog event.

The dialog handler must be registered before the alert appears:

```
page.onDialog(dialog -> {
    Assertions.assertEquals("Record saved successfully", dialog.message());
    dialog.accept();
});

page.getByRole(
    AriaRole.BUTTON,
    new Page.GetByRoleOptions().setName("Save")
).click();
```

In this example, Playwright is ready to handle the alert before the Save button is clicked. When the alert appears, the handler verifies the message and accepts it.

After accepting the alert, the test should validate the application result:

```
PlaywrightAssertions.assertThat(
    page.getByText("Record saved successfully")
).isVisible();
```

Good alert handling includes:

1. Register the dialog handler before the triggering action.
2. Verify the alert message using `dialog.message()`.
3. Accept the alert using `dialog.accept()`.
4. Assert the final page state after the alert is handled.
5. Avoid accepting every alert blindly in a global handler.

---

This is important because an alert blocks page interaction until it is handled. If the handler is registered too late, the test may hang or timeout.

### **Common Mistake**

Clicking the button first and registering `page.onDialog()` afterward, which can leave the JavaScript alert already open and block the test before Playwright can handle it.

---

---

## **17. Events**

---

## **Part I - Core Questions**

## Question 17.5

### Why should `waitFor*` methods be preferred when waiting for browser events in Playwright Java?

#### Interview-Style Answer

`waitFor*` methods should be preferred because they wait for real browser events instead of relying on guessed delays. In Playwright Java, methods like `waitForResponse()`, `waitForRequest()`, `waitForPopup()`, `waitForDownload()`, and navigation waits keep Playwright's synchronous API message loop active while waiting.

They are more reliable because the wait is connected to the actual event caused by the user action. The best pattern is to start waiting first, perform the action inside the callback, and then validate the result.

#### Detailed Explanation

Modern web applications react to user actions asynchronously. A single click may trigger a network request, response, navigation, popup, download, or dialog. If the test uses a fixed wait, it is only guessing how long the application needs.

A better approach is to wait for the exact browser event:

```
Response response = page.waitForResponse(
    res -> res.url().contains("/api/orders") && res.status() == 200,
    () -> {
        page.getByRole(
            AriaRole.BUTTON,
            new Page.GetByRoleOptions().setName("Submit")
        ).click();
    }
);

System.out.println(response.status());
```

Here, Playwright starts waiting before the click happens. When the click triggers the matching response, Playwright captures it safely.

For popup events:

```
Page popup = page.waitForPopup(() -> {
    page.getByText("Open report").click();
});

popup.waitForLoadState();

PlaywrightAssertions.assertThat(
    popup.getByText("Report Summary")
).isVisible();
```

This is stronger than clicking first and then sleeping because it proves the expected popup was actually created.

Poor approach:

```
page.getByText("Open report").click();
Thread.sleep(5000);
```

This is weak because five seconds may be too short, unnecessarily long, or completely unrelated to the real browser event. Also, `Thread.sleep()` blocks the Java thread and can delay Playwright event dispatching in the synchronous API.

`waitFor*` methods are useful for events such as:

```
- waitForResponse()
- waitForRequest()
- waitForPopup()
- waitForDownload()
- waitForFileChooser()
- waitForNavigation-related conditions
```

After capturing the event, the test should still verify the user-visible outcome. For example, after an API response, assert that the order confirmation appears. After a download, verify the downloaded file. After a popup, verify the popup content.

## Common Mistake

using `Thread.sleep()` after an action and assuming the expected browser event happened. A reliable Playwright Java test should start the relevant `waitFor*` before the triggering action and then assert the visible or downloadable result produced by that event.

---

## **18. Test Isolation**



---

## **Part I - Core Questions**

## Question 18.2

### Why are browser contexts important for test isolation in Playwright?

#### Interview-Style Answer

Browser contexts are important because they give each test a clean and isolated browser environment. A `BrowserContext` has its own cookies, local storage, session storage, permissions, cache, and browser state.

In Playwright Java, this means one test can log in, change settings, add items to a cart, or update local storage without affecting another test. Each test can create its own context and page, run independently, and then close the context after completion.

This isolation is one of the main reasons Playwright tests are stable, parallel-safe, and easier to debug. The browser can be launched once, but each test can still run inside its own independent context.

#### Detailed Explanation

A `BrowserContext` is like a separate browser profile. Pages created inside one context share that context's cookies and storage, but they do not share state with pages in another context.

The usual structure is:

```
Browser → BrowserContext → Page
```

The browser process can be reused for efficiency, while each test gets a fresh context for isolation.

Example:

```
Browser browser = playwright.chromium().launch();

// Test 1 gets its own isolated context
BrowserContext context1 = browser.newContext();
Page page1 = context1.newPage();

page1.navigate("https://example.com");
// Test 1 actions

context1.close();

// Test 2 gets another isolated context
BrowserContext context2 = browser.newContext();
Page page2 = context2.newPage();

page2.navigate("https://example.com");
// Test 2 actions

context2.close();

browser.close();
```

This prevents hidden dependency between tests. Without context isolation, one test may leave behind cookies, login sessions, cart data, language settings, theme preferences, feature flags, or local storage values that affect the next test.

For example, if one test logs in as an admin and another test expects an unauthenticated user, reusing the same context can cause the second test to start in the wrong state. Similarly, if one test changes the application language or adds a product to the cart, another test may fail because it receives leftover state.

Browser contexts are also important for parallel execution. Each test can run with its own context, test data, files, and user role without interfering with other tests. For role-based testing, separate contexts can be created for different users:

```
BrowserContext adminContext = browser.newContext(
    new Browser.NewContextOptions()
        .setStorageStatePath(Paths.get("auth/admin.json"))
);

BrowserContext userContext = browser.newContext(
    new Browser.NewContextOptions()
        .setStorageStatePath(Paths.get("auth/user.json"))
);
```

This keeps admin and normal user sessions separate while still allowing efficient browser reuse.

## Common Mistake

reusing the same [Page](#) or [BrowserContext](#) across many independent tests. This can cause state leakage through old cookies, login sessions,

---

local storage, cached data, or previous test actions. A better practice is to create a fresh `BrowserContext` for each independent test.

---

---

## 19. Frames

---

## **Part I - Core Questions**

## Question 19.5

### How do you locate an element inside a nested iframe?

#### Interview-Style Answer

To locate an element inside a nested iframe in Playwright Java, chain `frameLocator()` calls for each iframe level and then use a normal locator inside the innermost frame.

Each iframe has its own separate document, so the locator chain must explicitly cross every iframe boundary. If the element is inside an iframe within another iframe, the test must first target the outer iframe, then the inner iframe, and only then locate the target element.

This approach is clearer and more maintainable than trying to use a long CSS or XPath selector because it shows the actual frame hierarchy in the test code.

#### Detailed Explanation

Nested iframes require frame-by-frame targeting. A locator from the main page cannot directly search inside an iframe, and a locator inside the outer iframe cannot directly search inside an inner iframe unless the locator chain crosses into that inner iframe.

Example:

```
Locator cardInput = page
    .frameLocator("iframe#outer-payment")
    .frameLocator("iframe#card-details")
    .getByLabel("Card number");

cardInput.fill("4111111111111111");

PlaywrightAssertions.assertThat(cardInput)
    .hasValue("4111111111111111");
```

In this example:

1. `frameLocator("iframe#outer-payment")` targets the outer iframe.
2. `frameLocator("iframe#card-details")` targets the nested iframe inside it.
3. `getByLabel("Card number")` locates the input inside the innermost iframe.

This is common in payment gateways, embedded checkout pages, authentication widgets, captcha flows, and third-party forms where one iframe may contain another iframe.

The advantage of chaining `frameLocator()` is readability. The test clearly shows where the element lives. If the test fails, it is easier to

---

debug whether the outer frame, inner frame, or final element locator is the problem.

A weak approach would be trying to write a long selector as if all elements were in the same DOM. That does not properly express iframe boundaries and can make the test brittle or incorrect.

### **Common Mistake**

skipping one iframe level and trying to locate the final element directly. Playwright will then search in the wrong document context, and the locator may fail even though the element is visible on the page.

---



---

## 20. Mock APIs

---

## **Part I - Core Questions**

## Question 20.6

**How would you decide whether to mock, modify, or observe network traffic in a test?**

### Interview-Style Answer

I would choose the network technique based on the purpose of the test. I would observe network traffic when I need evidence about which requests are sent, wait for a request or response when I need synchronization, mock a response when I need controlled backend behavior, modify a request when I need a small controlled variation, and use real traffic when the scenario needs frontend-backend integration confidence.

In Playwright Java, mocking is useful, but it should not be the default for every test. If everything is mocked, the suite may stop detecting real API contract issues, authentication problems, permission defects, or integration failures.

A balanced strategy uses mocks for controlled frontend states and real backend calls for critical end-to-end business flows.

### Detailed Explanation

Different network techniques solve different testing problems. The decision should depend on what the test is trying to prove.

Use observation when the test needs evidence about browser traffic:

```
page.onRequest(request -> {  
    System.out.println(request.method() + " " + request.url());  
});
```

This is useful during debugging or when validating that a user action sends the expected request.

Use `waitForResponse()` when the test needs synchronization with a specific backend call:

```

Response response = page.waitForResponse(
  res -> res.url().contains("/api/orders")
  && res.status() == 200,
  () -> {
    page.getByRole(
      AriaRole.BUTTON,
      new Page.GetByRoleOptions().setName("Search")
    ).click();
  }
);

PlaywrightAssertions.assertThat(
  page.getByTestId("orders-table")
).isVisible();

```

Use mocking when the frontend behavior needs controlled backend data:

```

page.route("**/api/profile", route -> {
  route.fulfill(new Route.FulfillOptions()
    .setStatus(200)
    .setContentType("application/json")
    .setBody("{\"name\":\"Raj\",\"role\":\"admin\"}"));
});

```

This is useful for testing empty states, error states, permission states, or rare data conditions that are difficult to create in the backend.

Use request modification when the backend should still process the request, but the test needs a controlled change:

```

page.route("**/api/profile", route -> {
  route.resume(new Route.ResumeOptions()
    .setHeaders(Map.of("x-test-run", "true")));
});

```

Here, the backend is still involved. The test only modifies the outgoing request.

Use real traffic when the scenario must prove actual integration, such as login, payment flow, order creation, permission validation, or report generation.

A practical strategy is:

```

Observe      -> understand or debug traffic
Wait         -> synchronize with a network event
Mock         -> control backend response
Modify       -> adjust request while keeping backend real
Real traffic -> validate actual integration

```

## Common Mistake

mocking network responses by default without checking whether the scenario needs real integration confidence. Over-mocking can make

---

tests fast and stable, but it can also hide API contract defects, backend failures, and permission issues.

---

---

## **21. Network**

---

## **Part I - Core Questions**

## Question 21.14

**What is the best practice for handling service workers in Playwright network tests?**

### Interview-Style Answer

The best practice is to block service workers when the test needs Playwright to reliably observe, mock, modify, or intercept network traffic.

Service workers can sit between the page and the network. They may intercept requests, serve cached responses, or mock responses before Playwright's native routing APIs handle them. Because of this, `page.route()` or `browserContext.route()` may appear to miss requests.

In Playwright Java, when testing network interception or API mocking, create the `BrowserContext` with service workers blocked and use Playwright's native routing APIs with specific route patterns.

### Detailed Explanation

Playwright provides network capabilities such as request monitoring, response waiting, request modification, API mocking, request blocking, and route handling through `page.route()` or `browserContext.route()`.

However, service workers can interfere with this because they operate between the browser page and the network. A service worker can:

- Intercept requests
- Return cached responses
- Mock responses
- Modify network behavior
- Prevent some requests from reaching the normal network layer

This is especially relevant when the application uses service-worker-based mocking tools such as Mock Service Worker. In that case, the request may be handled by the service worker before Playwright routing gets the expected control.

For reliable Playwright Java network tests, block service workers at the context level:



```
import com.microsoft.playwright.*;
import com.microsoft.playwright.options.ServiceWorkerPolicy;

BrowserContext context = browser.newContext(
    new Browser.NewContextOptions()
        .setServiceWorkers(ServiceWorkerPolicy.BLOCK)
);

context.route("**/api/products", route -> {
    route.fulfill(new Route.FulfillOptions()
        .setStatus(200)
        .setContentType("application/json")
        .setBody("[{\"name\":\"Laptop\"}]"));
});

Page page = context.newPage();
page.navigate("https://example.com/products");

PlaywrightAssertions.assertThat(
    page.getByText("Laptop")
).isVisible();
```

The route should be registered before navigation or before the action that triggers the request. The route pattern should also be specific enough to avoid intercepting unrelated APIs.

A practical debugging checklist is:

1. Is a service worker active?
2. Is the application using MSW or another service-worker-based mock?
3. Is the response coming from cache?
4. Was the route registered before navigation or action?
5. Does the route pattern match the actual request URL?
6. Is the route registered on the correct Page or BrowserContext?

Blocking service workers makes Playwright network tests more predictable because the request flow is controlled by Playwright's native routing layer instead of being intercepted earlier by a service worker.

## Common Mistake

assuming `page.route()` or `browserContext.route()` is broken when a service worker is actually handling the request first. For network interception and API mocking tests, block service workers unless the scenario specifically needs to test service-worker behavior.

---

## **22. Timeout**

---

## **Part I - Core Questions**

## Question 22.2

### Why should `waitForTimeout()` generally be avoided in Playwright Java tests?

#### Interview-Style Answer

`waitForTimeout()` should generally be avoided because it creates a fixed hard wait that does not understand the real application state. It may slow down fast tests, still fail on slower runs, and hide the actual reason for instability.

In Playwright Java, a better approach is to wait for meaningful conditions such as a visible message, updated table row, changed URL, completed download, specific API response, or enabled button. Playwright's locators, auto-waiting, event waits, and web-first assertions are designed to wait for real readiness instead of guessing with fixed delays.

Use `waitForTimeout()` only for rare debugging or demonstration purposes, not as a normal synchronization strategy.

#### Detailed Explanation

A hard wait pauses the test for a fixed amount of time:

```
page.waitForTimeout(3000);
```

The problem is that the wait is not connected to what the application is doing. If the application becomes ready in 300 milliseconds, the test still wastes 3 seconds. If the application takes 5 seconds in CI/CD, the test still fails after waiting only 3 seconds.

Instead of waiting blindly, the test should wait for the real expected result.

Better approach:

```
page.getByRole(
    AriaRole.BUTTON,
    new Page.GetByRoleOptions().setName("Refresh")
).click();

PlaywrightAssertions.assertThat(
    page.getByText("Order loaded")
).isVisible();
```

Here, the test waits for Order loaded, which is the actual user-visible result.

---

For network-driven flows, wait for the specific response and then assert the UI:

```
Response response = page.waitForResponse(
  res -> res.url().contains("/api/orders")
    && res.status() == 200,
  () -> {
    page.getByRole(
      AriaRole.BUTTON,
      new Page.GetByRoleOptions().setName("Refresh")
    ).click();
  }
);

PlaywrightAssertions.assertThat(
  page.getByText("Order loaded")
).isVisible();
```

For downloads, wait for the download event:

```
Download download = page.waitForDownload(() -> {
  page.getByRole(
    AriaRole.BUTTON,
    new Page.GetByRoleOptions().setName("Export")
  ).click();
});
```

If an action times out, adding `waitForTimeout()` is usually not the right fix. The better debugging approach is to check whether the locator is correct, the element is visible, enabled, stable, receiving events, not covered by an overlay, inside an iframe, or blocked by missing application state.

## Common Mistake

adding `waitForTimeout()` after every flaky action. This makes tests slower and hides the real issue; use locator assertions, event waits, response waits, and user-visible outcome checks instead.

---

---

## **23. Clock**

---

## **Part II - Additional Questions**

### Question 23.14

**What is the difference between `pauseAt`, `fastForward`, `runFor`, and `resume` in Playwright Clock?**

### Interview-Style Answer

In Playwright Clock, these methods are used after installing the clock when the test needs controlled browser-time behavior.

`pauseAt()` freezes time at a specific date and time. `fastForward()` jumps time forward quickly. `runFor()` lets controlled time advance for a specific duration so timers and intervals can execute. `resume()` returns the clock to normal time progression.

They are useful for testing session expiry, countdowns, delayed messages, scheduled UI updates, token expiry, and date-sensitive behavior without using `Thread.sleep()` or `waitForTimeout()`.

### Detailed Explanation

`setFixedTime()` is usually enough when the test only needs a fixed current date or time. But when the application uses timer-based APIs such as `setTimeout()`, `setInterval()`, or scheduled UI updates, install the clock first and then control how time moves.

`pauseAt()` freezes browser time at a specific moment. It is useful when the application must behave as if the current time is a known date or time.

Example use cases:

- Freeze time at midnight
- Verify a festival or year-end banner
- Test date-sensitive UI without depending on today's real date

`fastForward()` advances browser time quickly without waiting in real time. It is useful when the application has delayed behavior and the test only needs to move to the later state.

Example use cases:

- Fast-forward 30 minutes to test session expiry
- Fast-forward 5 seconds to show a delayed notification
- Fast-forward 1 day to test token expiry

`runFor()` runs controlled time for a specific duration. This is useful when timers or intervals need to execute during that period instead of



---

only jumping to a final time.

Example use cases:

- Run the clock for 10 seconds and verify countdown changes
- Run the clock for 1 minute and verify interval-based refresh
- Run scheduled UI updates for a controlled duration

`resume()` returns the page to normal time flow after controlled-time testing is complete.

Simple comparison:

Method	Purpose	Best use case
<code>pauseAt()</code>	Freeze time at a specific moment	Date/time-based UI
<code>fastForward()</code>	Move time forward quickly	Expiry, timeout, delayed behavior
<code>runFor()</code>	Let controlled time run for a duration	Countdown, interval, scheduled updates
<code>resume()</code>	Continue normal time flow	Return page to normal clock behavior

## Common Mistake

using `Thread.sleep()` or `page.waitForTimeout()` to test time-based features. That makes tests slow and flaky; Clock methods should be used to control browser time directly and assert the resulting UI state.

---

---

## **24. Screenshots**

---

## **Part I - Core Questions**

## Question 24.1

### How do you capture screenshots in Playwright Java, and when are they useful?

#### Interview-Style Answer

In Playwright Java, screenshots can be captured using `page.screenshot()` for the full page or current viewport, and `locator.screenshot()` for a specific element. They are useful for debugging failures because they show the visible UI state at the exact point where the test reached.

Screenshots help identify issues such as wrong page navigation, hidden elements, layout problems, overlapping modals, loading overlays, unexpected validation messages, missing data, or environment-specific UI differences. They are especially useful when combined with Trace Viewer, logs, videos, and test failure messages.

#### Detailed Explanation

Screenshots provide visual evidence of what the browser displayed during test execution. When a test fails, the screenshot can quickly show whether the problem is in the locator, application state, test data, environment, or timing.

For example, a click failure may happen because the button is covered by a loading overlay. A text assertion may fail because the user was redirected to the login page. A field may not be visible because the wrong test data loaded. A screenshot helps confirm these conditions visually.

Page screenshot example:

```
page.screenshot(new Page.ScreenshotOptions()
    .setPath(Paths.get("failure-page.png"))
    .setFullPage(true));
```

Element screenshot example:

```
Locator heading = page.getByText("Example Domain");

heading.screenshot(new Locator.ScreenshotOptions()
    .setPath(Paths.get("heading.png")));
```

Use `page.screenshot()` when you want to understand the overall page state, such as current route, visible layout, modal dialogs, banners,

---

tables, or error pages. Use `locator.screenshot()` when you want focused evidence for one component, such as a chart, button, form field, toast message, or table row.

Screenshots are most valuable in these situations:

- Test failure debugging
- Wrong page or unexpected redirect
- Missing or hidden UI element
- Layout or responsive design issue
- Overlay, spinner, modal, or toast blocking interaction
- Visual confirmation of error messages
- Environment-specific UI differences
- Important checkpoints in complex workflows

In a framework, screenshots are usually captured automatically on failure. This gives useful evidence without creating too many artifacts for successful tests. Capturing screenshots after every step can slow down execution and make reports noisy.

For deeper debugging, screenshots should not be used alone. A screenshot shows what was visible, but Trace Viewer can also show actions, snapshots, network calls, console messages, and locator behavior. Together, they make it easier to understand whether the failure came from locator design, actionability, data setup, environment, or actual application behavior.

## Common Mistake

capturing screenshots too early or too often. A useful screenshot should be taken after the application reaches the failure state or an important checkpoint; otherwise, it may not show the real reason for the failure.

---

---

## **25. Videos**

---

## **Part I - Core Questions**

### Question 25.3

**Why must the `BrowserContext` be closed for Playwright Java videos to be saved?**

### Interview-Style Answer

In Playwright Java, recorded videos are finalized and saved only when the related `Page` or `BrowserContext` is closed.

So, when video recording is enabled at the `BrowserContext` level, the framework must close the context during teardown:

```
context.close();
```

Until the context is closed, the video may still be incomplete, not flushed to disk, or unavailable through `page.video().path()`. This is important for CI reports because a missing `context.close()` can result in missing video artifacts even though video recording was configured correctly.

### Detailed Explanation

Video recording is configured when creating a `BrowserContext`:

```
BrowserContext context = browser.newContext(  
    new Browser.NewContextOptions()  
        .setRecordVideoDir(Paths.get("videos/"))  
);
```

All pages created inside this context can produce video recordings:

```
Page page = context.newPage();  
page.navigate("https://example.com");  
page.getByRole(  
    AriaRole.BUTTON,  
    new Page.GetByRoleOptions().setName("Submit")  
).click();
```

However, Playwright does not fully finalize the video while the context is still open. The recording is completed at the end of the page/context lifecycle. That means the video file may not be complete or available until the page or context is closed.

Correct pattern:



```
BrowserContext context = browser.newContext(  
    new Browser.NewContextOptions()  
        .setRecordVideoDir(Paths.get("videos/"))  
);  
  
Page page = context.newPage();  
  
try {  
    page.navigate("https://example.com");  
  
    page.getByRole(  
        AriaRole.BUTTON,  
        new Page.GetByRoleOptions().setName("Submit")  
    ).click();  
  
} finally {  
    context.close();  
}
```

In a JUnit framework, this should usually happen in teardown:

```
@AfterEach  
void tearDown() {  
    if (context != null) {  
        context.close();  
    }  
}
```

In a TestNG framework, the same idea applies in `@AfterMethod`.

If the test needs to attach the video to a report, the path should be read only after the page or context has been closed:

```
Path videoPath = page.video().path();
```

This matters especially in CI/CD because videos are often collected as artifacts after the test run. If the context is not closed properly, the pipeline may upload an empty, incomplete, or missing video file.

## Common Mistake

enabling `setRecordVideoDir()` and assuming the video is immediately available while the context is still open. Always close the `Page` or `BrowserContext` in teardown before reading, attaching, or uploading the video artifact.

---

## **26. Multithreading**

---

## **Part I - Core Questions**

## Question 26.1

### Is Playwright Java thread-safe?

#### Interview-Style Answer

No. Playwright Java is not thread-safe. The `Playwright` instance and objects created from it, such as `Browser`, `BrowserContext`, `Page`, `Locator`, and `Frame`, should be used from the same thread that created the `Playwright` object.

If tests need to run in parallel, the safer design is to avoid sharing Playwright objects across threads. Each test thread or worker should create and own its own Playwright lifecycle, or access must be synchronized so that only one thread calls Playwright methods at a time.

In most automation frameworks, separate ownership per thread is cleaner than synchronization.

#### Detailed Explanation

Playwright Java objects have thread affinity. That means the thread that creates the `Playwright` instance should also call methods on that instance and on the objects created from it.

A safe multi-thread model is:

```
Thread 1 → Playwright 1 → Browser 1 → BrowserContext/Page  
Thread 2 → Playwright 2 → Browser 2 → BrowserContext/Page  
Thread 3 → Playwright 3 → Browser 3 → BrowserContext/Page
```

Example:

```

public class PlaywrightThread extends Thread {
    private final String browserName;

    public PlaywrightThread(String browserName) {
        this.browserName = browserName;
    }

    @Override
    public void run() {
        try (Playwright playwright = Playwright.create()) {
            BrowserType browserType = switch (browserName) {
                case "chromium" -> playwright.chromium();
                case "firefox" -> playwright.firefox();
                case "webkit" -> playwright.webkit();
                default -> throw new IllegalArgumentException(
                    "Unsupported browser: " + browserName
                );
            };

            Browser browser = browserType.launch();
            BrowserContext context = browser.newContext();
            Page page = context.newPage();

            page.navigate("https://playwright.dev/");
            System.out.println(browserName + " title: " + page.title());

            context.close();
            browser.close();
        }
    }
}

```

### Usage:

```

new PlaywrightThread("chromium").start();
new PlaywrightThread("firefox").start();
new PlaywrightThread("webkit").start();

```

Here, each thread creates its own **Playwright**, **Browser**, **BrowserContext**, and **Page**. No Playwright object is shared between threads, so each thread controls its own browser session safely.

A risky pattern is:

```

static Page page;

```

If JUnit or TestNG runs tests in parallel, multiple tests may use the same **Page**. One test may navigate while another test clicks, fills, waits for a response, or closes the context. This can cause wrong-page actions, missed events, race conditions, and flaky CI failures.

If a framework decides to share a Playwright object, it must synchronize access carefully. But synchronization usually reduces parallelism and makes the framework harder to maintain. For practical parallel execution, isolated Playwright ownership per thread or worker is usually the better design.

---

## Common Mistake

assuming Java multithreading makes Playwright objects safe to share. A static shared `Page`, `BrowserContext`, or `Playwright` may work in sequential execution, but parallel tests can interfere with each other and produce random failures.

---

---

## **27. Flakiness**

---

## **Part I - Core Questions**



## Question 27.4

### How do Playwright's auto-waiting and web-first assertions help reduce flaky tests?

#### Interview-Style Answer

Playwright reduces flaky tests by handling timing more intelligently. Auto-waiting makes actions stable by waiting until the target element is ready for interaction, and web-first assertions make validations stable by retrying until the expected UI condition is met.

In Playwright Java, this means a `click()` waits for the button to become visible, stable, enabled, and ready to receive events, while assertions such as `isVisible()`, `hasText()`, or `hasCount()` wait until the application reaches the expected state. Together, they reduce the need for `Thread.sleep()` and make tests more reliable for dynamic web applications.

#### Detailed Explanation

Flaky tests often happen because automation runs faster than the application. The test may try to click a button before it is enabled, fill an input before it is editable, or assert text before the UI has finished updating after an API response.

Examples of timing-related problems include:

- The button is not visible yet.
- The element is still moving because of animation.
- The input field is not enabled or editable yet.
- A loading overlay is still blocking the target.
- The page is updating after an API response.
- The expected success message has not appeared yet.

Playwright handles many of these issues through auto-waiting. Before actions such as `click()`, `fill()`, `check()`, or `selectOption()`, Playwright waits for the element to satisfy required actionability conditions.

For a click, Playwright checks whether the element is:

- Attached to the page
- Visible
- Stable
- Enabled
- Able to receive pointer events

So instead of writing:

```
Thread.sleep(3000);

page.getByRole(
    AriaRole.BUTTON,
    new Page.GetByRoleOptions().setName("Submit")
).click();
```

we can usually write:

```
page.getByRole(
    AriaRole.BUTTON,
    new Page.GetByRoleOptions().setName("Submit")
).click();
```

Playwright will wait for the button to become actionable before clicking it. This is better than a hard wait because a hard wait does not know the real UI condition. Three seconds may be too much on a fast run and not enough on a slow CI run.

Web-first assertions help on the validation side. They do not check the condition only once. They keep retrying until the expected condition becomes true or the timeout is reached.

Example:

```
page.getByRole(
    AriaRole.BUTTON,
    new Page.GetByRoleOptions().setName("Submit")
).click();

PlaywrightAssertions.assertThat(
    page.getByText("Submitted successfully")
).isVisible();
```

In this example, Playwright waits for the **Submit** button to be ready before clicking. Then the assertion waits until the success message becomes visible. This is much more stable than clicking and immediately checking the DOM once.

Auto-waiting and web-first assertions solve different parts of the same problem:

- Auto-waiting stabilizes actions.
- Web-first assertions stabilize validations.
- Locators re-resolve elements when actions or assertions run.
- Hard waits become unnecessary in most normal UI flows.

However, they do not replace business-level waiting completely. If the test needs to wait for a specific API response, popup, download, navigation, or backend-driven state change, the framework should wait for that specific event or assert the final user-visible result.

---

## Common Mistake

adding `Thread.sleep()` everywhere to fix flaky tests. A better Playwright approach is to use reliable locators, allow auto-waiting to handle action readiness, and use web-first assertions to wait for the expected user-visible state.

---

---

## **28. Performance**

---

## **Part I - Core Questions**

## Question 28.4

### How do you detect slow API calls during UI execution?

#### Interview-Style Answer

I would detect slow API calls by capturing request start time and response completion time during the UI flow, then reporting important backend endpoints that exceed agreed thresholds. This helps separate API slowness from frontend rendering, locator, or synchronization issues.

In Playwright Java, I would focus on business APIs such as dashboard data, search results, checkout, reports, or profile APIs, not static assets or unrelated third-party scripts.

#### Detailed Explanation

Slow API calls often appear as slow UI behavior. A dashboard may not render because `/api/dashboard` is slow, or a search result may appear late because `/api/search` is delayed. Playwright can observe network traffic while the user flow is running, which helps identify whether the delay is coming from the backend or the frontend.

Useful data to capture includes:

1. Endpoint URL.
2. HTTP method.
3. Status code.
4. Request start time.
5. Response completion time.
6. Duration in milliseconds.
7. Business flow or test name.
8. Whether the slow API delayed the visible UI result.
9. Threshold exceeded or not.

Example:

```

Map<String, Long> requestStartTimes = new ConcurrentHashMap<>();

page.onRequest(request -> {
    if (request.url().contains("/api/")) {
        requestStartTimes.put(
            request.method() + " " + request.url(),
            System.currentTimeMillis()
        );
    }
});

page.onResponse(response -> {
    String key = response.request().method() + " " + response.url();

    if (response.url().contains("/api/")
        && requestStartTimes.containsKey(key)) {

        long durationMs = System.currentTimeMillis()
            - requestStartTimes.get(key);

        if (durationMs > 2000) {
            System.out.println(
                "Slow API detected: "
                + response.status() + " "
                + key + " took "
                + durationMs + " ms"
            );
        }
    }
});

page.navigate("https://example.com/dashboard");

PlaywrightAssertions.assertThat(
    page.getByText("Dashboard")
).isVisible();

```

For stronger validation, I would connect the API timing to the visible UI outcome. For example, if `/api/dashboard` takes 4 seconds, the test should also check whether the dashboard cards appeared late or whether the loading spinner stayed visible too long.

Example:

```

long start = System.currentTimeMillis();

Response response = page.waitForResponse(
    res -> res.url().contains("/api/dashboard")
    && res.status() == 200,
    () -> page.navigate("https://example.com/dashboard")
);

long apiDurationMs = System.currentTimeMillis() - start;

PlaywrightAssertions.assertThat(
    page.getTestId("dashboard-card")
).isVisible();

Assertions.assertTrue(
    apiDurationMs <= 2000,
    "Dashboard API was slow: " + apiDurationMs + " ms"
);

```

---

In a real framework, slow API details should be added to test logs or CI reports with the browser name, environment, test name, endpoint, status code, and duration. This makes performance problems easier to discuss with backend, frontend, and DevOps teams.

### **Common Mistake**

Timing static assets, images, fonts, analytics calls, or third-party scripts and reporting them as product API performance issues, instead of focusing on business-critical backend APIs that affect the user-visible UI state.

---



---

## 29. CodeGen

---

## **Part I - Core Questions**

## Question 29.12

### How would you refactor Codegen output into Page Objects?

#### Interview-Style Answer

I would refactor Codegen output into Page Objects by identifying page-specific locators and actions from the generated script, moving them into focused Page Object methods, and keeping the test class readable at the business-flow level.

In Playwright Java, the Page Object should hide locator details but expose meaningful methods such as `loginAs()`, `searchProduct()`, `createOrder()`, or `shouldBeDisplayed()`. This improves maintainability because locator changes are handled in one place, repeated flows are reusable, and tests become easier to review and debug.

#### Detailed Explanation

Raw Codegen output usually places all recorded actions directly inside one test method. That may work for a simple draft, but it becomes hard to maintain when the same login, search, checkout, or navigation flow is repeated across many tests.

Raw generated test:

```
page.getByLabel("Username").fill("admin");
page.getByLabel("Password").fill("password");
page.getByRole(
    AriaRole.BUTTON,
    new Page.GetByRoleOptions().setName("Login")
).click();
```

A better approach is to move the login behavior into a Page Object:

```
public class LoginPage {
    private final Page page;

    public LoginPage(Page page) {
        this.page = page;
    }

    public void loginAs(String username, String password) {
        page.getByLabel("Username").fill(username);
        page.getByLabel("Password").fill(password);

        page.getByRole(
            AriaRole.BUTTON,
            new Page.GetByRoleOptions().setName("Login")
        ).click();
    }
}
```

The test then focuses on the scenario instead of low-level UI actions:

```
loginPage.loginAs(adminUser, password);
dashboardPage.shouldBeDisplayed();
```

The Page Object can also include assertion methods for page-level validation:

```
public class DashboardPage {
    private final Page page;

    public DashboardPage(Page page) {
        this.page = page;
    }

    public void shouldBeDisplayed() {
        PlaywrightAssertions.assertThat(
            page.getByRole(
                AriaRole.HEADING,
                new Page.GetByRoleOptions().setName("Dashboard")
            )
        ).isVisible();
    }
}
```

While refactoring, I would avoid creating one huge Page Object that contains every action on the application. Each Page Object or component object should have a clear responsibility. Reusable UI parts such as header, sidebar, product card, modal, or date picker can be modeled as separate components with scoped locators.

This structure makes failures easier to diagnose. If login fails, the issue is isolated to [LoginPage](#). If dashboard validation fails, the issue is isolated to [DashboardPage](#). It also reduces duplication and makes locator updates safer when the UI changes.

## Common Mistake

---

keeping all Codegen-generated actions in one long test method or moving everything into one bloated Page Object, instead of creating focused Page Objects and reusable component methods with clear responsibilities.

---

---

## **30. Trace viewer**

---

## **Part II - Additional Questions**

## Question 30.20

**How do you open a Playwright trace in the browser using `trace.playwright.dev`?**

### Interview-Style Answer

You can open a Playwright trace in a browser by visiting the Playwright Trace Viewer website:

```
https://trace.playwright.dev
```

Then upload the saved `trace.zip` file using drag-and-drop or the file selection option.

The browser-based Trace Viewer loads the trace and allows you to inspect actions, snapshots, source locations, logs, console messages, network activity, and errors without running the Playwright CLI locally.

### Detailed Explanation

Playwright provides a web-based Trace Viewer at:

```
https://trace.playwright.dev
```

This is useful when you want to analyze a trace quickly without installing anything additional or running the `show-trace` CLI command.

Typical workflow:

1. Record a trace during test execution.
2. Save the trace as `trace.zip`.
3. Open `https://trace.playwright.dev`.
4. Upload or drag-and-drop `trace.zip`.
5. Explore the recorded execution.

Once the trace is loaded, the browser viewer provides the same debugging capabilities available in the desktop Trace Viewer, including:

- Test actions and timelines
- Locators used by each action
- Action duration and timing information
- Before, Action, and After snapshots
- Screenshots and film strip
- Source code locations
- Call details
- Playwright logs
- Browser console messages
- Network requests and responses
- Error information and stack traces
- Environment metadata



---

This is particularly useful for CI/CD troubleshooting. For example, a failed pipeline can publish `trace.zip` as a build artifact. A developer can download the artifact and inspect the complete execution directly in the browser.

Another useful option is opening a remotely hosted trace through a URL:

```
https://trace.playwright.dev/?trace=https://example.com/trace.zip
```

This can simplify collaboration because team members can open the trace directly from a shared artifact location. The trace URL must be publicly accessible or accessible to the user's browser, and CORS restrictions may apply depending on the hosting platform.

The browser Trace Viewer is especially valuable when investigating:

- CI-only failures
- Timing-related issues
- Locator problems
- Navigation failures
- Unexpected UI states
- Intermittent test failures

Because the viewer combines actions, snapshots, logs, network activity, and source information in one place, it often provides enough evidence to identify the root cause without rerunning the test.

## Common Mistake

assuming that opening a trace in `trace.playwright.dev` sends the trace to a Playwright server. The trace is processed locally in the browser, but traces may still contain sensitive information such as URLs, tokens, screenshots, user data, or application content, so trace files should be shared and stored according to the team's security policies.

---

---

## **31. Extensibility**

---

## **Part I - Core Questions**

## Question 31.1

### How do you register a custom selector engine in Playwright Java?

#### Interview-Style Answer

In Playwright Java, a custom selector engine is registered using `playwright.selectors().register()` before creating the `Page`.

```
playwright.selectors().register("tag", createTagNameEngine);
```

After registration, the engine can be used through its selector prefix:

```
Locator button = page.locator("tag=button");  
button.click();
```

The selector name, such as `tag`, becomes the prefix. The selector value after `=` is passed to the custom engine's `query()` and `queryAll()` functions.

#### Detailed Explanation

A custom selector engine defines custom logic for locating elements. The engine is usually written as a JavaScript object with `query(root, selector)` and `queryAll(root, selector)` functions.

Example:

```
String createTagNameEngine = "{\n" +  
  "  query(root, selector) {\n" +  
    "    return root.querySelector(selector);\n" +  
  "  },\n" +  
  "  queryAll(root, selector) {\n" +  
    "    return Array.from(root.querySelectorAll(selector));\n" +  
  "  }\n" +  
  "}";  
  
playwright.selectors().register("tag", createTagNameEngine);  
  
Browser browser = playwright.chromium().launch();  
Page page = browser.newPage();  
  
Locator button = page.locator("tag=button");  
button.click();
```

Here, `"tag"` is the custom selector engine name. When the test uses:

```
page.locator("tag=button");
```

Playwright sends `button` as the selector value to the custom engine.

---

The engine then resolves the element using:

```
root.querySelector(selector)
```

or resolves all matching elements using:

```
root.querySelectorAll(selector)
```

The `root` parameter is important because it allows the custom selector engine to work correctly with locator scoping and chaining.

For example:

```
page.locator("#login-form")
  .locator("tag=input")
  .first()
  .fill("admin");
```

In this case, the custom selector should search only inside `#login-form`, not across the whole document.

Custom selector engines can also be registered with options such as `contentScript: true` when the engine should run in a safer isolated content-script environment:

```
playwright.selectors().register(
  "tag",
  createTagNameEngine,
  new Selectors.RegisterOptions().setContentScript(true)
);
```

The correct order is:

1. Create Playwright
2. Register the selector engine
3. Launch browser
4. Create BrowserContext/Page
5. Use the custom selector

In real frameworks, this registration usually belongs in central setup code, not inside individual test methods, so all tests use the same selector behavior consistently.

## Common Mistake

registering the custom selector engine after creating pages or using the selector prefix. The engine should be registered during framework setup before `BrowserContext` or `Page` creation, and it should only be introduced when built-in locators such as `getByRole()`, `getByLabel()`, `getByText()`, or `getByTestId()` cannot solve the problem cleanly.

---

---

## **Book 2 - Automation Scenarios & Debugging**

---

## **1. Automation scenarios**

---

## **Part I - Core Questions**



## Question 1.35

### How do you validate toast notifications in Playwright?

#### Interview-Style Answer

I would validate toast notifications by locating the toast message using a stable role, text content, or test ID, then asserting the exact notification text and visible state. If the toast is expected to auto-close, I would also assert that it disappears after the expected duration.

In Playwright Java, this ensures the test verifies both the correct user-visible feedback and its lifecycle. This approach avoids flakiness when multiple toasts may appear or overlap.

#### Detailed Explanation

Toast notifications are transient and may appear dynamically after user actions. Proper validation should ensure that the correct toast is matched and no unrelated notification is accidentally captured.

Key checks:

1. Correct toast text appears for the action.
2. Correct toast type or status (success, error, warning) if user-visible.
3. Toast is triggered by the intended user action.
4. Toast disappears automatically if expected.
5. No old or unrelated toast is matched.
6. Multiple simultaneous toasts are handled correctly.

Example:

```
// Trigger the action that shows the toast
page.getByRole(
    AriaRole.BUTTON,
    new Page.GetByRoleOptions().setName("Submit")
).click();

// Locate the toast notification
Locator toast = page.getByRole(AriaRole.STATUS)
    .filter(new Locator.FilterOptions().setHasText("Submitted
        successfully"));

// Assert the toast is visible
PlaywrightAssertions.assertThat(toast).isVisible();

// Optional: Assert toast disappears if auto-close is expected
PlaywrightAssertions.assertThat(toast).isHidden();
```

For multiple toasts:

---

```
List<Locator> toasts = page.locator(".toast").all();
for (Locator t : toasts) {
    System.out.println(t.innerText());
}
```

## Common Mistake

Selecting the first toast on the page without scoping to the current user action, which can cause validation of an old or unrelated notification and make the test flaky.

---

## Question 1.46

### How do you wait for an element to disappear after submitting a form?

#### Interview-Style Answer

I would use `assertThat(locator).isHidden()` if the element should remain in the DOM but become invisible, or `assertThat(locator).isDetached()` if the element should be removed from the DOM entirely. After the element disappears, I would also validate the final expected outcome, such as a success message, updated record, or refreshed table.

This ensures the test proves both that the transient element is gone and that the user-visible result is correct.

#### Detailed Explanation

UI elements can disappear in two ways:

- Hidden: The element stays in the DOM but is visually hidden (e.g., a loading spinner fading out).
- Detached: The element is removed from the DOM (e.g., a modal or temporary notification).

The assertion you choose depends on the application's behavior.

Example for a hidden spinner:

```
Locator spinner = page.getByTestId("saving-spinner");

page.getByRole(
    AriaRole.BUTTON,
    new Page.GetByRoleOptions().setName("Submit")
).click();

// Wait for the spinner to become hidden
PlaywrightAssertions.assertThat(spinner).isHidden();

// Validate final result
PlaywrightAssertions.assertThat(
    page.getByText("Saved successfully")
).isVisible();
```

Example for a detached modal:

```
Locator modal = page.getByRole(  
    AriaRole.DIALOG,  
    new Page.GetByRoleOptions().setName("Edit Customer")  
);  
  
modal.getByRole(AriaRole.BUTTON, new  
    Locator.GetByRoleOptions().setName("Save")).click();  
  
// Wait for the modal to be removed from the DOM  
PlaywrightAssertions.assertThat(modal).isDetached();  
  
// Validate saved changes  
PlaywrightAssertions.assertThat(page.getByText("Customer updated  
    successfully")).isVisible();
```

### Key points:

- Choose ``isHidden()`` vs ``isDetached()`` based on how the element disappears.
- Always validate the final visible outcome, not only the transient element.
- Use web-first assertions instead of fixed sleeps for stability.

## Common Mistake

Waiting only for the spinner or modal to disappear without checking that the actual operation completed successfully, which can result in false-positive tests.

## Question 1.53

### How do you test progress indicators during file upload?

#### Interview-Style Answer

I would test upload progress by making the upload take long enough to observe the intermediate state, then assert the full state transition: progress indicator appears after upload starts, remains visible while upload is in progress, disappears after completion, and the final success state is shown.

I would not validate exact timing or exact percentage unless the product specifically requires it. The reliable validation is that the user receives correct upload feedback and the UI does not mark the upload as complete too early.

#### Detailed Explanation

Progress indicators during file upload are important when users need feedback for large files, slow networks, or document-heavy workflows such as invoice upload, resume upload, profile photo upload, or report attachment upload.

A good test should not check only the final success message. It should validate the upload lifecycle:

1. File upload starts.
2. Progress indicator appears.
3. Upload is not shown as complete too early.
4. Progress indicator disappears after completion.
5. Uploaded file name, success message, or attachment row appears.
6. Progress indicator does not remain stuck.

To make the progress state testable, the upload request can be intercepted and delayed in a controlled way:

```
page.route("**/api/upload", route -> {
  try {
    Thread.sleep(1000);

    route.fulfill(new Route.FulfillOptions()
      .setStatus(200)
      .setContentType("application/json")
      .setBody("{\"status\":\"uploaded\",\"fileName\":\"invoice.pdf\"}"));
  } catch (InterruptedException e) {
    Thread.currentThread().interrupt();
    route.abort();
  }
});
```

---

Then trigger the file upload:

```
page.locator("input[type='file']")
    .setInputFiles(Paths.get("src/test/resources/invoice.pdf"));
```

Now assert the progress indicator and final result:

```
Locator progress = page.locator("[role='progressbar']");

PlaywrightAssertions.assertThat(progress).isVisible();

PlaywrightAssertions.assertThat(
    page.getByText("Upload complete")
).isVisible();

PlaywrightAssertions.assertThat(progress).isHidden();

PlaywrightAssertions.assertThat(
    page.getByText("invoice.pdf")
).isVisible();
```

If the product shows percentage progress, the test can assert broad meaningful behavior such as progress appears or reaches completion, but it should avoid brittle millisecond-based checks. Network speed, browser behavior, CI load, and file size can affect exact timing.

Failure behavior should be tested separately by returning an error response and asserting a visible error message, retry option, or failed upload state. The success test should prove the normal transition; the failure test should prove the error path.

## Common Mistake

Validating only the final “Upload complete” message while missing broken progress behavior, such as the progress bar never appearing, disappearing too early, staying stuck after completion, or showing success before the upload response is actually handled.

---

## Question 1.56

### How do you test drag-and-drop file upload areas?

#### Interview-Style Answer

If the drag-and-drop upload area is backed by a real file input, I would prefer `setInputFiles()` because it is stable, avoids native OS dialogs, and validates the actual file-upload path used by the application.

If the product specifically requires drag-and-drop behavior, I would separately validate the drop-zone UI behavior, such as drag-over styling, accepted file feedback, rejected file messages, and whether the dropped file appears in the upload list. The main workflow should still assert the user-visible upload result, not only that a file was attached.

#### Detailed Explanation

Many drag-and-drop upload components are visually custom, but internally they still use a hidden `<input type="file">`. In that case, the most reliable Playwright Java approach is to set the file directly on the input.

```
page.locator("input[type='file']")
    .setInputFiles(Paths.get("src/test/resources/profile.png"));

PlaywrightAssertions.assertThat(
    page.getByText("profile.png")
).isVisible();

PlaywrightAssertions.assertThat(
    page.getByText("Upload successful")
).isVisible();
```

This avoids OS-level file picker or physical drag-and-drop automation, which is not reliable in browser automation. The test should then validate the business behavior: the file name appears, upload starts, validation passes, and the final success state is shown.

A stronger drag-and-drop upload test should cover:

- Valid file upload
- Uploaded filename displayed
- File type validation
- File size validation
- Remove-file option
- Upload progress or processing state if applicable
- Upload success or error message

For example, to validate file-type rejection:

---

```
page.locator("input[type='file']")
    .setInputFiles(Paths.get("src/test/resources/invalid.exe"));

PlaywrightAssertions.assertThat(
    page.getByText("Only PNG, JPG, and PDF files are allowed")
).isVisible();
```

If the requirement is specifically to test the drop-zone behavior, then the test may need to simulate lower-level drag-and-drop events or use a helper that creates a [DataTransfer](#) object in the browser context. That should be used only when the visual drop behavior itself is the product requirement, because it is more complex and less readable than [setInputFiles\(\)](#).

The final assertion should always prove the user outcome: file accepted, file rejected, upload completed, upload failed, or file removed. Testing only the drag gesture is not enough.

## Common Mistake

Trying to automate the operating-system drag-and-drop action or file chooser when the upload area already has an underlying file input that can be tested more reliably with [setInputFiles\(\)](#).

---



## Question 1.68

### How would you test a form that saves data asynchronously?

#### Interview-Style Answer

I would test an asynchronously saved form by submitting the form and waiting for the actual user-visible save result instead of using `Thread.sleep()`.

If the save depends on an API call, I may use `waitForResponse()` to confirm the relevant request completed successfully. But the final validation should still be based on the UI result, such as a success toast, updated record, saved status, enabled button, or refreshed data shown to the user.

#### Detailed Explanation

Asynchronous saving is common in modern web applications. After clicking Save, the application may show a spinner, disable the Save button, send an API request, update the UI after the response, and then show a success message.

A reliable test should validate this full save behavior:

```
Response response = page.waitForResponse(
    res -> res.url().contains("/api/customers")
    && res.status() == 200,
    () -> {
        page.getByRole(
            AriaRole.BUTTON,
            new Page.GetByRoleOptions().setName("Save")
        ).click();
    }
);
```

After the response, assert the visible result:

```
PlaywrightAssertions.assertThat(
    page.getByText("Saved successfully")
).isVisible();

PlaywrightAssertions.assertThat(
    page.getByText("Ramesh Kumar")
).isVisible();
```

If the application shows a saving state, the test can validate that the user gets proper feedback during the async operation:

```
Locator saveButton = page.getByRole(
    AriaRole.BUTTON,
    new Page.GetByRoleOptions().setName("Save")
);

saveButton.click();

PlaywrightAssertions.assertThat(
    page.getByText("Saving...")
).isVisible();

PlaywrightAssertions.assertThat(
    page.getByText("Saved successfully")
).isVisible();

PlaywrightAssertions.assertThat(saveButton).isEnabled();
```

The test should also confirm that no error message remains after a successful save:

```
PlaywrightAssertions.assertThat(
    page.getByText("Unable to save")
).not().isVisible();
```

Good async save validation includes:

1. Save action is triggered.
2. Saving or loading state appears if required.
3. Relevant API response succeeds if the test is checking the network flow.
4. Success message or saved status appears.
5. Updated data is visible.
6. Save button returns to the expected state.
7. No stale error message remains.

For failure scenarios, I would test separately by mocking or controlling the failed response and asserting the visible error message, retry option, or unsaved state.

## Common Mistake

Using `Thread.sleep()` after clicking Save instead of waiting for the real save condition, such as the API response, success toast, updated record, or Save button returning to its normal state.

## Question 1.83

### How do you test whether a modal traps keyboard focus correctly?

#### Interview-Style Answer

To test whether a modal traps keyboard focus correctly, I would open the modal using a normal user action, navigate with the keyboard, and assert that focus stays inside the modal until it is closed. This validates that keyboard users cannot tab into background page content while the modal is active.

In Playwright Java, I would use role-based locators to open the dialog, assert that the dialog is visible, press `Tab` or `Shift+Tab`, and verify that `document.activeElement` remains inside the element with `role="dialog"`. I would also verify that closing the modal using `Escape` or the close button returns focus to the original trigger when that is the expected behavior.

#### Detailed Explanation

A modal dialog should trap keyboard focus while it is open. This means that when the user presses `Tab`, focus should move only between interactive elements inside the modal. It should not move to links, buttons, or form fields behind the modal because that background content is visually unavailable or inactive.

Example:

```
Locator openDialog = page.getByRole(
    AriaRole.BUTTON,
    new Page.GetByRoleOptions().setName("Open dialog")
);

openDialog.click();

Locator dialog = page.getByRole(AriaRole.DIALOG);

PlaywrightAssertions.assertThat(dialog).isVisible();

page.keyboard().press("Tab");

Boolean focusInsideDialog = (Boolean) page.evaluate(
    "()" => document.activeElement.closest('[role=dialog]') !== null"
);

Assertions.assertTrue(focusInsideDialog);
```

This verifies that after keyboard navigation, the focused element is still inside the modal.

---

For stronger validation, the test can press **Tab** multiple times to ensure focus cycles inside the modal and does not escape to the background page:

```
for (int i = 0; i < 5; i++) {
    page.keyboard().press("Tab");

    Boolean isFocusInsideDialog = (Boolean) page.evaluate(
        "() => document.activeElement.closest('[role=dialog]') !== null"
    );

    Assertions.assertTrue(isFocusInsideDialog);
}
```

The test should also validate the closing behavior if the product supports closing with **Escape**:

```
page.keyboard().press("Escape");

PlaywrightAssertions.assertThat(dialog).isHidden();

Assertions.assertTrue(openDialog.isFocused());
```

This confirms that the modal closed and focus returned to the original trigger. If the application intentionally returns focus to another logical element, that behavior should be documented and tested.

This type of test is important because a modal that is visually correct can still be inaccessible if keyboard focus escapes behind it. A user relying on the keyboard or a screen reader may lose context or interact with inactive background controls.

## Common Mistake

checking only that the modal is visible. A complete modal accessibility test should also verify keyboard focus trapping while the modal is open and correct focus return after the modal is closed.

---

## Question 1.86

**How would you validate that a modal closes after saving?**

### Interview-Style Answer

I would perform the save action inside the modal, then assert that the modal is no longer visible. I would also validate that the underlying page reflects the saved changes, such as updated text, table values, or success messages, ensuring the save action truly completed.

This approach confirms both the modal's disappearance and the business outcome, making the test meaningful and robust.

### Detailed Explanation

Simply checking that the modal is hidden is insufficient because the user-visible result may not have been applied. A complete validation ensures the operation succeeded and the page updated correctly.

Example:

```
Locator modal = page.getByRole(
    AriaRole.DIALOG,
    new Page.GetByRoleOptions().setName("Edit Customer")
);

// Fill a field
modal.getByLabel("Customer Name").fill("Ramesh Kumar");

// Click Save inside the modal
modal.getByRole(
    AriaRole.BUTTON,
    new Locator.GetByRoleOptions().setName("Save")
).click();

// Assert the modal is closed
PlaywrightAssertions.assertThat(modal).isHidden();

// Assert the success message is visible
PlaywrightAssertions.assertThat(
    page.getByText("Customer updated successfully")
).isVisible();

// Assert the updated value is displayed on the page
PlaywrightAssertions.assertThat(
    page.getByText("Ramesh Kumar")
).isVisible();
```

Good validations include:

- 
- Modal is no longer visible after save
  - Success message or toast appears
  - Updated data is reflected in the UI
  - Any dependent components refresh correctly
  - Focus and accessibility state return appropriately

## Common Mistake

Asserting only that the modal disappeared without verifying that the save action updated the underlying page or data correctly.

---

## Question 1.110

### How do you switch between multiple pages in Playwright Java?

#### Interview-Style Answer

In Playwright Java, you do not switch pages using Selenium-style window handles. You keep explicit [Page](#) references and interact with the correct [Page](#) object directly.

When a new tab or popup opens, Playwright returns a new [Page](#) object through methods such as `page.waitForPopup()` or `browserContext.waitForPage()`. The original page remains available through its existing reference, so the test can move between pages by using the appropriate [Page](#) variable.

#### Detailed Explanation

Playwright uses a page-reference model. Each browser tab or popup is represented as a [Page](#) object. Instead of switching by index or window handle, the test stores the returned page reference and performs actions on that object.

Example:

```
Page originalPage = page;

Page popup = originalPage.waitForPopup(() -> {
    originalPage.getByText("Open Details").click();
});

PlaywrightAssertions.assertThat(
    popup.getByText("Details")
).isVisible();

popup.close();

PlaywrightAssertions.assertThat(
    originalPage.getByText("Dashboard")
).isVisible();
```

In this example, [originalPage](#) represents the starting page, and [popup](#) represents the newly opened tab or popup. After validating the popup, the test closes it and continues using the original page reference.

For context-level page creation, `browserContext.waitForPage()` can also be used:

---

```
Page newPage = context.waitForPage() -> {
  page.getByRole(
    AriaRole.LINK,
    new Page.GetByRoleOptions().setName("Open Report")
  ).click();
});

PlaywrightAssertions.assertThat(
  newPage.getByRole(
    AriaRole.HEADING,
    new Page.GetByRoleOptions().setName("Report")
  )
).isVisible();
```

This approach is clearer and safer than switching by index because each variable describes the page being controlled. It also avoids accidentally performing actions on the wrong tab.

A good test should validate something meaningful on each page, such as URL, title, heading, or important content:

```
PlaywrightAssertions.assertThat(newPage).hasURL(Pattern.compile(".*report.*"
));
```

If the popup is no longer needed, close it to keep the test clean and avoid later confusion.

## Common Mistake

losing the original page reference or continuing actions on the wrong [Page](#) object. In Playwright Java, reliable multi-page handling depends on storing clear page references and interacting with the correct page directly.

---



## Question 1.115

### How would you test role-based access using Playwright Java?

#### Interview-Style Answer

I would test role-based access by logging in as each role or loading a role-specific storage-state file, then validating both allowed and restricted behavior.

For each role, I would check visible permissions, hidden or disabled restricted actions, blocked direct URL access, and the expected business outcome. For example, an Admin may see User Management and create users, while a Viewer should not see the Create User button and should be redirected or shown an access-denied message when opening the User Management URL directly.

#### Detailed Explanation

Role-based access testing should cover both positive and negative permissions. Testing only the Admin role is not enough because most access-control defects appear in restricted roles.

A practical setup uses separate storage-state files or login flows for each role:

```
BrowserContext adminContext = browser.newContext(  
    new Browser.NewContextOptions()  
        .setStorageStatePath(Paths.get("auth/admin.json"))  
);  
  
BrowserContext viewerContext = browser.newContext(  
    new Browser.NewContextOptions()  
        .setStorageStatePath(Paths.get("auth/viewer.json"))  
);  
  
Page adminPage = adminContext.newPage();  
Page viewerPage = viewerContext.newPage();
```

For the Admin role, the test should validate allowed UI and business behavior:

```

adminPage.navigate(baseUrl + "/users");

PlaywrightAssertions.assertThat(
    adminPage.getByRole(
        AriaRole.BUTTON,
        new Page.GetByRoleOptions().setName("Create User")
    )
).isVisible();

adminPage.getByRole(
    AriaRole.BUTTON,
    new Page.GetByRoleOptions().setName("Create User")
).click();

PlaywrightAssertions.assertThat(
    adminPage.getByRole(
        AriaRole.DIALOG,
        new Page.GetByRoleOptions().setName("Create User")
    )
).isVisible();

```

For the Viewer role, the same restricted action should be hidden, disabled, or unavailable according to the product rule:

```

viewerPage.navigate(baseUrl + "/users");

PlaywrightAssertions.assertThat(
    viewerPage.getByRole(
        AriaRole.BUTTON,
        new Page.GetByRoleOptions().setName("Create User")
    )
).isHidden();

```

The test should also validate direct URL access because hiding a menu item does not prove access control. A restricted user should not be able to open protected pages directly:

```

viewerPage.navigate(baseUrl + "/users/create");

PlaywrightAssertions.assertThat(
    viewerPage.getByText("Access denied")
).isVisible();

```

For record-level permissions, locators should be scoped to the correct row or card. For example, a Manager may edit assigned records but not other users' records:

```

Locator row = viewerPage.getByRole(
    AriaRole.ROW,
    new Page.GetByRoleOptions().setName("Request 1001 Approved")
);

PlaywrightAssertions.assertThat(
    row.getByRole(
        AriaRole.BUTTON,
        new Locator.GetByRoleOptions().setName("Edit")
    )
).isHidden();

```

---

Role-based UI testing should not be treated as complete security testing. It validates what the user sees and can do through the UI, but backend authorization should also be tested separately to ensure restricted users cannot perform protected actions by calling APIs directly.

### **Common Mistake**

Testing only the Admin happy path and assuming role-based access works, while missing restricted-role checks for hidden actions, disabled controls, direct URL access, and record-level permissions.

---

## Question 1.116

**How would you test access to a protected page without login?**

### Interview-Style Answer

I would test protected-page access by opening the protected URL in a fresh unauthenticated `BrowserContext` and asserting that the user is redirected to the login page or shown an access-denied message.

The test should also verify that protected content is not visible, no sensitive data is exposed, and the original requested URL is preserved if the product supports post-login redirect. Using a fresh context is important because it avoids accidentally reusing cookies, tokens, local storage, or storage state from a previously logged-in test.

### Detailed Explanation

Protected-page testing validates that unauthenticated users cannot access restricted application areas by directly typing the URL. This is different from normal logged-in navigation because it checks the security boundary before authentication.

A reliable test should start with a fresh browser context:

```
BrowserContext context = browser.newContext();
Page page = context.newPage();

page.navigate(baseUrl + "/admin/users");

PlaywrightAssertions.assertThat(page)
    .hasURL(Pattern.compile(".*login"));

PlaywrightAssertions.assertThat(
    page.getByText("Please sign in")
).isVisible();
```

The test should also assert that protected content is not visible:

```
PlaywrightAssertions.assertThat(
    page.getByRole(
        AriaRole.HEADING,
        new Page.GetByRoleOptions().setName("User Management")
    )
).not().isVisible();
```

If the application preserves the originally requested URL for post-login redirect, validate that behavior too. For example, the login page may contain a `redirect` query parameter:

---

```
PlaywrightAssertions.assertThat(page)
    .hasURL(Pattern.compile(".*login.*redirect=.*admin/users.*"));
```

For applications that show an access-denied page instead of redirecting to login, the assertion should match the product rule:

```
PlaywrightAssertions.assertThat(
    page.getByText("Access denied")
).isVisible();
```

A strong protected-route test should be isolated and parallel-safe. It should not reuse an authenticated [BrowserContext](#), role-specific storage state, or a page from a previous test. Otherwise, the test may pass or fail for the wrong reason because old authentication data is still present.

Good checks include:

1. Protected content is not visible.
2. User is redirected to login or shown access denied.
3. Original requested URL is preserved if expected.
4. No sensitive data is shown before redirect.
5. The test uses a fresh unauthenticated BrowserContext.

## Common Mistake

Testing protected pages only after login and missing direct unauthenticated URL access, or accidentally reusing an authenticated context so the test does not truly verify access without login.

---

## Question 1.119

### How do you test forbidden access scenarios for role-based users?

#### Interview-Style Answer

I would test forbidden access by logging in as a valid but lower-privileged user, then trying to access a restricted route or action.

Forbidden access is different from unauthenticated access. The user is already authenticated, but does not have the required permission. So the test should validate that the user stays logged in, restricted menus or actions are hidden or disabled as expected, direct URL access is blocked, and the application shows an access-denied or forbidden message without rendering restricted content.

#### Detailed Explanation

Forbidden access scenarios are common in role-based applications. For example, a Viewer may be allowed to view reports but not manage users. A normal user may access the dashboard but not admin settings. A Manager may edit assigned records but not records owned by another team.

A good test should use a role-specific storage state or login flow for the lower-privileged user:

```
BrowserContext viewerContext = browser.newContext(  
    new Browser.NewContextOptions()  
        .setStorageStatePath(Paths.get("auth/viewer.json"))  
);  
  
Page viewerPage = viewerContext.newPage();
```

First, the test can confirm that the user is authenticated:

```
viewerPage.navigate("https://example.com/dashboard");  
  
PlaywrightAssertions.assertThat(  
    viewerPage.getByText("Dashboard")  
).isVisible();
```

Then try to open the restricted route directly:

---

```
viewerPage.navigate("https://example.com/admin/users");

PlaywrightAssertions.assertThat(viewerPage)
    .hasURL(Pattern.compile(".*forbidden|.*access-denied"));

PlaywrightAssertions.assertThat(
    viewerPage.getByText("Access denied")
).isVisible();
```

The test should also confirm that restricted content is not rendered:

```
PlaywrightAssertions.assertThat(
    viewerPage.getByRole(
        AriaRole.HEADING,
        new Page.GetByRoleOptions().setName("User Management")
    )
).not().isVisible();
```

For menu-based restrictions, the test should verify that restricted actions are hidden or disabled according to the product rule:

```
viewerPage.navigate("https://example.com/dashboard");

PlaywrightAssertions.assertThat(
    viewerPage.getByRole(
        AriaRole.LINK,
        new Page.GetByRoleOptions().setName("Admin Console")
    )
).isHidden();
```

For critical permissions, UI validation alone is not enough. Hiding a button or menu item does not prove security. Backend authorization should also reject restricted actions if the user bypasses the UI and calls the endpoint directly.

Each role should use an isolated [BrowserContext](#) so cookies, tokens, permissions, and storage state do not leak between admin, manager, viewer, or normal-user tests.

## Common Mistake

Testing only whether the admin menu is hidden, without checking direct restricted URL access, forbidden-page behavior, absence of protected content, and backend rejection of unauthorized actions.

---

## Question 1.124

### How do you test UI updates that happen without full page reload?

#### Interview-Style Answer

I would trigger the UI action and then assert the specific updated state, such as changed text, added row, updated count, visible toast, changed button state, or refreshed component data.

Modern applications often update the UI through client-side state changes or API responses without full page reload. So I would not wait for navigation unless the product actually changes route.

If the update depends on an API call, I may wait for the relevant response, but I would still validate the final rendered UI result because the user only cares whether the updated state is visible and correct.

#### Detailed Explanation

Many React, Angular, and Vue applications update parts of the page without reloading the browser document. The URL may remain the same, but the UI changes after a state update, API response, WebSocket message, or background refresh.

Examples include:

1. Table refresh
2. Status update
3. Inline edit
4. Search filter
5. Notification update
6. Cart count update
7. Dashboard widget refresh
8. Save operation showing updated field value

A good test waits for the exact updated UI state:

```
page.getByRole(
  AriaRole.BUTTON,
  new Page.GetByRoleOptions().setName("Refresh Orders")
).click();

PlaywrightAssertions.assertThat(
  page.getByText("Updated just now")
).isVisible();

Locator row = page.getByRole(AriaRole.ROW)
  .filter(new Locator.FilterOptions().setHasText("ORD-1001"));

PlaywrightAssertions.assertThat(row).isVisible();
```



---

For an inline edit, validate the value after save:

```
Locator row = page.getByRole(AriaRole.ROW)
    .filter(new Locator.FilterOptions().setHasText("ORD-1001"));

row.getByRole(
    AriaRole.BUTTON,
    new Locator.GetByRoleOptions().setName("Edit")
).click();

page.getByLabel("Status").selectOption("Approved");

page.getByRole(
    AriaRole.BUTTON,
    new Page.GetByRoleOptions().setName("Save")
).click();

PlaywrightAssertions.assertThat(row).containsText("Approved");
```

If the update is API-driven, combine network wait with UI validation:

```
Response response = page.waitForResponse(
    res -> res.url().contains("/api/orders")
    && res.status() == 200,
    () -> page.getByRole(
        AriaRole.BUTTON,
        new Page.GetByRoleOptions().setName("Refresh Orders")
    ).click()
);

PlaywrightAssertions.assertThat(
    page.getByText("Updated just now")
).isVisible();
```

The response confirms that the backend returned, but the locator assertion confirms that the browser rendered the updated state.

## Common Mistake

Waiting for navigation or page reload when the application updates content in place, instead of asserting the exact changed UI state such as updated text, row, count, status, toast, or refreshed component data.

---

## Question 1.132

**How would you test frontend behavior when an API returns an empty response?**

### Interview-Style Answer

To test frontend behavior when an API returns an empty response, I would mock the API with a valid empty response and verify that the UI shows the correct empty-state behavior. The page should not crash, show stale data, display misleading content, or leave the user with a blank unexplained area.

In Playwright Java, this is useful because it separates frontend empty-state handling from backend data availability. The test controls the API response and then validates the browser-visible result, such as "No orders found," "No notifications," "No search results," or a disabled action button.

### Detailed Explanation

Empty responses are common in real applications. A new user may have no orders, a search may return no results, a notification list may be empty, or a customer may have no saved addresses. The frontend should handle these cases safely and clearly.

Example:

```
page.route("**/api/orders", route -> {
    route.fulfill(new Route.FulfillOptions()
        .setStatus(200)
        .setContentType("application/json")
        .setBody("{\"orders\":[]}"));
});

page.navigate("https://example.com/orders");

PlaywrightAssertions.assertThat(
    page.getByText("No orders found")
).isVisible();
```

This test validates that the backend returned a legitimate empty result and that the frontend displayed the correct empty-state message.

A stronger empty-state test may also verify that old or misleading data is not shown:

---

```
PlaywrightAssertions.assertThat(
    page.getByTestId("orders-table")
).isHidden();

PlaywrightAssertions.assertThat(
    page.getByText("No orders found")
).isVisible();
```

For search results:

```
page.route("**/api/search?query=unknown*", route -> {
    route.fulfill(new Route.FulfillOptions()
        .setStatus(200)
        .setContentType("application/json")
        .setBody("{\"results\":[]}"));
});

page.getByLabel("Search").fill("unknown item");

page.getByRole(
    AriaRole.BUTTON,
    new Page.GetByRoleOptions().setName("Search")
).click();

PlaywrightAssertions.assertThat(
    page.getByText("No results found")
).isVisible();
```

The important point is to connect the mocked API response to a user-visible outcome. The test should prove that the frontend handles the empty response gracefully, with a clear message, correct layout, and no stale records.

## Common Mistake

testing only data-present scenarios. A reliable frontend test suite should also validate empty API responses so users get meaningful empty-state behavior instead of broken layouts, stale data, or confusing blank screens.

---

### Question 1.138

**A critical end-to-end test is slow because it depends on real backend APIs. Would you mock the APIs?**

### Interview-Style Answer

I would not automatically mock the APIs. If the purpose of the test is release-critical end-to-end confidence, I would keep the real backend path because the integration itself is part of what the test is validating.

Mocks are useful for speed, control, and edge cases, but they reduce integration confidence. For a critical flow such as payment, order creation, approval, authentication, or checkout, mocking the backend may hide the exact risk the end-to-end test is supposed to catch. I may create additional mocked tests for rare errors, empty states, unauthorized responses, or slow third-party behavior, but I would preserve at least one real-backend E2E path for release validation.

### Detailed Explanation

A slow end-to-end test should be reviewed carefully before deciding to mock APIs. The right decision depends on the purpose of the test. If the test is meant to validate frontend behavior only, mocks may be appropriate. But if the test is meant to certify that the frontend, backend, authentication, database, business rules, and integrations work together, mocking the backend removes important coverage.

Keep the real backend for flows such as:

- Payment
- Order creation
- Approval workflow
- Authentication
- Checkout
- Critical business integration
- Release smoke validation

Use mocks for scenarios such as:

- Rare server errors
- Empty responses
- Unauthorized or forbidden responses
- Partial or malformed API data
- Slow third-party service behavior
- UI fallback and retry behavior

For example, if the release-critical test validates order creation, the real backend should usually remain part of that test:

```
page.getByRole(
  AriaRole.BUTTON,
  new Page.GetByRoleOptions().setName("Submit Order")
).click();

PlaywrightAssertions.assertThat(
  page.getByText("Order created successfully")
).isVisible();

Locator orderRow = page.getByTestId("orders-table")
  .getByRole(AriaRole.ROW)
  .filter(new Locator.FilterOptions().setHasText("ORD-1001"));

PlaywrightAssertions.assertThat(orderRow).isVisible();
```

If the same flow is slow, I would first optimize the test without removing critical integration coverage. For example:

1. Use API setup for preconditions.
2. Remove unnecessary UI steps.
3. Use stable and unique test data.
4. Avoid repeated login by using storage state.
5. Run the critical E2E test only in the release suite.
6. Improve backend test environment reliability.
7. Split edge-case coverage into faster mocked tests.

A balanced framework can have both real-backend and mocked coverage. The real-backend test proves integration confidence. Mocked tests prove frontend behavior for controlled edge cases. This keeps the suite both trustworthy and efficient.

## Common Mistake

mocking the exact backend behavior that the critical end-to-end test is supposed to validate. Mocks are useful, but they should not remove release-critical integration confidence.

## Question 1.148

### How would you test an empty-state UI using route mocking?

#### Interview-Style Answer

I would test an empty-state UI by mocking the relevant API response to return an empty array or empty result set, then asserting that the correct empty-state message appears.

I would also verify that stale data is not shown, table rows or cards are absent, pagination is hidden or disabled if expected, and any allowed recovery action such as Create New or Add Order is visible. This makes the empty-state test deterministic instead of depending on the real backend having no data.

#### Detailed Explanation

Empty-state UI should be tested with controlled data because relying on the backend to be empty by chance can make the test flaky. In Playwright Java, `page.route()` can be used to mock the API response before navigating to the page.

Example:

```
page.route("**/api/orders", route -> {
    route.fulfill(new Route.FulfillOptions()
        .setStatus(200)
        .setContentType("application/json")
        .setBody("[]"));
});

page.navigate(baseUrl + "/orders");

PlaywrightAssertions.assertThat(
    page.getByText("No orders found")
).isVisible();

PlaywrightAssertions.assertThat(
    page.getTestId("orders-table").locator("tbody tr")
).hasCount(0);
```

If the real API contract wraps data inside an object, the mock should follow the same contract:

```
page.route("**/api/orders", route -> {
  route.fulfill(new Route.FulfillOptions()
    .setStatus(200)
    .setContentType("application/json")
    .setBody("""
      {
        "orders": [],
        "total": 0
      }
      """));
});
```

The test should then validate the complete empty-state behavior:

```
PlaywrightAssertions.assertThat(
  page.getByText("No orders found")
).isVisible();

PlaywrightAssertions.assertThat(
  page.getByTestId("orders-table-row")
).hasCount(0);

PlaywrightAssertions.assertThat(
  page.getByRole(
    AriaRole.BUTTON,
    new Page.GetByRoleOptions().setName("Create Order")
  )
).isVisible();
```

Good empty-state checks include:

1. Empty-state message appears.
2. Table rows, cards, or list items are absent.
3. Pagination is hidden or disabled.
4. Create/new action appears if expected.
5. No stale or cached data remains visible.
6. The mocked response matches the real API contract.

If the empty state appears only after filtering, the test should mock or trigger the filtered API response and assert that the empty message is related to the filter result, not a generic page failure.

## Common Mistake

Depending on real backend data to be empty by chance, or mocking an unrealistic response shape that does not match the real API contract, causing the test to pass without proving the actual empty-state behavior.

---

## **Part II - Additional Questions**



---

## Question 1.200

**How do you test responsive UI behavior across desktop, tablet, and mobile profiles in Playwright Java?**

### Interview-Style Answer

I would test responsive UI behavior by defining a small, risk-based set of browser context profiles and running the same critical workflows across those profiles.

In Playwright Java, teams usually implement this through JUnit parameterized tests, TestNG data providers, Maven profiles, Gradle properties, or CI/CD matrix jobs. The test logic should remain reusable, while the `BrowserContext` configuration changes for desktop, tablet, and mobile-like profiles.

This helps validate whether important user journeys work correctly across screen sizes without creating duplicate test classes for each device type.

### Detailed Explanation

Responsive testing should focus on business-critical workflows and layout-sensitive areas, not every test in the suite. For example, login, checkout, search, navigation menus, forms, dashboards, and payment flows may need responsive coverage, while backend-heavy admin flows may not need to run on every profile.

In Playwright Java, the equivalent of device/profile-based execution is usually handled through framework configuration. The test receives a profile name, creates the matching `BrowserContext`, and then runs the same workflow.

Example:

```
static Browser.NewContextOptions profile(String name) {
    return switch (name.toLowerCase()) {
        case "mobile" -> new Browser.NewContextOptions()
            .setViewportSize(390, 844)
            .setIsMobile(true)
            .setHasTouch(true)
            .setDeviceScaleFactor(3);

        case "tablet" -> new Browser.NewContextOptions()
            .setViewportSize(768, 1024)
            .setIsMobile(true)
            .setHasTouch(true)
            .setDeviceScaleFactor(2);

        default -> new Browser.NewContextOptions()
            .setViewportSize(1366, 768);
    };
}
```

The test can then create a context based on the selected profile:

```
BrowserContext context = browser.newContext(profile("mobile"));
Page page = context.newPage();

page.navigate(baseUrl);

PlaywrightAssertions.assertThat(
    page.getByRole(
        AriaRole.BUTTON,
        new Page.GetByRoleOptions().setName("Menu")
    )
).isVisible();
```

A good responsive test should validate both functionality and layout-relevant behavior. For example, on desktop the navigation may be visible as a full menu, while on mobile it may be hidden behind a hamburger button.

Useful responsive validations include:

1. Desktop navigation appears correctly.
2. Mobile hamburger menu opens and closes.
3. Tablet layout does not overlap or clip content.
4. Important buttons remain visible and tappable.
5. Forms are usable on smaller screens.
6. Sticky headers, footers, and modals do not block actions.
7. Critical data remains readable without broken layout.

The same workflow should be reused wherever possible. Only the profile setup and profile-specific assertions should change.

Example idea:

---

```
void verifySearchWorks(Page page) {
    page.getByRole(
        AriaRole.SEARCHBOX,
        new Page.GetByRoleOptions().setName("Search")
    ).fill("laptop");

    page.getByRole(
        AriaRole.BUTTON,
        new Page.GetByRoleOptions().setName("Search")
    ).click();

    PlaywrightAssertions.assertThat(
        page.getByText("Search results")
    ).isVisible();
}
```

Then this reusable workflow can run against desktop, tablet, and mobile contexts.

It is also important to understand the limits of emulation. Playwright can simulate viewport size, touch support, user agent, and device scale factor, but it is not a complete replacement for real-device testing. Real devices may still reveal issues with mobile browser behavior, virtual keyboard, gestures, performance, rendering, and touch interaction.

## Common Mistake

Creating separate duplicate test classes for desktop, tablet, and mobile. A better approach is to parameterize the browser context profile and reuse the same workflow, adding only profile-specific assertions where the UI behavior genuinely differs.

---

## Question 1.211

### How would you test a tooltip using Playwright Java?

#### Interview-Style Answer

I would trigger the tooltip the same way a user does, usually by hovering over or focusing the target element, then assert that the expected tooltip text becomes visible.

For icon-only tooltip triggers, I would use a stable accessible name or `data-testid`, and I would verify that the tooltip belongs to the correct field or control. If the tooltip is accessible, I would prefer `getByRole(AriaRole.TOOLTIP)` for validation.

I would also test that the tooltip disappears when the pointer moves away or focus is removed, if that is the expected behavior.

#### Detailed Explanation

Tooltips are often used for help text, validation hints, field descriptions, disabled-action explanations, or icon-only controls. A reliable test should not only check that some tooltip appears. It should prove that the correct tooltip appears for the correct trigger.

Example:

```
page.getByLabel("Info").hover();

PlaywrightAssertions.assertThat(
    page.getByText("Password must contain at least 8 characters")
).isVisible();
```

For icon-only tooltip triggers, a `data-testid` or accessible label is usually better than a fragile CSS selector:

```
page.getByTestId("password-info-icon").hover();

PlaywrightAssertions.assertThat(
    page.getByRole(AriaRole.TOOLTIP)
).containsText("Password must contain at least 8 characters");
```

Good tooltip checks include:

1. Tooltip appears on hover or focus.
2. Tooltip text is correct.
3. Tooltip is associated with the correct control.
4. Tooltip disappears when hover moves away if expected.
5. Tooltip does not appear for the wrong field or icon.
6. Keyboard/focus behavior works if accessibility is required.

---

If the tooltip has a small animation delay, avoid `Thread.sleep()`. Use a Playwright assertion on the tooltip text or role so the test waits until the tooltip is visible.

To validate disappearance:

```
page.getByRole(
    AriaRole.BUTTON,
    new Page.GetByRoleOptions().setName("Submit")
).hover();

PlaywrightAssertions.assertThat(
    page.getByRole(AriaRole.TOOLTIP)
).isVisible();

page.mouse().move(0, 0);

PlaywrightAssertions.assertThat(
    page.getByRole(AriaRole.TOOLTIP)
).isHidden();
```

## Common Mistake

checking only that a tooltip appears somewhere on the page, without verifying the correct tooltip text, correct trigger element, hover/focus behavior, or disappearance rule.

---

## Question 1.232

**How do you validate that duplicate API calls are not triggered by a single UI action?**

### Interview-Style Answer

I validate duplicate API calls by intercepting and counting network requests triggered by a single user action and asserting that only the expected number of calls (usually one) is made.

This ensures the frontend does not accidentally trigger multiple backend operations due to issues like double-clicks, re-renders, or duplicate event bindings.

### Detailed Explanation

Duplicate API calls can lead to serious production issues such as duplicate orders, duplicate payments, or inconsistent application state. These often occur due to:

- Double-click or rapid user interactions
- Missing button disable after first click
- React/Vue re-render triggering multiple handlers
- Incorrect retry or debounce logic
- Multiple event listeners attached accidentally

In Playwright Java, we validate this using request interception and counters.

Example: counting API calls

```
AtomicInteger createCalls = new AtomicInteger();

page.route("**/api/orders", route -> {
    if ("POST".equals(route.request().method())) {
        createCalls.incrementAndGet();
    }

    route.fulfill(new Route.FulfillOptions()
        .setStatus(201)
        .setContentType("application/json")
        .setBody("{\"id\":\"ORD-1001\"}"));
});
```

Trigger UI action

---

```
page.navigate(appUrl + "/orders/new");

page.getByLabel("Order name").fill("No duplicate order");

Locator createButton = page.getByRole(
    AriaRole.BUTTON,
    new Page.GetByRoleOptions().setName("Create order")
);

// Simulate aggressive user interaction
createButton.dblclick();
```

## Validate single API call

```
PlaywrightAssertions.assertThat(
    page.getByText("Order created")
).isVisible();

Assertions.assertEquals(1, createCalls.get());
```

## Strong validation strategy (real projects)

- Assert request count (no duplicates)
- Validate backend state (no duplicate records)
- Ensure UI disables button after first click
- Verify idempotency at API level (where applicable)

## Key principle

UI action → should map to exactly one business operation unless explicitly designed otherwise.

## Common Mistake

Only clicking once and assuming the system is correct, without validating actual network traffic or backend side effects, which allows hidden duplicate API calls to go unnoticed in production.

---

---

## 2. Debugging



---

## **Part I - Core Questions**

## Question 2.4

### How does `page.pause()` help during local Playwright Java debugging?

#### Interview-Style Answer

`page.pause()` pauses test execution at a specific point and opens the Playwright Inspector during local debugging. It lets me inspect the current page state, try locators, verify whether the expected elements are present, and continue the test step by step.

It is useful when I need interactive investigation, especially for locator issues, unexpected page states, overlays, dialogs, navigation problems, or actionability failures. However, it should only be used locally and should not be committed into normal test code because it blocks unattended CI/CD execution.

#### Detailed Explanation

`page.pause()` is a local debugging tool in Playwright. When execution reaches this line, Playwright stops the test and opens the Inspector. From there, I can look at the browser state, inspect elements, test selectors, and continue execution manually.

Example:

```
Browser browser = playwright.chromium().launch(
    new BrowserType.LaunchOptions().setHeadless(false)
);

Page page = browser.newPage();
page.navigate("https://example.com/login");

page.pause();

page.getByLabel("Username").fill("admin");
page.getByLabel("Password").fill("secret");
page.getByRole(
    AriaRole.BUTTON,
    new Page.GetByRoleOptions().setName("Login")
).click();
```

This is useful when I want to stop just before the failing step and check what the application really looks like. For example, I can verify whether the login page loaded correctly, whether the username field has the expected accessible label, whether a modal is covering the button, or whether the test is on a different page than expected.

---

It also helps when debugging locators. I can pause the test, inspect the DOM and accessibility information, and confirm whether a locator such as `getByRole()`, `getByLabel()`, `getByText()`, or `locator()` is matching the intended element.

For example, if this click fails:

```
page.getByRole(  
  AriaRole.BUTTON,  
  new Page.GetByRoleOptions().setName("Submit")  
) .click();
```

I can insert `page.pause()` before it and check whether the button is visible, enabled, covered by another element, renamed, inside an `iframe`, or not yet rendered.

`page.pause()` is best for interactive local debugging. For CI failures, Playwright traces, screenshots, videos, console logs, and network details are usually better because they can be collected automatically and reviewed after the run.

## Common Mistake

leaving `page.pause()` in committed test code. Since it waits for human interaction, it can block Maven, Gradle, JUnit, TestNG, or CI/CD execution and make the pipeline hang.

---

## Question 2.17

**An element is visible on screen, but `getByRole()` cannot find it. How would you debug this?**

### Interview-Style Answer

I would debug it by checking what Playwright can identify through the accessibility model, not just what is visually visible on the screen. `getByRole()` depends on the element's semantic role and accessible name, so I would verify whether the element is really exposed as the expected role, whether its accessible name matches the locator, and whether it is hidden from the accessibility tree.

I would also check whether the element is inside an iframe, duplicated elsewhere on the page, rendered after an API call, or visually styled to look like a button or link without proper HTML semantics. If the role locator fails, the fix may be improving the application's accessibility or locator scope, not immediately replacing it with XPath.

### Detailed Explanation

A visually visible element is not always findable using `getByRole()`. Role locators work based on accessibility semantics. For example, Playwright can easily find a real `<button>` with the accessible name Submit, but it may not find a `<div>` styled like a button unless the application exposes it correctly with a role and accessible name.

Example:

```
page.getByRole(
  AriaRole.BUTTON,
  new Page.GetByRoleOptions().setName("Submit")
).click();
```

If this fails even though the Submit button is visible, I would first inspect the element using Playwright Inspector or Trace Viewer and check:

1. Is the element really exposed as a button, link, checkbox, textbox, or heading?
2. Does the accessible name exactly match "Submit"?
3. Is the visible text different from the accessible name?
4. Is the element hidden using `aria-hidden`, `hidden`, `display:none`, or `visibility:hidden`?
5. Is the element inside an iframe?
6. Are there multiple matching elements causing strict mode issues?
7. Is the element rendered only after an API response or client-side update?
8. Is another overlay or loading layer covering the real interactive element?

---

A common issue is that the UI looks like a button but is implemented like this:

```
<div class="primary-button">Submit</div>
```

This may look clickable to the user, but it does not have proper button semantics. A better implementation would be:

```
<button>Submit</button>
```

or, if custom markup is unavoidable:

```
<div role="button" aria-label="Submit">Submit</div>
```

If the visible text and accessible name are different, the locator should use the accessible name. For example, an icon button may visually show only a search icon, but the accessible name may be Search products:

```
page.getByRole(
  AriaRole.BUTTON,
  new Page.GetByRoleOptions().setName("Search products")
).click();
```

If the element is inside an iframe, `page.getByRole()` will not find it from the main page. I should first target the iframe with `frameLocator()`, then locate the element inside the iframe:

```
FrameLocator frame = page.frameLocator("iframe[title='Payment']");

frame.getByRole(
  AriaRole.BUTTON,
  new Locator.GetByRoleOptions().setName("Pay")
).click();
```

Here, `frameLocator()` targets the iframe, and the chained `getByRole()` locates the actual button inside that iframe.

If multiple elements match the same role and name, Playwright strict mode may fail. In that case, I should scope the locator to the correct section, dialog, form, card, or table row instead of using a broad page-level locator:

```
Locator loginForm = page.getByRole(
  AriaRole.FORM,
  new Page.GetByRoleOptions().setName("Login")
);

loginForm.getByRole(
  AriaRole.BUTTON,
  new Locator.GetByRoleOptions().setName("Submit")
).click();
```

---

If the element appears after an API call, I would use a web-first assertion before interacting:

```
Locator submitButton = page.getByRole(  
    AriaRole.BUTTON,  
    new Page.GetByRoleOptions().setName("Submit")  
);  
  
PlaywrightAssertions.assertThat(submitButton).isVisible();  
submitButton.click();
```

If the role locator failure reveals that the application has poor accessibility, I would raise that as a product issue. Role-based locators are valuable because they encourage tests to interact with the application like a real user and make the test more maintainable.

## Common Mistake

replacing a failed `getByRole()` locator with a long XPath without checking the actual role, accessible name, iframe context, visibility, and strict mode behavior. A failed role locator often points to an accessibility, scoping, or semantic HTML issue that should be understood first.

---

## Question 2.22

### How would you debug locator failures inside web components or Shadow DOM?

#### Interview-Style Answer

I would first check whether the web component uses open or closed Shadow DOM, because Playwright can work with open Shadow DOM but cannot pierce closed Shadow DOM internals directly.

For debugging, I would prefer user-facing locators such as `getByRole()`, `getByLabel()`, `getByText()`, or `getByTestId()` where possible. If the component exposes proper roles, labels, and accessible names, the test can validate behavior without depending on fragile internal markup.

#### Detailed Explanation

Web components can hide their internal DOM structure behind Shadow DOM. If the Shadow DOM is open, Playwright can usually locate elements using normal locators when the elements are accessible through the composed tree. If the Shadow DOM is closed, the internal elements are intentionally hidden from automation and page scripts, so the test should interact through the public behavior of the component instead of trying to access private internals.

A good debugging checklist is:

1. Check whether the component uses open or closed Shadow DOM.
2. Inspect whether the target element has a proper role and accessible name.
3. Try role, label, text, or test-id locators first.
4. Check whether the locator matches zero, one, or multiple elements.
5. Scope the locator to the component or surrounding section when duplicates exist.
6. Avoid long CSS paths into component internals.
7. Validate the visible behavior of the component, not private markup.

Example of a user-facing locator:

```
page.getByRole(
  AriaRole.BUTTON,
  new Page.GetByRoleOptions().setName("Choose date")
).click();

PlaywrightAssertions.assertThat(
  page.getByText("May 2026")
).isVisible();
```

This is better than relying on internal implementation details such as:

---

```
page.locator("date-picker >>> div.calendar > button.next").click();
```

A better test focuses on the component's public behavior. For example, if the date picker opens a calendar and allows selecting a date, the test should assert that the calendar appears, the expected month is shown, and the selected date is reflected in the input or page state.

If the locator fails, I would inspect the component in Playwright Inspector or Trace Viewer and check whether the visible text is actually exposed as an accessible name. If not, the issue may be an accessibility problem in the component, not only a test issue.

For custom controls, adding stable accessible names or test IDs at the component boundary is often better than exposing internal structure. This keeps tests stable even if the component implementation changes.

## Common Mistake

using long internal CSS paths inside web components. Such locators break easily when the component markup changes; role, label, text, or test-id locators aimed at public component behavior are usually more reliable.

---



## Question 2.30

**A test fails because the DOM element is detached during interaction. How would you solve it?**

### Interview-Style Answer

I would treat this as a re-rendering or stale DOM reference problem. In Playwright Java, the best approach is to use `Locator` instead of storing an element too early, because a `Locator` re-resolves the matching element when the action or assertion is performed.

I would also check why the DOM is changing during the interaction, such as React/Angular/Vue re-rendering, grid refresh, filtering, sorting, lazy loading, animation, or auto-refresh. Then I would wait for the UI to reach a stable application state before interacting with the element.

### Detailed Explanation

A detached element failure usually means the test found an element, but before Playwright completed the action, that DOM node was removed and replaced by the application. To the user, the button or row may look the same, but internally it is a new DOM element.

This commonly happens in modern applications with:

- React, Angular, or Vue re-rendering
- Virtualized tables or grids
- Sorting or filtering
- Auto-refreshing lists
- Skeleton loaders replaced by real content
- Dropdowns or modals being recreated
- Conditional rendering after API responses
- Animations or layout transitions

The wrong approach is to capture an element-like reference too early and reuse it after the page updates. A safer approach is to keep the interaction locator-based:

```
Locator row = page.getByTestId("orders-grid")
    .getByRole(AriaRole.ROW)
    .filter(new Locator.FilterOptions().setHasText("ORD-1001"));

PlaywrightAssertions.assertThat(row).isVisible();

row.getByRole(
    AriaRole.BUTTON,
    new Locator.GetByRoleOptions().setName("Approve")
).click();

PlaywrightAssertions.assertThat(
    page.getByTestId("order-status")
).hasText("Approved");
```

Here, the row and button are located through [Locator](#) chains. Playwright resolves them when the assertion or click runs, instead of depending on an old DOM node.

I would also wait for the real readiness condition before clicking:

```
PlaywrightAssertions.assertThat(
    page.getByTestId("orders-loading")
).not().isVisible();

PlaywrightAssertions.assertThat(row).isVisible();
```

If the detachment happens because the row refreshes after an API call, I would wait for the relevant response and then assert the final UI state:

```
Response response = page.waitForResponse(
    r -> r.url().contains("/api/orders")
    && r.status() == 200,
    () -> page.getByRole(
        AriaRole.BUTTON,
        new Page.GetByRoleOptions().setName("Refresh")
    ).click()
);

PlaywrightAssertions.assertThat(row).isVisible();
```

This fixes the real issue: the test interacts only after the currently rendered UI is ready.

## Common Mistake

Storing an element reference too early or reusing it after the application re-renders, instead of using [Locator](#) chains and waiting for the stable UI state before interaction.

## Question 2.45

**A mocked response from one test affects another test. What is the likely design issue?**

### Interview-Style Answer

The likely design issue is route leakage caused by a mock being registered too broadly, globally, or on a shared `BrowserContext`. Network mocks should normally be scoped to the test that needs them. If one test's mocked response affects another test, the framework may be reusing browser state, keeping routes active across tests, or using a URL pattern that intercepts unrelated requests.

I would fix this by creating a fresh `BrowserContext` per test, registering mocks only inside the relevant test or fixture scope, using narrow route patterns, and closing the context after the test. In parallel execution, mocks, users, data, files, and storage state should be isolated.

### Detailed Explanation

A mocked response should not affect another independent test unless the test framework design allows state to leak. In Playwright Java, routes are attached either to a `Page` or a `BrowserContext`. If the same context is reused across multiple tests, a route registered in one test can continue affecting later tests.

For example, this kind of broad mock is risky:

```
browserContext.route("**/api/**", route -> {
    route.fulfill(new Route.FulfillOptions()
        .setStatus(200)
        .setContentType("application/json")
        .setBody("{\"mocked\":true}"));
});
```

This can accidentally intercept many API calls, not just the endpoint needed for one test. If the same `BrowserContext` is reused, other tests may receive mocked data when they expect real backend data.

Common causes include:

1. Shared `BrowserContext` across tests
2. Global route registration in setup
3. Broad route pattern such as `**/api/**`
4. Mock placed in a base class used by all tests
5. Context not closed after the test
6. Route not removed when the test finishes
7. Parallel tests sharing the same context, user, or storage state
8. Mock intercepting more endpoints than intended

---

A better design is to scope the mock narrowly inside the test that needs it:

```
BrowserContext context = browser.newContext();
Page page = context.newPage();

page.route("**/api/orders/summary", route -> {
    route.fulfill(new Route.FulfillOptions()
        .setStatus(200)
        .setContentType("application/json")
        .setBody("{\"totalOrders\":5}"));
});

page.navigate("https://example.com/dashboard");

PlaywrightAssertions.assertThat(
    page.getByText("Total Orders: 5")
).isVisible();

context.close();
```

If the mock must apply to all pages inside the same test, `browserContext.route()` is fine, but the context should still be test-scoped:

```
BrowserContext context = browser.newContext();

context.route("**/api/orders/summary", route -> {
    route.fulfill(new Route.FulfillOptions()
        .setStatus(200)
        .setContentType("application/json")
        .setBody("{\"totalOrders\":5}"));
});
```

The route pattern should be as specific as possible. Mocking `**/api/orders/summary` is safer than mocking `**/api/**` because it limits the mock to the endpoint required by the scenario.

For parallel execution, I would also make sure each test uses isolated users, isolated test data, isolated downloads/uploads, safe storage-state files, and cleanup. A mock leak is often a symptom of a broader isolation problem in the automation framework.

## Common Mistake

putting all network mocks into global setup for convenience. This makes tests look shorter, but it can cause hidden dependencies, route leakage, false positives, and failures when tests run in a different order or in parallel.

---

## **Book 3 - Framework & Test Design**

---

## **1. Test Design**

---

## **Part I - Core Questions**

## Question 1.10

### How do you review a Playwright test for maintainability?

#### Interview-Style Answer

I review whether the test has a clear business purpose, readable structure, stable locators, meaningful assertions, isolated data, and reliable synchronization. A maintainable Playwright test should be easy to understand, safe to run in CI, and simple to debug when it fails.

I also check whether the test follows team standards: proper Page Object or component usage, no hard waits, no shared mutable data, no hidden assertions, and no unnecessary implementation details in the test body.

#### Detailed Explanation

A maintainable Playwright Java test should tell a clear business story. A reviewer should be able to understand what the test proves without reading every internal locator or knowing the page's DOM structure.

Example:

```
@Test
void shouldApprovePendingOrder() {
    // Arrange
    String orderId = testDataApi.createPendingOrder();

    // Act
    ordersPage.open(orderId);
    ordersPage.approveOrder();

    // Assert
    ordersPage.shouldShowStatus("Approved");
}
```

I would review the test against these points:

- Does the test validate one clear business outcome?
- Is the Arrange-Act-Assert structure visible?
- Are locators based on user intent, such as role, label, text, or stable test IDs?
- Are assertions meaningful and tied to business state?
- Does the test avoid Thread.sleep() and hard waits?
- Does each test own its data?
- Is BrowserContext/Page isolation handled correctly?
- Are repeated actions moved to readable Page Object or component methods?
- Are important assertions visible or clearly named?
- Is cleanup targeted and safe for parallel execution?

I would also check locator quality. This is harder to maintain:



---

```
page.locator(".btn-primary:nth-child(2)").click();
```

This is usually clearer:

```
page.getByRole(  
  AriaRole.BUTTON,  
  new Page.GetByRoleOptions().setName("Approve")  
) .click();
```

For synchronization, I would reject hard waits and prefer web-first assertions or meaningful event waits:

```
PlaywrightAssertions.assertThat(  
  page.getByTestId("order-status")  
) .hasText("Approved");
```

From a framework point of view, maintainability also includes ownership and review rules. Teams should agree on locator strategy, Page Object boundaries, data setup patterns, artifact capture, naming conventions, and CI execution standards. This keeps the suite scalable as more tests and contributors are added.

## Common Mistake

Reviewing only whether the test passes, without checking whether it has a clear business purpose, stable locator strategy, isolated data, meaningful assertions, reliable synchronization, and long-term maintainability in CI.

---

### Question 1.13

## How would you avoid duplicate or low-value tests in a Playwright Java suite?

### Interview-Style Answer

I would avoid duplicate or low-value tests by reviewing whether each test validates a unique business risk, has meaningful assertions, and belongs at the right test level. Not every UI scenario needs to be an end-to-end Playwright test.

In a healthy Playwright Java suite, test value should be measured by risk coverage, defect detection, maintainability, and CI execution cost, not by the total number of tests.

### Detailed Explanation

Duplicate tests increase execution time, maintenance effort, flakiness, and CI cost without improving confidence. Two tests may not look identical in code, but they can still validate the same behavior through slightly different data or navigation paths.

A useful review checklist is:

1. Does this test validate a unique business behavior?
2. Is the same risk already covered by another test?
3. Does the test have a clear pass/fail business assertion?
4. Is this scenario better covered by an API, unit, or component test?
5. Does the test repeat a long setup flow only to check a small variation?
6. Will this test catch a real regression?
7. Is the ownership of this test clear?
8. Is the test stable enough for regular CI execution?

A low-value test usually performs UI actions without validating a meaningful outcome:

Click a button and assert that another button is visible, without checking whether the expected business state changed.

A better Playwright test should connect the action to a user-visible result:

---

```
page.getByRole(  
  AriaRole.BUTTON,  
  new Page.GetByRoleOptions().setName("Submit Order")  
)<div data-bbox="102 246 901 367" data-label="Text">

To control duplication, teams should define review rules for new tests. For example, every new Playwright test should explain the risk it covers, the expected business outcome, and why it belongs in the UI suite instead of a lower-level test. Existing tests should be periodically reviewed for overlap, weak assertions, unstable flows, and scenarios that can be moved to API or component coverage.


```

Good suite governance includes grouping tests by feature area, assigning ownership, tagging smoke/regression tests, tracking flaky tests, and removing tests that no longer provide unique value.

## Common Mistake

judging automation maturity by the number of Playwright tests, while ignoring duplicate coverage, weak assertions, long CI execution time, and tests that do not protect any meaningful business behavior.

---

## Question 1.23

**A test uses `.first()` to fix a strict mode violation. How would you review that change?**

### Interview-Style Answer

I would not accept `.first()` as a fix unless the first matching element is truly the business requirement. A strict mode violation usually means the locator matches more than one element, so using `.first()` may hide the real ambiguity instead of solving it.

I would ask the author to make the locator more specific by using user intent, business identity, and proper scoping. For example, if the test wants to edit a specific invoice, the locator should first identify that invoice row and then click the Edit button inside that row.

### Detailed Explanation

Playwright strict mode is useful because it warns us when an action locator is ambiguous. If a locator matches multiple buttons, links, rows, or fields, Playwright does not know which one represents the intended user action.

A weak fix is:

```
page.getByRole(
  AriaRole.BUTTON,
  new Page.GetByRoleOptions().setName("Edit")
).first().click();
```

This may make the test pass, but it does not prove that the correct Edit button was clicked. If the page has multiple rows, cards, dialogs, or repeated components, `.first()` may click the wrong item when sorting, filtering, or layout changes.

A better fix is to scope the action to a meaningful parent element:

```
Locator row = page.getByRole(AriaRole.ROW)
    .filter(new Locator.FilterOptions().setHasText("INV-1001"));

row.getByRole(
  AriaRole.BUTTON,
  new Locator.GetByRoleOptions().setName("Edit")
).click();
```

This version clearly says: edit invoice `INV-1001`. It uses business identity first, then finds the action button within that row. That makes the locator more reliable, readable, and aligned with the user scenario.

---

I would allow `.first()` only when the requirement is genuinely about the first item. For example, if the scenario says “open the first search result,” then `.first()` can be valid:

```
page.getByRole(AriaRole.LINK)
  .filter(new Locator.FilterOptions().setHasText("Search result"))
  .first()
  .click();
```

Even then, I would prefer a clearer locator if the first result has a known title, label, or testable business meaning.

When reviewing this change, I would ask:

1. Why does the locator match multiple elements?
2. Which exact element should the test interact with?
3. Can we scope it to a row, card, dialog, section, or form?
4. Can we use a role locator with accessible name?
5. Can we use business identity such as invoice ID, email, order ID, or title?
6. Is the first element truly required by the scenario?

## Common Mistake

using `.first()` or `nth()` as a shortcut to silence strict mode. A better fix is to remove locator ambiguity by using meaningful scoping, stable user-facing locators, and business identity.

---

## Question 1.24

**A test uses a broad global mock for `**/api/**`. What review concerns would you raise?**

### Interview-Style Answer

I would raise a serious review concern because `**/api/**` is too broad for most Playwright UI tests. It can intercept login, permissions, configuration, feature flags, dashboard data, orders, users, and many unrelated API calls. That can make the test pass in an unrealistic environment and hide real integration defects.

I would ask the author to narrow the route to the exact endpoint needed for the scenario, keep the mock scoped to the specific test, and verify the final user-visible UI behavior. A mock should control only the dependency required for the test, not replace the application's entire backend behavior.

### Detailed Explanation

A broad global mock such as this is risky:

```
page.route("**/api/**", route => {
  route.fulfill(new Route.FulfillOptions()
    .setStatus(200)
    .setContentType("application/json")
    .setBody("{\"mocked\":true}"));
});
```

This pattern may intercept every API under `/api/`, including endpoints that the test did not intend to mock. For example:

1. Login API may be affected.
2. Permission API may be affected.
3. User profile API may be affected.
4. Feature flag API may be affected.
5. Configuration API may be affected.
6. Dashboard or menu API may be affected.
7. Other tests may receive unintended mocked responses.
8. Real backend contract issues may be hidden.
9. The mocked response may not match the real API schema.
10. The test may pass without proving the real workflow.

A better approach is to mock only the endpoint required for the scenario:

---

```
page.route("**/api/orders/123", route -> {
  route.fulfill(new Route.FulfillOptions()
    .setStatus(200)
    .setContentType("application/json")
    .setBody("{\"id\":123,\"status\":\"APPROVED\"}"));
});
```

This makes the test more intentional. It controls only the order response and does not accidentally affect authentication, permissions, or application configuration.

After mocking the response, the test should still validate the visible user outcome:

```
page.navigate("https://example.com/orders/123");

PlaywrightAssertions.assertThat(
  page.getByText("APPROVED")
).isVisible();
```

The test should not stop at confirming that the route returned status **200**. It should prove that the UI correctly processed the mocked data and displayed the expected result to the user.

If the mock is needed across multiple pages in the same test, `browserContext.route()` can be used, but it should still be scoped to a fresh test context and a narrow URL pattern. It should not be placed globally in a base setup unless every test genuinely requires it.

## Common Mistake

mocking `**/api/**` because it is convenient. Broad mocks can hide integration defects, create false confidence, leak into unrelated tests, and make the UI behave differently from the real application.

---

## Question 1.31

### Why is a static `Page` object dangerous in parallel Playwright Java tests?

#### Interview-Style Answer

A static `Page` object is dangerous because it can be shared across multiple tests, threads, or classes. In parallel execution, one test may navigate, close, or modify the same `Page` while another test is still using it.

In Playwright Java, each test should normally get its own `BrowserContext` and `Page`. This keeps browser state, navigation, cookies, local storage, dialogs, downloads, and tracing isolated, making the suite safer for CI/CD parallel execution.

#### Detailed Explanation

A static `Page` looks convenient, but it creates shared mutable state:

```
public static Page page;
```

This is unsafe because parallel tests may interact with the same browser tab at the same time. For example:

```
Test A opens the Login page.  
Test B navigates the same Page to the Orders page.  
Test A tries to fill the username field.  
Test A fails because the shared Page is no longer on the Login page.
```

This can cause random failures such as wrong page state, closed page errors, unexpected navigation, mixed user sessions, incorrect screenshots, corrupted traces, and tests passing locally but failing in CI.

A safer pattern is to create a fresh `BrowserContext` and `Page` per test:

```
@BeforeEach  
void setup() {  
    context = browser.newContext();  
    page = context.newPage();  
}  
  
@AfterEach  
void cleanup() {  
    if (context != null) {  
        context.close();  
    }  
}
```



---

The **Browser** can often be reused for performance, but **BrowserContext** and **Page** should not be shared across independent tests. A fresh context gives each test isolated cookies, local storage, session storage, permissions, and cache.

This also makes framework ownership clearer. Page Objects should receive the current test's **Page** instance through constructor injection instead of reading a global static page:

```
LoginPage loginPage = new LoginPage(page);  
loginPage.login("buyer@example.com", "password");
```

In a mature framework, code review should reject static **Page**, static **BrowserContext**, and shared mutable test state unless there is a very specific controlled reason. This keeps the suite maintainable, parallel-safe, and easier to debug.

## Common Mistake

making **Page** static for convenience, then blaming Playwright flakiness when tests randomly fail because multiple tests are navigating, clicking, closing, or asserting against the same shared page.

---

---

## **2. Page Object Model**

---

## **Part I - Core Questions**

## Question 2.6

### How do you design Page Object methods in Playwright Java?

#### Interview-Style Answer

Page Object methods should represent meaningful user or business actions, not thin wrappers around every Playwright API call. A method like `loginAs()` or `approveOrder()` is more valuable than generic methods like `clickButton()` or `enterText()`.

In Playwright Java, Page Object methods should hide locator details, keep test flows readable, use scoped `Locator` objects, and expose actions that match how a real user interacts with the page.

#### Detailed Explanation

A good Page Object method should describe intent. The test should read like a business scenario, while the Page Object handles the page-specific locators and interactions.

Good Page Object methods usually:

1. Represent user actions or business actions.
2. Hide locator and DOM details from the test.
3. Use stable role, label, text, or test-id locators.
4. Scope locators to the correct section, row, modal, or component.
5. Avoid unnecessary wrappers around every Playwright method.
6. Return useful page or component objects when navigation or modal opening happens.
7. Include only page-readiness assertions or page-specific validation when appropriate.

Example:

```

class OrdersPage {
    private final Page page;

    OrdersPage(Page page) {
        this.page = page;
    }

    private Locator orderRow(String orderId) {
        return page.getByRole(AriaRole.ROW)
            .filter(new Locator.FilterOptions().setHasText(orderId));
    }

    void approveOrder(String orderId) {
        Locator row = orderRow(orderId);

        row.getByRole(
            AriaRole.BUTTON,
            new Locator.GetByRoleOptions().setName("Approve")
        ).click();
    }
}

```

The test becomes readable:

```

ordersPage.approveOrder(orderId);

PlaywrightAssertions.assertThat(
    page.getByTestId("order-status")
).hasText("Approved");

```

This is better than writing all locators directly in the test or creating generic wrapper methods like:

```

clickButton("Approve");
enterText("Username", "admin");

```

Generic wrappers often hide the real page structure without adding business meaning. They can also make debugging harder because the failure report points to a generic method instead of a clear page action.

Assertions should be placed carefully. Page Objects may include readiness checks such as `waitUntilLoaded()` or page-specific checks such as verifying a modal is open. But the main business assertion is often clearer in the test so reviewers can see exactly what the scenario proves.

## Common Mistake

creating Page Object methods like `clickButton()`, `enterText()`, or `waitForElement()` that simply wrap Playwright APIs without expressing user intent, improving locator scoping, or making the test easier to understand.

## Question 2.13

### What are common Page Object Model mistakes in Playwright Java?

#### Interview-Style Answer

Common Page Object Model mistakes in Playwright Java include creating one giant Page Object, duplicating locators, using weak CSS/XPath selectors everywhere, over-wrapping every Playwright method, hiding all assertions, and mixing page interaction logic with test data, API setup, or cleanup.

A good Playwright Page Object should keep tests readable, locators scoped, methods business-focused, and responsibilities clear. Page Objects should describe how to interact with a page or component, not become a place for all framework, data, API, and assertion logic.

#### Detailed Explanation

Page Object Model improves maintainability only when it is designed with clear responsibility. If it is copied from old Selenium-style frameworks without adapting to Playwright's [Locator](#), auto-waiting, strict mode, and web-first assertions, it can make the suite harder to maintain.

Common mistakes include:

1. Creating one Page Object for the whole application.
2. Duplicating the same locator in many page classes.
3. Not creating reusable component objects for common UI sections.
4. Using raw CSS or XPath everywhere instead of role, label, text, or test-id locators.
5. Creating wrapper methods for every Playwright action such as `click()`, `fill()`, and `waitFor()`.
6. Hiding all assertions inside Page Objects so the test no longer shows what it proves.
7. Generating test data inside Page Objects.
8. Calling API setup or cleanup from Page Objects.
9. Using vague method names like `doSubmit()`, `clickButton()`, or `handlePage()`.
10. Exposing raw locators unnecessarily to every test class.

A better structure is:

1. Page Objects for full pages.
2. Component objects for reusable sections such as header, sidebar, table, modal, or toast.
3. Test classes for scenario intent and key business assertions.
4. API and test-data utilities outside Page Objects.
5. Stable Playwright locators with proper scoping.
6. Business-readable methods such as `submitOrder()`, `approveInvoice()`, or `openCustomer()`.

---

## Example of a focused Page Object method:

```
public class OrdersPage {
    private final Page page;

    public OrdersPage(Page page) {
        this.page = page;
    }

    private Locator orderRow(String orderId) {
        return page.getByTestId("orders-table")
            .getByRole(
                AriaRole.ROW,
                new
                    Locator.GetByRoleOptions().setName(Pattern.compile(order
                        Id))
            );
    }

    public void openOrder(String orderId) {
        orderRow(orderId).getByRole(
            AriaRole.LINK,
            new Locator.GetByRoleOptions().setName("View")
        ).click();
    }
}
```

The test should still show the business intent and important assertion:

```
ordersPage.openOrder(orderId);

PlaywrightAssertions.assertThat(
    page.getByTestId("order-status")
).hasText("Approved");
```

This keeps responsibilities clear. The Page Object handles page interaction, while the test expresses what business behavior is being validated.

## Common Mistake

copying Selenium-style Page Object patterns directly into Playwright Java, creating heavy wrapper classes and stale-element-style abstractions instead of using Playwright's [Locator](#), auto-waiting, strict mode, scoping, and web-first assertions properly.

---

## Question 2.15

**A Page Object has 80 methods and controls many unrelated screens. What would you improve?**

### Interview-Style Answer

I would refactor it into smaller Page Objects and component objects based on page ownership, reusable UI widgets, and business responsibility. A Page Object with 80 methods controlling unrelated screens is usually doing too much and becomes difficult to review, maintain, and debug.

I would keep each Page Object focused on one page or logical screen. Repeated UI parts such as tables, filters, dialogs, menus, tabs, and pagination should be moved into component classes. API setup, test data generation, environment logic, and cleanup should be moved out of Page Objects into separate utilities or fixtures.

### Detailed Explanation

A bloated Page Object is a sign that the framework structure is not well separated. If one class controls dashboard, orders, invoices, reports, settings, filters, dialogs, uploads, downloads, and admin actions, any change in the application can make that class risky to modify.

A good refactoring approach is:

1. Identify separate pages or screens.
2. Split page-specific methods into separate Page Objects.
3. Move reusable widgets into component classes.
4. Move API setup and cleanup logic out of Page Objects.
5. Keep Page Object methods focused on user actions and validations.
6. Remove duplicate locator logic.
7. Give each class clear ownership.
8. Keep test methods readable at business-flow level.

For example, instead of one large `DashboardPage`, I would split it like this:

```
DashboardPage
OrdersPage
OrderDetailsPage
InvoicesPage
ReportsPage
FilterPanelComponent
PaginationComponent
DataTableComponent
ConfirmationDialogComponent
UploadComponent
```



If multiple pages use the same table behavior, I would create a reusable table component:

```
public class DataTableComponent {
    private final Page page;

    public DataTableComponent(Page page) {
        this.page = page;
    }

    public Locator rowByText(String text) {
        return page.getByRole(AriaRole.ROW)
            .filter(new Locator.FilterOptions().setHasText(text));
    }

    public void clickRowAction(String rowText, String actionName) {
        rowByText(rowText).getByRole(
            AriaRole.BUTTON,
            new Locator.GetByRoleOptions().setName(actionName)
        ).click();
    }
}
```

Then the page can expose business-specific methods:

```
public class OrdersPage {
    private final DataTableComponent table;

    public OrdersPage(Page page) {
        this.table = new DataTableComponent(page);
    }

    public void approveOrder(String orderId) {
        table.clickRowAction(orderId, "Approve");
    }

    public void shouldShowOrderStatus(String orderId, String status) {
        PlaywrightAssertions.assertThat(
            table.rowByText(orderId)
        ).containsText(status);
    }
}
```

This keeps the test readable:

```
ordersPage.approveOrder("ORD-1001");
ordersPage.shouldShowOrderStatus("ORD-1001", "Approved");
```

The test now describes the business scenario, while the locator and component details stay in the right layer.

## Common Mistake

putting the whole application into one [HomePage](#) or [DashboardPage](#). This creates a large, fragile class with unclear ownership; smaller Page Objects and reusable components make the Playwright Java framework easier to maintain and scale.

---

## **3. Framework Design**

---

## **Part I - Core Questions**

### Question 3.8

**Can a `Browser` be shared across parallel Playwright Java tests?**

### Interview-Style Answer

Yes, a `Browser` can often be shared across parallel Playwright Java tests for performance, but each test should still create its own isolated `BrowserContext` and `Page`.

The `Browser` represents the browser process. The `BrowserContext` represents an isolated browser session with its own cookies, local storage, session storage, permissions, cache, and storage state. Sharing the browser is usually fine; sharing the context or page across tests is dangerous.

### Detailed Explanation

A good parallel-safe lifecycle separates what can be shared from what must be isolated:

```
Shared:
- Playwright
- Browser

Per test:
- BrowserContext
- Page
- Test data
- Downloads/uploads/temp files
- Storage state when needed
```

Example:

```
BrowserContext context = browser.newContext();
Page page = context.newPage();
```

This gives each test its own clean session without launching a completely new browser process for every test. It improves execution speed while still keeping browser-side state isolated.

A typical setup can reuse the `Browser` and create a fresh context per test:

---

```
@BeforeEach
void setup() {
    context = browser.newContext();
    page = context.newPage();
}

@AfterEach
void cleanup() {
    if (context != null) {
        context.close();
    }
}
```

This is safer than sharing one [Page](#) or one [BrowserContext](#) because parallel tests may navigate, log in, open popups, download files, or modify storage independently.

For authenticated tests, use role-specific or worker-specific storage state carefully:

```
BrowserContext context = browser.newContext(
    new Browser.NewContextOptions()
        .setStorageStatePath(Paths.get("auth/buyer-worker-1.json"))
);
```

Even with separate contexts, backend state must also be isolated. If two tests use the same user, cart, order, or file path, they can still interfere with each other.

## Common Mistake

assuming that sharing a [Browser](#) means sharing a session. Session data belongs to [BrowserContext](#), not the browser process, so tests should share the browser only while keeping each test's context, page, data, and files isolated.

---

### Question 3.12

## What Java framework utilities must be thread-safe for parallel Playwright execution?

### Interview-Style Answer

For parallel Playwright Java execution, utilities that manage shared state, files, data, browser lifecycle, reporting, API calls, and configuration must be thread-safe. This includes configuration readers, test data utilities, API clients, report loggers, artifact writers, storage-state managers, cleanup utilities, and any browser/page manager.

The main rule is simple: a utility should not use mutable static state unless it is intentionally synchronized, immutable, or isolated per test/thread. Otherwise, parallel tests may overwrite each other's data, artifacts, sessions, or reports.

### Detailed Explanation

Thread-safety matters because multiple tests may run at the same time in JUnit, TestNG, Maven Surefire, Gradle, or CI workers. If a utility stores shared mutable values, one test can accidentally change the state used by another test.

Utilities that need careful thread-safe design include:

1. ConfigReader
2. Browser/Page manager
3. Test data generator
4. API client
5. Report logger
6. Screenshot utility
7. Download utility
8. Storage-state manager
9. Cleanup utility
10. Environment helper
11. Trace/video artifact writer
12. Temporary file manager

A dangerous pattern is using static mutable fields for [Page](#), test data, current user, current test name, download path, or report status:

```
public class PageManager {  
    public static Page page;  
}
```

In parallel execution, this can cause one test to overwrite another test's page reference. A safer approach is to keep [Page](#) and [BrowserContext](#)

---

per test, or use `ThreadLocal` only when the framework genuinely needs thread-bound access:

```
public class PageManager {
    private static final ThreadLocal<Page> threadPage = new ThreadLocal<>();

    public static void setPage(Page page) {
        threadPage.set(page);
    }

    public static Page getPage() {
        return threadPage.get();
    }

    public static void removePage() {
        threadPage.remove();
    }
}
```

File-related utilities should also create unique paths per test or worker:

```
Path screenshotPath = Paths.get(
    "target/screenshots",
    testName + "-" + UUID.randomUUID() + ".png"
);
```

API clients should avoid storing request-specific data such as tokens, user IDs, payloads, or created record IDs in shared static variables. Test data and cleanup utilities should track records owned by the current test so cleanup does not delete another parallel test's data.

A mature framework should review all utilities for shared state, unique file paths, role-specific storage state, parallel-safe users, and cleanup boundaries before enabling parallel execution in CI/CD.

## Common Mistake

using static mutable fields inside framework utilities for convenience, such as current `Page`, current user, token, test data ID, artifact path, or report object, causing random failures and corrupted results during parallel execution.

---

### Question 3.22

**A junior engineer added a helper that automatically retries every click three times. How would you review it?**

### Interview-Style Answer

I would reject it as a general framework rule because automatic click retries can hide real problems in locator design, actionability, timing, test data, or application behavior. Playwright already performs auto-waiting before actions such as `click()`, so wrapping every click with custom retries usually weakens the framework instead of improving stability.

I would ask the engineer to investigate why the click is failing. The correct fix may be a better locator, a proper business-state assertion, a scoped row action, a frame correction, a disabled-button check, or a real product bug. Retries should be controlled and exceptional, not applied blindly to every click.

### Detailed Explanation

Playwright's `Locator.click()` already waits for actionability conditions within the configured timeout. It checks that the element is attached, visible, stable, enabled, and able to receive events. If a click fails, the framework should treat that failure as evidence, not immediately retry and hide it.

A generic helper like this is risky:

```
void retryClick(Locator locator) {
    for (int i = 0; i < 3; i++) {
        try {
            locator.click();
            return;
        } catch (Exception ignored) {
            // retry
        }
    }
}
```

Problems with automatic click retries:



1. It hides weak or non-unique locators.
2. It can mask real actionability problems.
3. It makes failures slower.
4. It can create duplicate actions such as double submit or double payment.
5. It may click after the page state has changed.
6. It makes Trace Viewer analysis harder.
7. It can hide application bugs such as buttons becoming enabled too early.
8. It encourages engineers to avoid root-cause debugging.

A better review comment would be: remove the generic retry wrapper and fix the actual synchronization or locator issue.

Better pattern:

```
Locator submitButton = page.getByRole(
    AriaRole.BUTTON,
    new Page.GetByRoleOptions().setName("Submit")
);

PlaywrightAssertions.assertThat(submitButton).isEnabled();

submitButton.click();

PlaywrightAssertions.assertThat(
    page.getByText("Order submitted successfully")
).isVisible();
```

For event-based flows, use the correct Playwright wait instead of retrying clicks:

```
Response response = page.waitForResponse(
    res -> res.url().contains("/api/orders") && res.status() == 200,
    () -> {
        page.getByRole(
            AriaRole.BUTTON,
            new Page.GetByRoleOptions().setName("Submit Order")
        ).click();
    }
);

PlaywrightAssertions.assertThat(
    page.getByText("Order submitted successfully")
).isVisible();
```

If retries are needed at all, they should be handled at test or pipeline level for known transient infrastructure issues, with trace, screenshot, video, logs, and retry status clearly reported. They should not be hidden inside every click helper.

## Common Mistake

adding Selenium-style retry wrappers around Playwright actions. This hides the real failure cause and can make tests less reliable than using Playwright's locator model, auto-waiting, web-first assertions, and Trace Viewer evidence correctly.

### Question 3.32

**How would you migrate a framework from CSS/XPath-heavy locators to Playwright's recommended locator strategy?**

#### Interview-Style Answer

I would migrate gradually, starting with high-value, high-failure, and high-maintenance tests. I would replace fragile CSS/XPath selectors with role, label, text, test-id, and scoped locators that describe the user-facing identity of the element.

The goal is not only to change selector syntax, but to improve test meaning, strictness, readability, and long-term maintainability.

#### Detailed Explanation

A locator migration should be prioritized instead of rewriting the whole framework blindly. Start with tests that fail often, cover critical business flows, or contain brittle selectors tied to DOM position or styling.

Migration priority:

1. Frequently failing tests
2. Critical release or smoke tests
3. High-maintenance pages
4. Repeated table/list actions
5. Accessibility-sensitive forms
6. Newly created tests
7. Page Objects with duplicated selectors

Weak locator:

```
page.locator("//div[3]/button[2]").click();
```

Better locator:

```
Locator row = page.getByRole(AriaRole.ROW)
    .filter(new Locator.FilterOptions().setHasText("INV-1001"));

row.getByRole(
    AriaRole.BUTTON,
    new Locator.GetByRoleOptions().setName("Approve")
).click();
```

This version is better because it expresses the user intent: find invoice **INV-1001** and click its **Approve** button. It is also scoped to the correct row, which reduces accidental matches when multiple **Approve** buttons exist on the page.

---

For forms, prefer label-based locators:

```
page.getByLabel("Email").fill("buyer@example.com");  
page.getByLabel("Password").fill("secret");
```

For important stable elements that do not have good accessible names, use test ids:

```
page.getByTestId("order-status").click();
```

A good migration should also include review rules. New tests should avoid fragile DOM-position XPath, styling-based CSS selectors, and broad locators unless there is a clear reason. Page Objects should own locator changes so tests become more readable and consistent.

### Common Mistake

mechanically converting CSS/XPath selectors to another syntax without improving locator meaning, scoping, strictness, accessibility alignment, or business readability.

---

### Question 3.33

**A team wants to use only `data-testid` locators for all Playwright tests. What would you recommend?**

### Interview-Style Answer

I would recommend a balanced locator strategy. `data-testid` is useful as a stable testability hook, especially when text is dynamic, localized, duplicated, or when the element needs a reliable technical identifier. But I would not recommend using only `data-testid` for every test.

Playwright tests should prefer user-facing locators such as role, label, text, and accessible name when they clearly represent how a user interacts with the application. These locators improve readability and can also reveal accessibility or semantic HTML issues. `data-testid` should be used deliberately where user-facing locators are not stable or unique enough.

### Detailed Explanation

Using only `data-testid` can make tests stable, but it can also disconnect the tests from real user behavior. A user does not interact with `data-testid`; the user sees buttons, labels, headings, links, forms, and accessible names. Playwright's role and label locators help validate that the UI is not only present but also exposed meaningfully.

A balanced locator priority can be:

1. Role with accessible name
2. Label for form fields
3. Meaningful visible text
4. Placeholder where appropriate
5. Test ID for stable technical hooks
6. Scoped CSS when needed
7. XPath as a last resort

For example, if the button has a clear accessible name, this is usually better:

```
page.getByRole(  
  AriaRole.BUTTON,  
  new Page.GetByRoleOptions().setName("Submit")  
)<code>.click();
```

This validates that the button is exposed as a button and has the expected accessible name.

For a form field, `getByLabel()` is also strong:

---

```
page.getByLabel("Email").fill("raj@example.com");
```

This proves that the input is connected to a meaningful label, which is good for accessibility and user experience.

However, `data-testid` is useful in many real projects. For example, dashboards, charts, repeated cards, icon-only buttons, dynamic text, translated applications, and complex tables may need stable hooks:

```
page.getByTestId("invoice-status").click();
```

or:

```
PlaywrightAssertions.assertThat(  
    page.getByTestId("order-total")  
).hasText("₹12,500");
```

I would use `data-testid` when:

- UI text changes frequently.
- The application supports multiple languages.
- Several elements have the same visible text.
- Accessibility name is not unique.
- The element is a non-user-facing technical container.
- Stable business identity is needed for a component.
- Complex widgets are difficult to locate reliably by role or text alone.

For iframe-based pages, the locator strategy still needs to respect the frame boundary. `frameLocator()` should target the iframe first, and then role, label, text, or test-id locators should be chained inside that iframe:

```
FrameLocator paymentFrame = page.frameLocator("iframe[title='Secure  
payment']");  
  
paymentFrame.getByTestId("card-number").fill("4111111111111111");
```

Here, `frameLocator()` targets the iframe, and `getByTestId()` locates the actual element inside that iframe.

## Common Mistake

using only `data-testid` to make tests stable while completely ignoring user-visible behavior and accessibility. A strong Playwright locator strategy should balance stability, readability, strictness, scoping, accessibility, and maintainability.

---

### Question 3.40

**How would you prevent storage-state files from becoming stale or invalid?**

### Interview-Style Answer

I would prevent storage-state files from becoming stale by generating them through a controlled authentication setup, storing them separately by role and environment, validating the logged-in identity before running dependent tests, and regenerating them when they expire or become invalid.

In Playwright Java, storage state should be treated as temporary authentication data, not permanent test data. The framework should fail early if the loaded state does not represent the expected user, role, tenant, or environment.

### Detailed Explanation

Storage-state files are useful because they allow tests to reuse an authenticated session without logging in through the UI for every test. However, they can become invalid when sessions expire, passwords change, cookies are cleared by the backend, permissions change, environments are refreshed, or the wrong state file is used for the wrong environment.

A safe strategy is to generate storage state in a setup step:

```
BrowserContext context = browser.newContext();
Page page = context.newPage();

page.navigate(baseUrl + "/login");

page.getByLabel("Email").fill(adminEmail);
page.getByLabel("Password").fill(adminPassword);
page.getByRole(
    AriaRole.BUTTON,
    new Page.GetByRoleOptions().setName("Login")
).click();

PlaywrightAssertions.assertThat(
    page.getByRole(
        AriaRole.HEADING,
        new Page.GetByRoleOptions().setName("Dashboard")
    )
).isVisible();

context.storageState(
    new BrowserContext.StorageStateOptions()
        .setPath(Paths.get("auth/admin-dev.json"))
);
```

---

The file should be named clearly by role and environment, for example:

```
auth/admin-dev.json
auth/buyer-dev.json
auth/approver-qa.json
auth/reviewer-staging.json
```

After loading storage state, the test should validate that the session is still valid before running business steps:

```
Browser.NewContextOptions options = new Browser.NewContextOptions()
    .setStorageStatePath(Paths.get("auth/admin-dev.json"));

BrowserContext context = browser.newContext(options);
Page page = context.newPage();

page.navigate(baseUrl + "/profile");

PlaywrightAssertions.assertThat(
    page.getByText("Admin")
).isVisible();
```

This validation prevents misleading failures. Without it, a test may fail later on a dashboard, invoice, approval, or settings page, while the real issue is that the user is unauthenticated or has the wrong role.

For parallel execution, storage-state files should not be overwritten by multiple workers at the same time. A setup job should create them before tests start, or each worker should generate its own worker-specific state file when needed.

Good practices include:

1. Generate storage state from a controlled setup flow.
2. Keep files separate by role, environment, tenant, and worker when required.
3. Validate the logged-in user and role after context creation.
4. Regenerate state when login validation fails.
5. Avoid sharing one mutable state file across parallel workers.
6. Store state files securely because they may contain cookies or tokens.
7. Exclude generated auth files from Git using .gitignore.
8. Avoid storing storage state inside traces, reports, or public artifacts.
9. Rotate credentials or tokens if state files are accidentally exposed.

Storage state should improve speed, but it should not reduce security or reliability. A mature framework treats authentication setup as a first-class part of execution, with clear ownership, regeneration rules, and early validation.

## Common Mistake

Blindly loading an old `auth/admin.json` file and debugging page-level failures later, when the real problem is that the stored cookies or tokens

---

expired, the user role changed, or the state file belongs to a different environment.

---



---

## **Book 4 - Architect/Leadership**

---

## **1. Architect**

---

## **Part I - Core Questions**

## Question 1.6

**How would you identify the top 20% of tests that provide 80% of regression value?**

### Interview-Style Answer

I would identify the top 20% regression tests by mapping each test to business-critical workflows, production usage, customer impact, defect history, release-blocking value, compliance risk, and integration importance.

The most valuable tests are not always the fastest or longest tests. They are the tests that give the highest release confidence. For example, login, checkout, payment, approval, permission checks, data submission, critical reporting, and third-party integrations often provide high regression value because failure in these areas directly affects users or business operations.

I would score tests using objective signals such as escaped defects, production usage, module criticality, failure impact, and how often the test detects real regressions. These tests can become the core smoke or critical regression suite.

### Detailed Explanation

The top 20% of regression tests should be selected based on release confidence, not simply based on test count or execution speed. A short test may be valuable if it protects a critical permission rule. A long test may be low-value if it only repeats coverage already tested elsewhere.

I would start by building a test inventory with details such as module, business workflow, owner, runtime, flakiness, defect history, production usage, customer impact, and execution frequency. Then I would rank tests by risk and value.

Useful evaluation factors include:

1. Is this workflow used frequently in production?
2. Does failure block revenue, compliance, security, or customer operations?
3. Has this area caused escaped defects before?
4. Does this test cover a critical integration point?
5. Does it validate permissions or data integrity?
6. Does it detect real regressions, or mostly duplicate other tests?
7. Is it stable enough to support release decisions?
8. Is the same behavior already covered at API, unit, or component level?

Example scoring model:

```

record RegressionValue(
    String testName,
    int businessCriticality,
    int productionUsage,
    int escapedDefectHistory,
    int customerImpact,
    int integrationRisk,
    int flakinessCost
) {
    int score() {
        return businessCriticality
            + productionUsage
            + escapedDefectHistory
            + customerImpact
            + integrationRisk
            - flakinessCost;
    }
}

RegressionValue test = new RegressionValue(
    "approveHighValuePurchaseRequest",
    5,
    4,
    4,
    5,
    4,
    1
);

Assertions.assertTrue(test.score() >= 15);

```

High-value Playwright tests usually validate end-to-end user-visible outcomes. For example, a payment test should not only check that the Pay button is visible. It should verify that payment succeeds, the order status changes, the confirmation is shown, and the user can see the final result.

Once identified, these top-value tests should be kept fast, stable, owned, and reviewed carefully. They are good candidates for PR smoke, pre-release validation, or deployment gates. Lower-value tests can run nightly, be moved to API/component layers, consolidated, or removed if they duplicate stronger coverage.

This review should be repeated periodically because product risk changes over time. A workflow that was critical during rollout may become stable later, while a new module, integration, or customer-specific feature may become more important.

## Common Mistake

choosing the critical regression suite only because tests are fast, without checking whether they protect high-risk workflows, customer impact, production usage, permissions, integrations, or historically defect-prone areas.

## Question 1.11

### What principles would you follow for risk-based release automation with Playwright Java?

#### Interview-Style Answer

I would design risk-based release automation around release confidence, not just test volume. The focus should be on critical business workflows, production-like integration coverage, clear failure evidence, controlled edge-case simulation, and fast feedback for release decisions.

In Playwright Java, I would prioritize high-risk journeys such as login, checkout, payment, order creation, approval flows, role-based access, and customer-facing regression areas. These flows should run against real backend services where release confidence depends on integration behavior, while `page.route()` or `browserContext.route()` can be used carefully for rare edge cases, error responses, or hard-to-create states.

#### Detailed Explanation

Risk-based release automation should answer one main question: “Do we have enough confidence to release this build safely?”

The suite should not simply run the maximum number of tests. It should run the right tests at the right stage of the pipeline. High-risk, high-impact flows should be given priority because failures there can block users, affect revenue, break compliance, or damage customer trust.

Key principles:

1. Prioritize business-critical workflows.
2. Cover high-risk integrations with real backend behavior.
3. Use controlled mocks only for specific edge cases.
4. Validate visible user outcomes, not only technical events.
5. Keep release tests stable, fast, and maintainable.
6. Capture trace, screenshot, video, console logs, and network evidence for failures.
7. Classify failures before making release decisions.
8. Treat flaky tests as release risk, not as harmless noise.
9. Separate smoke, critical regression, full regression, and exploratory coverage.
10. Review escaped defects and add or improve coverage based on real production risk.

---

For example, a release smoke suite may verify that a buyer can log in, search for a product, place an order, and see the confirmation message. The test should validate user-visible results such as order number, confirmation status, updated dashboard, or email trigger indicator, not just that an API returned 200.

```
page.getByRole(AriaRole.BUTTON,
  new Page.GetByRoleOptions().setName("Place Order")
).click();

PlaywrightAssertions.assertThat(
  page.getByText("Order placed successfully")
).isVisible();

PlaywrightAssertions.assertThat(
  page.getByTestId("order-number")
).isVisible();
```

For rare scenarios, such as payment service timeout or inventory API failure, mocking may be appropriate:

```
page.route("**/api/inventory/**", route =>
  route.fulfill(new Route.FulfillOptions()
    .setStatus(503)
    .setContentType("application/json")
    .setBody("{\"message\":\"Inventory service unavailable\"}"))
);

page.navigate(baseUrl + "/checkout");

PlaywrightAssertions.assertThat(
  page.getByText("Inventory service unavailable")
).isVisible();
```

The release decision should be based on risk signals such as failed critical flows, flaky-test trends, defect history, changed modules, failed integrations, environment instability, and quality of failure evidence. A 98% pass rate is not enough if the failed 2% includes checkout, payment, login, or approval workflows.

## Common Mistake

treating risk-based release automation as running the largest possible suite before release. A better approach is to run a focused, stable, evidence-rich suite that protects the most important user journeys and gives stakeholders clear release risk information.

---

### Question 1.36

**How would you design a framework policy for using storage state across desktop and mobile Playwright Java suites?**

### Interview-Style Answer

I would define storage state as an authentication optimization, not as a replacement for proper desktop or mobile context setup. Storage state can help avoid repeated login, but each suite should still create a fresh `BrowserContext` with explicit profile settings such as viewport, user agent, permissions, locale, geolocation, and device-related options.

For desktop and mobile suites, I would keep storage state files role-specific, environment-specific, and refreshable. After loading storage state, every test should validate that the correct authenticated landing page and profile-specific UI are displayed.

### Detailed Explanation

Storage state is useful for saving login cookies and local storage after authentication. In Playwright Java, it is commonly used to speed up tests by avoiding repeated UI login flows. However, it should be controlled carefully because authentication state can become stale, unsafe, or misleading if reused across too many suites.

A good framework policy should define:

1. How storage state is generated.
2. Which roles have saved storage-state files.
3. Whether files are environment-specific.
4. How often storage state is refreshed.
5. Whether local storage values are allowed.
6. Whether storage state can be reused across desktop and mobile profiles.
7. How tests validate the authenticated landing state.
8. How storage-state files are protected from accidental modification.
9. How CI secrets and credentials are managed.
10. How expired or invalid state is detected and regenerated.

Example policy:



1. Generate storage state using a clean login setup flow.
2. Store separate files for each role, such as admin, buyer, seller, and viewer.
3. Store separate files per environment if needed, such as QA, staging, and pre-prod.
4. Treat storage-state files as read-only during normal test execution.
5. Create a fresh BrowserContext for every test.
6. Apply desktop, tablet, or mobile profile options explicitly when creating the context.
7. Do not use storage state as a shortcut for mobile emulation.
8. Validate the post-login UI after loading state.
9. Regenerate storage state when login tokens expire.
10. Never commit sensitive storage-state files to Git.

### Example for desktop:

```
BrowserContext desktopContext = browser.newContext(  
    new Browser.NewContextOptions()  
        .setStorageStatePath(Paths.get("auth/buyer-qa.json"))  
        .setViewportSize(1366, 768)  
);  
  
Page page = desktopContext.newPage();  
page.navigate(baseUrl + "/dashboard");  
  
PlaywrightAssertions.assertThat(  
    page.getByRole(AriaRole.HEADING,  
        new Page.GetByRoleOptions().setName("Dashboard"))  
).isVisible();
```

### Example for mobile:

```
BrowserContext mobileContext = browser.newContext(  
    new Browser.NewContextOptions()  
        .setStorageStatePath(Paths.get("auth/buyer-qa.json"))  
        .setViewportSize(390, 844)  
        .setIsMobile(true)  
        .setHasTouch(true)  
);  
  
Page page = mobileContext.newPage();  
page.navigate(baseUrl + "/dashboard");  
  
PlaywrightAssertions.assertThat(  
    page.getByRole(AriaRole.BUTTON,  
        new Page.GetByRoleOptions().setName("Menu"))  
).isVisible();
```

The same authenticated role may sometimes reuse the same storage state across desktop and mobile if the application authentication model supports it. But the framework should not assume that the UI behavior, layout, navigation, permissions, or local storage behavior will be identical across profiles.

For mobile tests, the suite should still verify mobile-specific behavior such as hamburger menu visibility, responsive layout, touch-friendly controls, mobile navigation, and viewport-specific content. For desktop tests, it should verify desktop navigation, wider layout, and

---

desktop-only controls where relevant.

### **Common Mistake**

using one old storage-state file for every role, browser, device profile, and environment, then assuming the test is valid without verifying the authenticated landing page and profile-specific UI after context creation.

---

---

### Question 1.49

**What code-review checklist would you use for Playwright Java pull requests?**

### Interview-Style Answer

I would review whether the pull request improves test value without reducing suite stability. My checklist would cover scenario relevance, locator quality, assertion strength, wait strategy, test-data isolation, [BrowserContext](#) lifecycle, Page Object/component design, network mocking, security, parallel execution safety, CI impact, and failure diagnostics.

For Playwright Java, I would especially check that the test uses stable [Locator](#) strategies, meaningful web-first assertions, no unnecessary hard waits, isolated state per test, scoped mocks, and useful artifacts such as trace, screenshot, video, or logs when failures occur.

### Detailed Explanation

A Playwright Java code review should check more than whether the test passes locally. It should verify that the test is valuable, stable, maintainable, secure, and safe to run repeatedly in CI/CD.

A practical checklist:

1. Does the test validate a meaningful user or business risk?
2. Is the test name clear and behavior-focused?
3. Are locators stable, readable, and user-facing where possible?
4. Are role locators, labels, text, test ids, or scoped locators used properly?
5. Are strict mode violations solved by better targeting instead of random `nth()` usage?
6. Are hard waits avoided?
7. Are waits based on visible UI state, request/response events, downloads, popups, or meaningful conditions?
8. Are assertions validating final user-visible outcomes?
9. Is test data unique, controlled, or cleaned up?
10. Is each test using a safe `BrowserContext/Page` lifecycle?
11. Are storage-state files role-safe and not leaking between users?
12. Are Page Objects or components small, readable, and responsibility-focused?
13. Are network mocks scoped to the exact endpoint and scenario?
14. If real backend traffic is used, does the test validate the visible UI result?
15. Are credentials, tokens, traces, screenshots, videos, logs, and reports handled securely?
16. Can the test run safely in parallel?
17. Are downloads, uploads, temp files, and generated data isolated?
18. Will the test behave consistently in CI and local execution?
19. Are failure artifacts enabled for debugging?
20. Does the change increase long-term maintainability instead of hiding flakiness?

Example of a better review comment:

```
Avoid page.locator("button").nth(2).click().  
Use a user-facing locator that describes the action clearly.
```

Improved Playwright Java style:

```
page.getByRole(  
    AriaRole.BUTTON,  
    new Page.GetByRoleOptions().setName("Submit Order")  
)<click();  
  
PlaywrightAssertions.assertThat(  
    page.getByText("Order submitted successfully")  
)<isVisible();
```

For framework-level pull requests, I would review ownership and governance also. For example, shared helpers should not hide flaky behavior, bypass actionability unnecessarily, or make debugging harder. A helper should improve readability and consistency without removing control from the test.

For network-related changes, I would check whether the PR clearly separates request interception, request modification, response mocking, and real backend validation. A mock should not be so broad that it hides integration issues, and the test should still assert what the user sees on the page.

---

## Common Mistake

approving a Playwright Java pull request only because it passes locally. A good review must check whether the test will remain stable, readable, secure, and trustworthy in parallel CI execution.

---

## Question 1.58

**How would you build a Playwright automation strategy for a product moving from manual testing to continuous testing?**

### Interview-Style Answer

I would build the Playwright automation strategy in phases: understand the product risks, identify critical workflows, create a stable framework foundation, automate high-value smoke tests first, integrate them into CI/CD, then expand into risk-based regression coverage.

I would avoid converting every manual test case directly into automation. Continuous testing needs reliable, fast, maintainable tests that give useful release feedback. The strategy should define `BrowserContext` and `Page` lifecycle, locator standards, test data setup, storage-state usage, artifact capture, failure ownership, review rules, and pipeline execution rules.

The goal is to move from manual regression dependency to a trusted automation suite that supports every pull request, nightly build, and release decision.

### Detailed Explanation

Moving from manual testing to continuous testing should start with business risk, not test count. I would first identify the most important workflows: login, permissions, checkout, payment, approvals, reports, search, role-based access, and integrations that frequently break or block releases.

The first phase is framework foundation. In Playwright Java, this means setting up Maven or Gradle, JUnit or TestNG, browser configuration, environment handling, reusable fixtures, `BrowserContext` creation, `Page` lifecycle, Page Objects or component objects, locator standards, and reporting. The team should prefer role locators, accessible names, stable test IDs where needed, and `PlaywrightAssertions` for web-first assertions.

The second phase is smoke automation. I would automate a small number of stable, high-value tests and run them in the PR pipeline. These tests should be fast, reliable, and parallel-safe. Each test should use isolated `BrowserContext` instances, controlled test data, separate users where needed, safe storage-state files, and predictable cleanup.

---

## Example execution strategy:

```
record PipelineStage(String stage, String scope, String trigger) {}

PipelineStage prSmoke = new PipelineStage(
    "PR Smoke",
    "Critical login, navigation, and core business flows",
    "Every pull request"
);

PipelineStage nightlyRegression = new PipelineStage(
    "Nightly Regression",
    "Broader risk-based regression suite",
    "Every night"
);

PipelineStage releaseCertification = new PipelineStage(
    "Release Certification",
    "Critical cross-browser and integration-heavy flows",
    "Before production release"
);
```

The third phase is CI/CD integration. The PR pipeline should run a small smoke suite. The nightly pipeline can run broader regression. The release pipeline can include critical cross-browser coverage, selected visual checks, and integration-heavy workflows. Failures should produce useful evidence such as Trace Viewer artifacts, screenshots, videos, logs, and network details.

The fourth phase is expansion. I would add tests based on release risk, defect history, customer impact, and manual regression effort. API setup can be used to create data quickly, but the UI test should still validate visible user outcomes such as confirmation messages, updated records, correct permissions, or final order status.

The fifth phase is team maturity. Developers, QA engineers, and automation engineers should share ownership. Pull request review rules should check locator quality, test data isolation, Page Object boundaries, assertion quality, and whether the test belongs in PR smoke, nightly regression, or release certification.

## Common Mistake

Automating a large manual regression pack before stabilizing the framework, CI pipeline, test data strategy, ownership model, artifact capture, and flaky test process, which creates a slow and unreliable suite that teams stop trusting.

---

## Question 1.64

### How do you decide which test cases should be automated using Playwright?

#### Interview-Style Answer

I choose Playwright automation for stable, repeatable, high-value browser workflows where real UI behavior, integration, cross-browser behavior, or end-to-end release confidence matters.

Good candidates include smoke flows, critical regression paths, role-based validations, permission checks, frequent release workflows, browser-specific behavior, downloads, uploads, popups, and important UI/API integration journeys. Poor candidates include unstable features, one-time checks, subjective visual judgment, exploratory testing, and backend logic that is better tested through unit, API, or component tests.

The decision should be based on business value, risk, repeatability, stability, maintainability, and whether Playwright UI automation is the right test layer.

#### Detailed Explanation

Not every test case should become a Playwright UI test. Playwright is powerful, but full browser automation has higher execution and maintenance cost than unit, API, or component tests. So I would automate scenarios where real browser behavior and user workflow confidence are important.

Good Playwright candidates include workflows such as login, checkout, payment, approvals, purchase request submission, role-based access, search, report download, file upload, order status updates, and cross-browser regression. These scenarios are valuable because they validate how the user actually interacts with the application.

Example:



---

```
@Tag("smoke")
@Test
void buyerCanSubmitPurchaseRequest() {
    purchaseRequestPage.open();

    purchaseRequestPage.submitValidRequest();

    PlaywrightAssertions.assertThat(
        page.getByTestId("request-status")
    ).hasText("Submitted");
}
```

This is a good Playwright candidate because it validates a critical user journey and checks the final business outcome.

I would avoid using Playwright for scenarios that do not need browser validation. For example, a tax calculation, pricing rule, permission matrix, or data transformation may be better tested through unit or API tests. Playwright can still have one UI test to verify that the calculated result is visible to the user, but the detailed rule coverage should usually live at a lower layer.

I would also check whether the test can run reliably in CI. A good Playwright test should have stable locators, meaningful assertions, isolated test data, safe [BrowserContext](#) usage, and predictable cleanup. If a feature is still changing every day, I may delay UI automation or create only a small smoke check until the workflow stabilizes.

## Common Mistake

Choosing Playwright test cases based only on manual test count, instead of selecting stable, repeatable, high-risk browser workflows that improve release confidence and defect detection value.

---

## Question 1.68

**How do you define a “high-quality Playwright test” in practical terms?**

### Interview-Style Answer

A high-quality Playwright test has clear business value, stable locators, meaningful assertions, isolated test data, reliable synchronization, readable structure, and useful failure evidence.

It is not enough that the test passes consistently. A good test should protect a real business risk and fail for the right reason. The test name should clearly describe the behavior or rule being validated. Locators should be user-focused, using role, label, accessible name, stable text, or agreed `data-testid` values where appropriate.

The test should also run safely in CI, support parallel execution, avoid hidden dependencies, and provide traces, screenshots, videos, logs, or network evidence when it fails.

### Detailed Explanation

A high-quality Playwright test gives confidence when it passes and useful information when it fails. It should validate something that matters to the product, such as login, permissions, checkout, payment, approvals, saved data, status changes, downloads, uploads, or important user workflows.

The test should be readable. A reviewer should quickly understand the Arrange, Act, and Assert flow:

Arrange: Prepare the required state.  
Act: Perform the user behavior.  
Assert: Verify the expected business outcome.

Example:

```
@Test
void approverCanApproveSubmittedRequest() {
    // Arrange
    String requestId = api.createSubmittedRequest("Laptop");

    // Act
    approvalPage.open(requestId);
    approvalPage.approve();

    // Assert
    PlaywrightAssertions.assertThat(
        page.getByTestId("request-status")
    ).hasText("Approved");

    PlaywrightAssertions.assertThat(
        page.getByRole(AriaRole.STATUS)
    ).containsText("Approved");
}
```

This test is high quality because it prepares its own data, performs a clear user action, and verifies the final business result. It does not depend on execution order or random existing data.

A high-quality test should also use reliable synchronization. It should avoid `Thread.sleep()` and `page.waitForTimeout()` for normal waits. Instead, it should use Playwright's auto-waiting, web-first assertions, event waits, response waits, URL waits, or visible business-state checks.

Good locator strategy is also important. The test should prefer user-facing locators such as `getByRole()`, `getByLabel()`, and accessible names. If those are not stable, agreed `data-testid` values are better than brittle CSS classes, long XPath, DOM hierarchy, or index-based selectors.

Finally, the test should be maintainable in CI/CD. It should use isolated `BrowserContext` where appropriate, unique or controlled test data, safe storage-state usage, targeted cleanup, clear ownership, and useful artifacts for debugging.

## Common Mistake

Calling a Playwright test high quality only because it passes, without checking whether it protects real business value, uses stable locators, has meaningful assertions, isolates data, avoids hard waits, and provides useful failure evidence.

## Question 1.73

**How do you decide naming conventions for `data-testid` attributes?**

### Interview-Style Answer

I decide `data-testid` naming conventions based on stability, readability, business meaning, component context, and long-term maintainability.

A good `data-testid` should describe the purpose of the element, not its styling, position, or temporary implementation. Names should be independent of CSS classes, colors, layout, component library, or DOM structure. For example, `orders-filter-status`, `orders-submit-button`, `invoice-summary-total`, and `approval-comment-input` are more useful than `btn1`, `blue-button`, `right-panel-submit`, or `test123`.

The convention should be documented, reviewed in pull requests, and shared between frontend and automation teams so that test IDs remain consistent and stable during UI refactoring.

### Detailed Explanation

`data-testid` attributes are useful when user-facing locators such as role, label, accessible name, or stable text are not reliable enough. But if test IDs are named poorly, they can become just as confusing and fragile as bad CSS selectors.

I would define a naming convention that includes business area, component or element purpose, and action or field type where useful.

Good examples:

```
orders-filter-status
orders-submit-button
orders-grid
invoice-summary-total
approval-comment-input
user-profile-save-button
payment-method-card-number-input
```

Weak examples:

```
btn1
test123
blue-button
right-panel-submit
new-submit
abc-id
```

The weak names are problematic because they either do not explain the business purpose or they are tied to layout, color, or temporary implementation. If the button moves from the right panel to a modal, `right-panel-submit` becomes misleading. If the color changes from blue to green, `blue-button` becomes wrong.

A Playwright Java test should remain readable:

```
page.getByTestId("orders-filter-status").click();
page.getByTestId("orders-submit-button").click();

PlaywrightAssertions.assertThat(
    page.getByTestId("orders-grid")
).containsText("Pending Approval");
```

The naming convention should also avoid conflicts. If multiple modules have a submit button, `submit-button` may be too generic. A better name is `orders-submit-button`, `invoice-submit-button`, or `approval-submit-button`, depending on the business context.

For reusable components, I would define consistent patterns. For example, filter panels, grids, modals, tabs, and forms should follow the same structure across the application. This helps automation engineers predict test IDs and reduces unnecessary locator discussions.

Finally, test ID governance matters. Frontend developers and automation engineers should agree that test IDs are a stable testability contract. They should not be renamed casually during redesigns unless the business meaning changes. Pull request reviews should check whether new test IDs are meaningful, unique, stable, and consistent with the project convention.

## Common Mistake

Using inconsistent or implementation-based names such as `btn1`, `test-id-abc`, `blue-button`, or `right-panel-submit`, which makes tests harder to read and causes unnecessary locator maintenance after UI refactoring.

## Question 1.76

### How do you create a locator migration plan for an old Playwright suite?

#### Interview-Style Answer

I create a locator migration plan by prioritizing based on risk, not by trying to rewrite every locator at once.

I would first define the new locator standard, then audit the existing suite for brittle selectors such as absolute XPath, generated CSS classes, long DOM chains, `nth-child`, layout-based selectors, duplicated selectors, and non-unique locators. The migration should start with flaky tests, critical workflows, frequently changed screens, XPath-heavy files, duplicated locators, and high-maintenance modules.

The goal is not only to change selector syntax. The goal is to make locators reflect user intent, improve strictness, reduce maintenance, and keep the same business coverage.

#### Detailed Explanation

An old Playwright suite may contain many weak locators, but migrating everything in one large rewrite is risky and expensive. A better approach is incremental migration with clear standards, priority, and validation after each change.

First, I would define the preferred locator strategy:

1. `getByRole()` with accessible name
2. `getByLabel()`
3. `getByText()` where text is stable
4. `getByTestId()` for stable testability contracts
5. Stable CSS attributes as fallback
6. XPath only when there is no better option

Then I would audit the current suite and classify locator risk. High-risk locators include absolute XPath, generated classes, `nth-child`, index-based selectors, long CSS chains, and selectors copied across many files. I would also check for strict mode problems where a locator matches multiple elements and the test hides the issue using `nth()` instead of proper scoping.

Example migration item:

```
record LocatorMigrationItem(  
    String testName,  
    String module,  
    String risk,  
    String recommendation  
) {}  
  
LocatorMigrationItem item = new LocatorMigrationItem(  
    "approveInvoiceTest",  
    "Invoices",  
    "High",  
    "Replace XPath with row-scoped role locator"  
);  
  
Assertions.assertEquals("High", item.risk());
```

The highest priority should be tests that are flaky, business-critical, frequently changed, or expensive to maintain. Shared components should also be migrated early because improving a modal, grid, filter panel, or navigation component can fix many tests at once.

For example, instead of this:

```
page.locator("xpath=/html/body/div[2]/table/tbody/tr[3]/td[5]/button").click  
();
```

I would prefer a scoped locator:

```
Locator invoiceRow = page.getByRole(  
    AriaRole.ROW,  
    new Page.GetByRoleOptions().setName(Pattern.compile("INV-1001"))  
);  
  
invoiceRow.getByRole(  
    AriaRole.BUTTON,  
    new Locator.GetByRoleOptions().setName("Approve")  
).click();
```

Each migration should preserve the original business intent and improve readability. After each batch of locator changes, I would run the affected tests in CI, review failures, and update Page Objects or component objects where needed.

## Common Mistake

Trying to migrate every locator in one large rewrite, instead of prioritizing critical, flaky, duplicated, and high-maintenance locators and validating each migration incrementally through CI.

## Question 1.96

### How would you prove that Playwright automation improved release speed?

#### Interview-Style Answer

I would prove it with before-and-after release metrics, not just by saying automation helped. I would compare manual regression duration, smoke testing time, CI feedback time, defect triage speed, release sign-off effort, and release delays before and after Playwright automation was introduced.

The strongest proof is showing that critical feedback is now available earlier, failures come with useful artifacts, manual testers spend less time repeating regression checks, and release decisions happen faster with clearer evidence.

#### Detailed Explanation

Playwright automation improves release speed when it reduces repeated manual effort and gives faster confidence on critical workflows. To prove that, the team needs baseline data from before automation and trend data after automation.

Useful before-and-after comparison:

##### Before:

- Manual regression took 5 days.
- Smoke testing took 1 day.
- Defects were found late in release week.
- Release sign-off depended on many repeated manual checks.
- Failed scenarios needed long manual reproduction.
- Stakeholders waited longer for release confidence.

##### After:

- Automated smoke runs on every important build.
- Critical Playwright regression runs overnight or on schedule.
- Failures include trace, screenshots, videos, console logs, and network evidence.
- Manual testing focuses more on exploratory and risk-based testing.
- Defects are found earlier in the pipeline.
- Release sign-off uses automation evidence and risk classification.

I would track measurable indicators such as:



- 
1. Regression cycle time
  2. Smoke feedback time
  3. Time from build completion to release recommendation
  4. Manual testing hours saved
  5. Defects found before release
  6. Defects escaped to production
  7. Average failure triage time
  8. Number of release delays caused by late regression defects
  9. Flaky-test rate
  10. Percentage of critical workflows covered by Playwright

For example:

Before Playwright:

- Regression duration: 5 days
- Smoke testing: 1 day
- Average failure triage: 2 hours
- Release delays from late defects: 4 per quarter

After Playwright:

- Critical regression: overnight
- Smoke feedback: 20 minutes
- Average failure triage: 30 minutes with Trace Viewer evidence
- Release delays from late defects: 1 per quarter

This shows automation impact in business terms: faster feedback, earlier defect discovery, reduced manual effort, and quicker release decisions. I would also review these metrics regularly so the framework continues to support release speed as the product and test suite grow.

## Common Mistake

claiming Playwright improved release speed without baseline data, trend metrics, failure-classification evidence, or a clear link between automation results and faster release decisions.

---

## Question 1.108

### How do you make tests parallel-safe in Playwright Java?

#### Interview-Style Answer

I would make Playwright Java tests parallel-safe by isolating browser state, test data, users, files, route mocks, and artifacts. Each test should use its own `BrowserContext` and `Page`, independent data, and unique output paths for downloads, screenshots, traces, and videos.

Parallel safety is not achieved only by enabling TestNG, JUnit, Maven, or CI parallel execution. The framework must avoid shared mutable state such as static `Page`, static `BrowserContext`, shared users, shared orders, shared download folders, and unsafe cleanup logic.

#### Detailed Explanation

Parallel failures usually happen because tests share hidden state. Two tests may run in different browser contexts but still interfere with each other through backend data, users, storage-state files, route mocks, downloads, or static variables.

A practical parallel-safety checklist:

1. Create a fresh `BrowserContext` per test.
2. Create a fresh `Page` per test.
3. Do not use static `Page` or static `BrowserContext`.
4. Avoid static mutable `Page` Objects, locators, users, tokens, or test data.
5. Use unique users or unique records where workflows can conflict.
6. Use test-specific API setup and cleanup.
7. Use unique download, upload, screenshot, trace, and video paths.
8. Keep route mocks scoped to the test context or page.
9. Avoid dependency on test execution order.
10. Ensure cleanup deletes only data created by the current test.

Example:

```
String uniqueOrderId = "ORD-" + UUID.randomUUID();

BrowserContext context = browser.newContext();
Page page = context.newPage();

Path downloadDir = Paths.get("target", "downloads", uniqueOrderId);
Files.createDirectories(downloadDir);
```

For test data, the framework should generate records that clearly belong to the current test:

---

```
String email = "buyer-" + UUID.randomUUID() + "@example.com";
String orderName = "automation-order-" + UUID.randomUUID();

String orderId = testDataApi.createOrder(orderName, email);
```

Cleanup should be targeted:

```
testDataApi.deleteOrder(orderId);
```

This is safer than deleting broad data such as all draft orders or all automation users, because another parallel test may still be using them.

If storage state is used, it should be role-specific and safe for parallel execution. Tests should not write back to the same storage-state file during execution. If route mocking is used with `page.route()` or `browserContext.route()`, the mock should be registered only for the test that needs it and should not accidentally affect unrelated tests.

## Common Mistake

using the same user, same order ID, same download folder, same storage-state file, unsafe cleanup rule, or static `Page` across parallel tests, then blaming Playwright for random instability.

---

## Question 1.119

### How would you recover automation credibility after automation missed a production defect?

#### Interview-Style Answer

I would recover automation credibility by treating the missed production defect as a learning event, not as a blame exercise.

I would analyze whether the defect was missed because the scenario was not automated, the assertion was weak, the test data was unrealistic, the issue was hidden by mocks, the wrong browser or role was not covered, or the test was skipped, flaky, or placed at the wrong test layer. Then I would add the right preventive coverage and improve standards so similar defects are less likely to escape again.

#### Detailed Explanation

When automation misses a production defect, the immediate response should be transparent and evidence-based. The team should not defend automation blindly or simply add one test and move on. The real goal is to understand why the automation signal failed.

A practical review should ask:

1. Was the affected scenario automated?
2. If automated, was the assertion strong enough?
3. Did the test validate the real user-visible outcome?
4. Did the test use realistic test data?
5. Did mocks, stubs, or `route.fulfill()` hide the real backend behavior?
6. Was the affected browser, device, role, permission, or environment covered?
7. Was the test skipped, quarantined, flaky, or ignored due to retry-pass behavior?
8. Was the check better suited for UI, API, contract, component, or unit testing?
9. What preventive test or monitoring is needed?
10. What review rule, framework standard, or CI check should improve?

For example, if a production defect happened because the UI showed Payment Successful even when the backend payment status was actually failed, adding only one Playwright test may not be enough. The team may need API-level validation, stronger UI assertions, realistic payment test data, better `waitForResponse()` usage, and a visible final-state check in the UI.

A preventive Playwright test should validate the business outcome, not only the action:

---

```
paymentsPage.submitPayment(orderId);

PlaywrightAssertions.assertThat(
    paymentsPage.paymentStatusForOrder(orderId)
).hasText("Payment Failed");
```

If the issue was caused by over-mocking, I would review where `page.route()`, `route.fulfill()`, or test doubles are used. Mocking is useful, but critical release workflows should also have some coverage with real backend behavior so integration defects are not hidden.

To rebuild trust, I would communicate clearly: what escaped, why automation missed it, what coverage is being added, what review rule is changing, who owns the fix, and how the team will measure prevention. This shows maturity and restores confidence in automation as a quality system.

## Common Mistake

adding one Playwright test for the escaped production bug, but not improving the weak assertion, unrealistic data, over-mocking, missing browser/role coverage, flaky-test handling, or review process that allowed the defect to escape.

---

## Question 1.151

**How would you prioritize fixing flaky tests when there are too many to fix immediately?**

### Interview-Style Answer

I would prioritize flaky tests by business risk, failure frequency, CI impact, release-blocking behavior, ease of diagnosis, ownership, and whether the failure may hide a real product defect.

I would first fix flaky tests covering critical workflows such as login, payment, checkout, permissions, order approval, or production-impacting paths. Then I would handle tests that fail frequently in CI, block pull requests, consume debugging time, or reduce stakeholder trust. Low-value, duplicate, or outdated flaky tests should be removed, rewritten, quarantined, or moved out of the PR pipeline instead of consuming the same priority as critical tests.

### Detailed Explanation

When there are many flaky tests, fixing them randomly wastes effort and does not improve release confidence quickly. The goal is to reduce the highest business and engineering risk first.

A useful prioritization order is:

1. Flaky tests covering critical business workflows.
2. Flaky tests that frequently fail in PR or release pipelines.
3. Flaky tests that block deployments or require repeated reruns.
4. Flaky tests that may be exposing real product instability.
5. Flaky tests with clear known causes and quick fixes.
6. Flaky tests with no duplicate coverage.
7. Low-value or duplicate flaky tests that should be deleted, rewritten, or moved out of PR checks.

Before fixing, I would classify the failure using evidence from Trace Viewer, screenshots, videos, console logs, network logs, CI logs, and retry history. The team should identify whether the issue is caused by weak locators, missing assertions, actionability problems, shared test data, unstable environment, backend/API timing, authentication/session issues, file/download conflicts, or actual application behavior.

Example prioritization:

---

Payment approval flaky test:

- Critical workflow
- High failure frequency
- Blocks release pipeline
- No duplicate coverage
- Fix immediately

Footer alignment visual test:

- Low business risk
- Duplicated by component visual coverage
- Not release critical
- Move out of PR pipeline or remove if not valuable

For CI stability, I would also separate flaky tests into clear categories:

Fix now:

- Critical and frequently failing tests

Investigate:

- Possible product defects or environment instability

Quarantine temporarily:

- Valuable tests that are unstable but not immediately fixable

Delete or rewrite:

- Duplicate, outdated, weak, or low-value tests

Quarantine should be temporary and tracked with an owner, reason, and target fix date. Otherwise, it becomes a place where bad tests are forgotten. The team should also monitor flaky-test percentage, retry rate, failure frequency, time lost in reruns, and escaped defects to make sure the flakiness backlog is reducing over time.

## Common Mistake

fixing flaky tests in the order they appear in the report, instead of prioritizing by business risk, CI impact, failure frequency, product-defect possibility, and long-term automation value.

---

## **2. Test Strategy**



---

## **Part I - Core Questions**

## Question 2.6

### Why is API testing useful in a Playwright Java automation framework?

#### Interview-Style Answer

API testing is useful because it validates backend behavior faster, more directly, and with less UI dependency than browser-based tests. It is ideal for checking business rules, permissions, request/response contracts, status codes, error handling, and test data setup or cleanup.

In a Playwright Java framework, API testing should complement UI testing. API tests prove backend correctness, while UI tests prove that the user can complete real browser workflows and see the expected result on the page.

#### Detailed Explanation

A good Playwright Java automation framework should not force every validation through the UI. If the requirement is mainly backend behavior, validating it through an API call is usually faster, more stable, and easier to diagnose than navigating through multiple screens.

API tests are useful for:

1. Status-code validation
2. Request and response body validation
3. Business-rule validation
4. Permission and role validation
5. Contract validation
6. Negative and error scenarios
7. Fast regression checks
8. Test data creation
9. Test data cleanup
10. Precondition setup before UI tests

For example, creating a customer through an API before validating it in the UI is often better than creating the same customer repeatedly through several UI screens.

```
APIRequestContext request = playwright.request().newContext(  
    new APIRequest.NewContextOptions()  
        .setBaseURL("https://example.com")  
);  
  
APIResponse response = request.post("/api/customers",  
    RequestOptions.create()  
        .setData(Map.of(  
            "name", "Test Customer",  
            "status", "ACTIVE"  
        ))  
);  
  
Assertions.assertEquals(201, response.status());
```

Then the UI test can focus on the user-visible behavior:

```
page.navigate("https://example.com/customers");  
  
PlaywrightAssertions.assertThat(  
    page.getByText("Test Customer")  
).isVisible();
```

UI tests are still important because APIs cannot prove browser behavior. UI tests validate whether the user can log in, navigate, fill forms, upload files, download reports, see validation messages, interact with dialogs, and complete workflows in the browser.

UI tests are useful for:

1. End-user workflows
2. Browser rendering
3. Form behavior
4. Navigation
5. User-visible validation
6. Accessibility-oriented locator behavior
7. File upload and download behavior
8. Dialog, popup, and iframe interactions

A strong framework uses both layers wisely. Backend rules should be covered mostly through API tests, while fewer high-value UI tests should validate the complete user journey and visible outcome.

## Common Mistake

testing every backend rule through a slow UI journey. This makes the suite slower, more brittle, and harder to debug than using API tests for backend logic and UI tests for user-facing browser behavior.

---

## Question 2.8

**How do you decide whether a scenario should be tested through UI, API, or both?**

### Interview-Style Answer

I decide the test layer based on the risk being validated. UI tests are best when the risk is user experience, browser interaction, navigation, rendering, accessibility, or end-to-end workflow. API tests are better when the risk is backend logic, data rules, permissions, contracts, schema, or large input combinations.

I would use both UI and API when the scenario is business-critical and the integration between frontend and backend is important. In that case, API tests can cover many backend combinations quickly, while UI tests validate that the user can complete the important journey and see the correct visible outcome.

### Detailed Explanation

Not every scenario should be automated through the UI. UI tests provide high confidence because they use the application like a real user, but they are slower, more expensive to maintain, and more sensitive to browser state, test data, locators, timing, and environment issues. API tests are usually faster and more stable for validating backend behavior, but they do not prove that the user can complete the workflow in the browser.

A practical decision guide is:

Use UI when:

- The real user journey matters.
- Browser interaction is the risk.
- Navigation, redirects, dialogs, popups, downloads, or uploads matter.
- Rendering or layout behavior matters.
- Accessibility behavior matters, such as role locators, accessible names, keyboard flow, focus, or ARIA state.
- The scenario is a critical end-to-end business path.

Use API when:

- Backend validation is the main risk.
- Business rules need many input combinations.
- Permissions, status transitions, or data processing must be verified.
- Request/response contracts or schema need validation.
- Setup and cleanup can be done faster through backend endpoints.
- The UI does not add meaningful confidence.

Use both when:

- The flow is business-critical.
- Frontend-backend integration is high risk.
- API response must produce the correct visible UI result.
- The backend rule and the browser workflow both need confidence.

For example, login can have API tests for invalid credentials, locked users, expired passwords, and permission rules. But at least one UI test should still verify that a valid user can sign in through the browser and land on the correct dashboard.

For an order workflow, API tests can validate pricing rules, tax calculation, coupon validation, and order status transitions. A smaller number of UI tests can validate that the user can add an item to the cart, apply a coupon, place the order, and see the order confirmation.

When API behavior is used inside a UI test, the final validation should still be user-visible:

```
Response response = page.waitForResponse(
    res -> res.url().contains("/api/orders")
    && res.status() == 201,
    () -> {
        page.getByRole(
            AriaRole.BUTTON,
            new Page.GetByRoleOptions().setName("Place Order")
        ).click();
    }
);

PlaywrightAssertions.assertThat(
    page.getByText("Order placed successfully")
).isVisible();
```

Here, the network event helps prove the backend call happened, but the final assertion proves the user-visible business result.

## Common Mistake

---

testing every backend rule through slow UI flows, which creates a large, flaky, and expensive regression suite when most rule combinations could be tested faster and more clearly at the API layer.

---

---

## Question 2.11

**How do you decide whether a generated test should become a UI test, API test, component-level test, or be discarded?**

### Interview-Style Answer

I would decide based on the risk the generated test is trying to validate. If the risk is a real user journey, browser behavior, navigation, rendering, accessibility, or cross-page workflow, it can become a UI test. If the risk is mainly a backend rule, validation, permission, or contract, it should usually become an API test.

If the risk belongs to a reusable UI component, such as a dropdown, modal, grid, date picker, or upload widget, I would move it to a component-level test. If the generated test duplicates existing coverage, checks implementation noise, or does not validate a meaningful user or business outcome, I would discard it.

### Detailed Explanation

Generated tests, especially from tools like Codegen, are useful starting points, but they should not be accepted directly into the regression suite. A recorded flow often captures every click, selector, and intermediate step, but it does not automatically understand test value, risk, duplication, maintainability, or the right automation layer.

A good decision guide is:

---

Keep as a UI test when:

- The complete browser workflow matters.
- User interaction is the main risk.
- Navigation between pages must be validated.
- Rendering, layout, dialogs, downloads, uploads, or popups matter.
- Accessibility behavior such as role, accessible name, keyboard flow, or focus matters.
- End-to-end confidence is required for a critical business journey.

Move to API test when:

- The main risk is backend validation.
- The test checks permissions, rules, status changes, or data processing.
- Many data combinations must be tested.
- The UI adds little or no additional confidence.
- Faster and more stable validation is possible through API.

Move to component-level test when:

- A reusable component has many states.
- The same behavior appears across many pages.
- Full application navigation is unnecessary.
- The risk is inside a modal, dropdown, table, date picker, wizard step, or form component.

Discard when:

- It duplicates existing UI, API, or component coverage.
- It validates no meaningful business behavior.
- It only checks temporary DOM structure or implementation details.
- It is too brittle compared with the value it provides.
- It does not have a clear assertion or pass/fail purpose.

For example, a generated test that logs in, opens the cart, applies a coupon, completes checkout, and verifies the order confirmation may deserve to become a UI test because the end-to-end user journey is the risk.

But testing 20 coupon validation rules through the full checkout UI is usually not ideal. Those combinations are better tested at API level, while the UI suite keeps one or two high-value coupon scenarios.

Similarly, if Codegen records repeated interactions with a custom date picker across many pages, the better design may be to create a component-level test for the date picker and keep only one or two end-to-end flows that prove integration with the application.

## Common Mistake

converting every generated or Codegen-recorded flow into a full UI regression test without checking whether the risk belongs at UI, API, component level, or should be removed as duplicate low-value coverage.



---

## **Book 5 - Advanced Playwright Topics**

---

## 1. Selenium Grid

---

## **Part I - Core Questions**

## Question 1.5

### What are the limitations of using Selenium Grid with Playwright compared to native Playwright execution?

#### Interview-Style Answer

Using Selenium Grid with Playwright has several limitations compared to native Playwright execution because it introduces an external infrastructure layer (Selenium Grid) and relies on CDP-based attachment instead of Playwright's native browser control model.

Native Playwright execution is simpler, faster, and fully integrated with Playwright's architecture, while Selenium Grid adds complexity and reduces some of Playwright's built-in advantages.

#### Detailed Explanation

Native Playwright execution:

```
Browser browser = playwright.chromium().launch();
Page page = browser.newPage();
```

This model is fully managed by Playwright:

- Browser lifecycle
- Context isolation
- Network handling
- Auto-waiting integration
- Tracing, screenshots, video

Selenium Grid execution introduces an external dependency:

```
SELENIUM_REMOTE_URL=http://<selenium-hub-ip>:4444 mvn test
```

Here Playwright must rely on Selenium Grid to:

- Start the browser session
- Expose CDP endpoint
- Manage remote execution

Key limitations of Selenium Grid + Playwright

- 
1. Experimental support in Playwright ecosystem
  2. Dependency on Selenium Grid infrastructure stability
  3. Only Chromium-based browsers are realistically supported
  4. Additional network latency due to remote execution
  5. More complex debugging across multiple layers
  6. Harder root cause analysis (Playwright vs Grid vs Node vs CDP)
  7. Version compatibility challenges (Grid, browser, CDP, Playwright)
  8. Not aligned with Playwright's native execution model
  9. Reduced simplicity compared to local/Playwright-managed execution
  10. Operational overhead (Grid setup, scaling, maintenance)

Debugging complexity difference

Native Playwright:

Test → Playwright → Browser

Selenium Grid + Playwright:

Test → Playwright → Selenium Grid Hub → Node → Browser → CDP → Back to Playwright

Each extra layer increases failure points:

- Grid routing issues
- Node registration issues
- CDP connection failures
- Network/firewall issues
- Container/Docker misconfigurations

When native Playwright is better

Native execution is preferred for:

- CI/CD pipelines
- Parallel execution (workers/sharding)
- Cross-browser testing (Chromium, Firefox, WebKit)
- Trace Viewer + debugging workflows
- Stable automation frameworks

Common Mistake

A common mistake is assuming Selenium Grid integration provides the same performance, simplicity, and reliability as native Playwright execution. In reality, it introduces additional infrastructure dependencies and complexity, and should only be used when there is a strong requirement to reuse existing Selenium Grid infrastructure.

## Question 1.8

### How would you design a Playwright Java framework to optionally run tests locally or on Selenium Grid?

#### Interview-Style Answer

I would design the framework so that local execution is the default mode, and Selenium Grid execution is enabled through external configuration.

The test code should remain completely unchanged. The framework layer should decide whether to run locally or on Selenium Grid based on environment variables or configuration properties such as `RUN_MODE` and `SELENIUM_REMOTE_URL`.

#### Detailed Explanation

A clean Playwright framework must separate test logic from execution strategy.

Configuration example:

```
RUN_MODE=local
RUN_MODE=grid
SELENIUM_REMOTE_URL=http://localhost:4444
BROWSER=chromium
```

Framework decision flow

1. Read execution mode from configuration
2. If `RUN_MODE = local` → launch browser locally
3. If `RUN_MODE = grid` → validate `SELENIUM_REMOTE_URL`
4. Ensure Grid supports required browser (Chrome/Edge only)
5. Create browser using Playwright API consistently
6. Keep all test classes independent of execution mode

Example framework bootstrap logic

```
String runMode = System.getenv().getOrDefault("RUN_MODE", "local");

if ("grid".equalsIgnoreCase(runMode)) {
    String gridUrl = System.getenv("SELENIUM_REMOTE_URL");

    if (gridUrl == null || gridUrl.isBlank()) {
        throw new RuntimeException("SELENIUM_REMOTE_URL is required for Grid execution");
    }
}

Browser browser = playwright.chromium().launch();
Page page = browser.newPage();
```

---

When Grid is enabled, the underlying execution is routed through Selenium Grid automatically, while the test code remains unchanged.

Test code (must stay identical for both modes)

```
page.navigate("https://example.com");  
page.getByRole(AriaRole.BUTTON).click();
```

### Recommended framework design practices

1. Keep execution configuration outside test classes
2. Centralize browser creation in a factory layer
3. Fail fast if Grid configuration is missing or invalid
4. Log execution mode at startup (local vs grid)
5. Keep local mode as default for developer productivity
6. Restrict Grid usage to Chrome/Edge compatibility
7. Capture trace, screenshots, and videos uniformly for both modes
8. Document Selenium Grid support as experimental in framework README

### Common Mistake

A common mistake is introducing separate test logic or branching inside test cases for local vs Grid execution. A well-designed framework should only change the execution layer, not the test behavior, ensuring consistency across environments.

---

---

## **2. Snapshot testing**



---

## **Part I - Core Questions**

## Question 2.3

### When is snapshot testing not suitable?

#### Interview-Style Answer

Snapshot testing is not suitable when the output is highly dynamic, changes frequently, or contains values that are expected to vary between test runs.

Examples include timestamps, random IDs, notification counts, live prices, search results, personalized recommendations, and rapidly changing dashboard data. In these situations, snapshots often produce noisy failures that do not represent real regressions.

For such scenarios, fine-grained assertions, partial matching, or regular expressions are usually better choices.

#### Detailed Explanation

Snapshot testing works best when the captured structure is stable and meaningful over time.

Example:

```
PlaywrightAssertions.assertThat(page.locator("main"))
    .matchesAriaSnapshot("""
      - heading "Checkout"
      - textbox "Name"
      - textbox "Email"
      - button "Pay now"
    """);
```

This is a good candidate because the checkout page structure is expected to remain relatively stable.

However, snapshot testing becomes less effective when the content naturally changes between executions.

Common examples include:

- timestamps and dates
- random IDs
- order numbers
- session identifiers
- notification counts
- live prices
- stock values
- search results
- personalized recommendations
- frequently changing dashboards

For example:

```
PlaywrightAssertions.assertThat(page.locator("main"))
    .matchesAriaSnapshot("""
        - heading "Dashboard"
        - text: Last updated at 10:45:21
        - text: Notifications 17
        - text: Recommended job: Java Automation Engineer
    """);
```

This snapshot is likely to fail often even when the application is working correctly because the values are expected to change.

In such cases, targeted assertions provide clearer validation:

```
PlaywrightAssertions.assertThat(
    page.getByRole(
        AriaRole.HEADING,
        new Page.GetByRoleOptions().setName("Dashboard")
    )
).isVisible();

PlaywrightAssertions.assertThat(
    page.getByText("Notifications")
).isVisible();
```

Or use flexible matching:

```
PlaywrightAssertions.assertThat(page.locator("main"))
    .matchesAriaSnapshot("""
        - heading "Dashboard"
        - text: /Notifications \\d+/
    """);
```

Snapshot testing may also be a poor choice when:

- snapshots become very large
- failures are difficult to review
- baseline updates occur frequently
- the output changes more often than the feature itself
- business logic requires precise validation

For critical workflows, explicit assertions are usually better because they communicate intent clearly.

Example:

```
PlaywrightAssertions.assertThat(
    page.getByText("Payment successful")
).isVisible();

PlaywrightAssertions.assertThat(page)
    .hasURL(Pattern.compile(".*orders/confirmation"));
```

These assertions clearly describe the expected business outcome and are easier to diagnose when they fail.

---

A practical rule is:

Use snapshot testing for:

- stable accessibility structure
- reusable UI components
- visual regression baselines
- broad regression coverage

Use assertions for:

- business rules
- exact values
- dynamic content
- user-visible outcomes
- workflow validation

## Common Mistake

using snapshots for highly dynamic pages and repeatedly updating baselines whenever they fail. If snapshot updates become routine maintenance rather than meaningful review, the test is no longer providing useful regression protection. Use focused assertions, partial matching, or regex-based matching instead.

---

## Question 2.12

### How can omitting accessible names or attributes make ARIA snapshot tests more flexible?

#### Interview-Style Answer

Omitting accessible names or attributes in an ARIA snapshot makes the test less strict. It allows the test to verify only the accessibility information that matters, such as role, hierarchy, or presence, without enforcing every label or state.

For example:

```
- button
```

checks that a button exists, but does not require a specific accessible name such as [Submit](#), [Save](#), or [Continue](#).

Similarly:

```
- checkbox
```

checks that a checkbox exists, but does not require it to be checked or unchecked.

This is useful when names, values, or ARIA states are dynamic and not important to the test objective.

#### Detailed Explanation

ARIA snapshots can be written with very specific expectations.

Example:

```
- button "Submit" [disabled]
```

This means the accessibility tree must contain:

```
- role: button  
- accessible name: Submit  
- state: disabled
```

This is useful when the exact accessible name and state are part of the requirement.

But if the test only cares that a button exists, the accessible name can be omitted:

---

```
- button
```

Example:

```
<button>Submit</button>
```

Flexible ARIA snapshot:

```
PlaywrightAssertions.assertThat(page.locator("body"))
    .matchesAriaSnapshot(""  
    - button  
    """);
```

This focuses on the role. If the button text changes from **Submit** to **Continue**, the snapshot can still pass because the accessible name was not part of the expectation.

The same applies to attributes.

Specific snapshot:

```
- checkbox [checked]
```

Flexible snapshot:

```
- checkbox
```

By omitting **[checked]**, the test checks only that a checkbox is present. It does not care whether the checkbox is currently checked, unchecked, enabled, or disabled.

This is useful when:

```
- accessible names contain dynamic values  
- labels change based on user data or locale  
- checkbox or toggle states are not relevant  
- optional ARIA attributes vary by state  
- the test only validates role and structure  
- strict matching would create noisy failures
```

However, omission should be intentional. If the requirement is that a button must be named **Pay now**, then the name should be included:

```
- button "Pay now"
```

If the requirement is that a checkbox must be checked, then the state should be included:

```
- checkbox "Accept terms" [checked]
```

---

So, omitting names or attributes improves flexibility only when those details are not important to the scenario.

### **Common Mistake**

omitting important accessibility details just to make the test pass. If the accessible name, state, or attribute is part of the requirement, keep it in the snapshot; omit only dynamic or irrelevant details to avoid brittle tests.

---

---

## **3. Accessibility Testing**



---

## **Part I - Core Questions**

### Question 3.8

## How do you integrate axe accessibility checks with Playwright Java?

### Interview-Style Answer

To integrate axe accessibility checks with Playwright Java, I would add the `axe-core` JavaScript file as a test resource, navigate the application to the required page state, inject axe into the page, run `axe.run()` using `page.evaluate()`, collect violations, and fail or warn based on the team's agreed accessibility policy.

This is useful because axe can automatically detect many rule-based accessibility issues such as missing labels, invalid ARIA attributes, missing alt text, contrast problems, landmark issues, and form accessibility problems. However, axe should be treated as one layer of accessibility validation, not a complete replacement for keyboard testing, screen-reader testing, ARIA snapshot testing, and manual review.

### Detailed Explanation

In a Playwright Java framework, axe is usually integrated by loading the `axe-core` script into the browser page and then executing `axe.run()` after the page reaches the state that needs to be tested.

A typical integration flow is:

1. Add `axe-core` JavaScript as a local test resource.
2. Navigate to the page or component state.
3. Wait until the important UI content is visible.
4. Inject axe into the page.
5. Run `axe.run()` using `page.evaluate()`.
6. Collect accessibility violations.
7. Filter or classify violations based on severity, tags, or agreed rules.
8. Fail the test or attach violations to the report based on project policy.

Example idea:

```

import com.microsoft.playwright.*;
import com.microsoft.playwright.options.AriaRole;

import java.nio.file.Files;
import java.nio.file.Paths;

import static
    com.microsoft.playwright.assertions.PlaywrightAssertions.assertThat;

public class AccessibilityTest {
    public static void main(String[] args) throws Exception {
        try (Playwright playwright = Playwright.create()) {
            Browser browser = playwright.chromium().launch();
            Page page = browser.newPage();

            page.navigate("https://example.com/login");

            assertThat(page.getByRole(
                AriaRole.HEADING,
                new Page.GetByRoleOptions().setName("Login")
            )).isVisible();

            String axeScript =
                Files.readString(Paths.get("src/test/resources/axe.min.js"))
                ;
            page.addScriptTag(new
                Page.AddScriptTagOptions().setContent(axeScript));

            Object results = page.evaluate("async () => await
                axe.run(document)");

            System.out.println(results);

            browser.close();
        }
    }
}

```

In a real framework, the **results** object should be parsed properly and converted into a readable report. The framework can fail the test when violations match agreed conditions, such as serious or critical impact violations.

For example, the team may define a policy like:

- Fail the test for critical and serious violations.
- Warn for moderate violations during early adoption.
- Ignore only approved known issues with ticket references.
- Run checks on stable page states, not during loading.
- Attach violations to CI reports.

This policy is important because blindly failing on every issue without ownership can create noise, while ignoring all violations makes the check useless. The team should define scope, severity rules, allowed exclusions, reporting format, and ownership.

Axe checks are useful for detecting technical accessibility problems, but they do not prove the full user experience. For example, axe may

---

detect a missing label, but it cannot fully judge whether the screen-reader flow is natural, whether alt text is meaningful, whether the instructions are clear, or whether the page is easy to use for users with cognitive disabilities.

### **Common Mistake**

assuming axe alone validates complete accessibility. Axe is valuable for automated rule-based checks, but it should be combined with role-based locators, ARIA snapshot testing, keyboard navigation checks, screen-reader review, and manual accessibility validation for critical flows.

---

### Question 3.9

## What are the limitations of automated accessibility testing?

### Interview-Style Answer

Automated accessibility testing can catch many technical accessibility issues, but it cannot fully prove that a page is accessible for real users. Tools can detect problems such as missing labels, invalid ARIA attributes, broken roles, missing alt text, and some contrast issues, but they cannot fully judge screen-reader experience, content clarity, logical reading order, cognitive accessibility, or whether the UI is actually easy to use.

In Playwright Java, automated checks, ARIA assertions, keyboard checks, and accessibility scans are useful as regression safety nets. However, they should be combined with manual review, keyboard-only testing, screen-reader validation, and real user experience checks for critical flows.

### Detailed Explanation

Automated accessibility testing is valuable because it can quickly detect many common accessibility problems during development and CI/CD execution. It helps prevent obvious issues from reaching production and gives teams fast feedback after UI changes.

Automated checks can help identify:

- Missing form labels
- Invalid ARIA attributes
- Incorrect or missing roles
- Missing alt text
- Some color contrast issues
- Some keyboard navigation problems
- Duplicate IDs
- Broken heading structure
- Missing accessible names
- Accessibility tree regressions

For example, a Playwright test can verify that an important button is exposed with a proper role and accessible name:

```
PlaywrightAssertions.assertThat(
    page.getByRole(
        AriaRole.BUTTON,
        new Page.GetByRoleOptions().setName("Submit")
    )
).isVisible();
```

---

ARIA snapshot testing can also help detect accessibility structure changes:

```
PlaywrightAssertions.assertThat(page.locator("body")).matchesAriaSnapshot("""
- heading "Login" [level=1]
- textbox "Email"
- textbox "Password"
- button "Sign in"
""");
```

But automated testing has limits because accessibility is not only about technical rules. A page can pass automated checks and still be difficult for users.

Manual review is still needed for:

- Whether screen-reader navigation feels natural
- Whether alt text is meaningful, not just present
- Whether instructions are clear
- Whether error messages are understandable
- Whether focus order matches the user journey
- Whether keyboard navigation is practical
- Whether headings create a logical content structure
- Whether the page is usable for users with cognitive disabilities
- Whether the design intent is accessible in real usage

For example, an image may have alt text, so an automated tool may not report a missing-alt violation. But if the alt text says only "[image](#)" or "[banner](#)", it is technically present but not meaningful. Similarly, a button may have an accessible name, but the complete screen-reader flow may still be confusing if the surrounding context is unclear.

Automated accessibility testing should therefore be treated as one layer of quality, not the complete solution. A strong accessibility strategy combines Playwright checks, ARIA snapshot testing, automated accessibility scans, keyboard testing, screen-reader testing, design review, and manual validation for important workflows.

## Common Mistake

claiming a page is accessible because automated scans show zero violations. Zero automated violations only means the tool did not detect rule-based issues; it does not prove that the page is fully usable, understandable, and accessible for real users.

---

---

## **4. Cross browser testing**

---

## **Part I - Core Questions**



## Question 4.2

### Should every Playwright Java test run on every browser?

#### Interview-Style Answer

No. Every Playwright Java test does not need to run on every browser. I would run critical workflows across all supported browsers and keep the full regression suite on the primary supported browser unless the product risk justifies wider coverage.

A good cross-browser strategy should be based on the product's browser support matrix, customer usage, business criticality, past browser-specific defects, execution cost, and CI capacity.

#### Detailed Explanation

Running every test on Chromium, Firefox, and WebKit may increase execution time and CI cost without giving proportional value. Cross-browser testing is valuable, but it should be applied strategically.

A practical strategy could be:

Chromium:

- Full regression suite
- Main development feedback
- Broad functional coverage

Firefox:

- Smoke tests
- Critical workflows
- Browser-sensitive flows
- Areas with past Firefox defects

WebKit:

- Smoke tests
- Safari-sensitive workflows
- Responsive and mobile-like flows
- Critical forms, checkout, upload, download, and layout checks

Good candidates for cross-browser execution include:

1. Login and logout.
2. Registration or onboarding.
3. Checkout, payment, or order submission.
4. File upload and download.
5. Complex forms.
6. Date, time, select, number, and file inputs.
7. Responsive layouts.
8. Keyboard and focus-sensitive flows.
9. Dashboard or reporting pages.
10. High-value business workflows.

---

The framework should separate browser setup from test logic. For example, the same test should run with different browser values through Maven, Gradle, JUnit, TestNG, or CI matrix jobs.

Example:

```
mvn test -Dbrowser=chromium
mvn test -Dbrowser=firefox
mvn test -Dbrowser=webkit
```

Each browser run should use isolated `BrowserContext` instances, browser-specific reports, separate screenshots, traces, videos, downloads, and clear cleanup. This makes it easier to compare whether a failure is caused by browser behavior, test data, environment, or automation design.

The strategy should also distinguish project setup, browser-binary setup, and runtime behavior. Browser binaries must be installed and available in CI, but that does not mean every test must execute against every browser.

## Common Mistake

Using an extreme strategy: either running no cross-browser tests and missing real browser defects, or running the entire suite on every browser without considering business risk, execution time, CI cost, and browser support priorities.

---

## Question 4.8

### How can locator behavior differ across browsers in Playwright Java tests?

#### Interview-Style Answer

Locator behavior can appear different across browsers when the underlying page is exposed differently by Chromium, Firefox, or WebKit. Playwright's locator API is consistent, but browser engines may differ in how they calculate visibility, accessible roles, accessible names, text rendering, focus behavior, shadow DOM behavior, or invalid HTML correction.

In Playwright Java, I would troubleshoot this by checking locator count, strict mode failures, visibility, accessible role and name, trace snapshots, screenshots, and whether the application markup is valid, semantic, and accessible across browsers.

#### Detailed Explanation

Role-based and accessibility-oriented locators are powerful, but they depend on the browser's accessibility tree and the application's HTML semantics. If the markup is weak, invalid, or inaccessible, different browsers may expose the same element slightly differently.

Examples of browser-sensitive locator issues include:

1. Icon-only buttons without accessible names.
2. Labels not properly associated with inputs.
3. Hidden text contributing differently to accessible names.
4. CSS-generated content affecting visible text expectations.
5. Elements hidden in one browser due to CSS support differences.
6. SVG buttons missing aria-label.
7. Placeholder text being used incorrectly as a label.
8. Invalid HTML being corrected differently by browser engines.
9. Text wrapping or font rendering changing visible text.
10. Duplicate accessible names causing strict mode failures.

Example:

```
Locator submitButton = page.getByRole(
    AriaRole.BUTTON,
    new Page.GetByRoleOptions().setName("Submit")
);

System.out.println("Submit button count: " + submitButton.count());

PlaywrightAssertions.assertThat(submitButton).isVisible();

submitButton.click();
```

---

If this passes in Chromium but fails in Firefox or WebKit, I would not immediately replace it with XPath or CSS. First, I would check whether the button has a proper semantic role and accessible name in all browsers. For example, an icon-only button should expose a meaningful name:

```
<button aria-label="Submit form">
  <svg></svg>
</button>
```

Then the test can use a stable role locator:

```
page.getByRole(
  AriaRole.BUTTON,
  new Page.GetByRoleOptions().setName("Submit form")
).click();
```

For repeated elements, the issue may not be cross-browser behavior but missing scoping. In that case, the locator should be scoped to a row, dialog, card, or section:

```
Locator row = page.getByRole(
  AriaRole.ROW,
  new Page.GetByRoleOptions().setName(Pattern.compile("ORD-101"))
);

row.getByRole(
  AriaRole.BUTTON,
  new Locator.GetByRoleOptions().setName("View")
).click();
```

A senior troubleshooting approach is to compare traces and screenshots across browsers, check whether the locator resolves to the same element, and decide whether the fix belongs in application accessibility, locator scoping, or test synchronization.

## Common Mistake

Blaming Playwright locator reliability when the real issue is invalid HTML, missing labels, duplicate accessible names, poor semantics, browser-specific visibility, or inaccessible UI that browsers expose differently.

---

---

## **5. Visual testing**

---

## **Part I - Core Questions**

## Question 5.5

### What makes visual tests flaky in Playwright Java?

#### Interview-Style Answer

Visual tests become flaky when screenshots capture unstable or environment-dependent UI states. Common causes include dynamic content, animations, timestamps, ads, random data, loading states, external images, different fonts, different viewport sizes, device scale factor differences, and CI rendering differences.

In Playwright Java, visual tests should compare a stable and repeatable UI state. The test should use controlled data, wait for the final visible state, keep browser and viewport settings consistent, and hide or control dynamic regions before screenshot comparison.

#### Detailed Explanation

Visual flakiness happens when the screenshot changes even though the application behavior is correct. Unlike normal locator assertions, screenshot comparison is sensitive to pixels, spacing, fonts, images, animations, and rendering differences.

Common visual flakiness causes:

1. Current date/time
2. Random IDs
3. Animations
4. Loading spinner
5. Dynamic charts
6. External images
7. Browser/font differences
8. Different viewport
9. Data changes
10. Cursor/focus state

For example, a test may fail in CI because a dashboard chart has different data, a timestamp changed, a web font loaded differently, or the screenshot was taken while a spinner was still disappearing. These are not always application bugs; they may be test stability issues.

A better approach is to wait for a meaningful final UI state before comparing screenshots:

```
PlaywrightAssertions.assertThat(
    page.getByTestId("dashboard-loaded")
).isVisible();

Locator dashboardCard = page.getByTestId("dashboard-card");

PlaywrightAssertions.assertThat(dashboardCard)
    .hasScreenshot("dashboard-card.png");
```

Dynamic regions should be controlled, hidden, or avoided when they are not part of the visual requirement:

```
page.locator("[data-testid='current-time']")
    .evaluate("el => el.style.visibility = 'hidden'");

PlaywrightAssertions.assertThat(
    page.getByTestId("report-summary")
).hasScreenshot("report-summary.png");
```

Stabilization methods:

- Use stable test data.
- Wait for the final UI state.
- Hide or mask dynamic sections.
- Disable or wait for animations where possible.
- Use consistent viewport and device scale factor.
- Keep browser versions and fonts consistent in CI.
- Prefer component screenshots over full-page screenshots.
- Capture trace, screenshot, and diff artifacts for failure analysis.

For reliable visual testing, the framework should separate real UI regressions from test noise. If a visual test fails, compare the expected, actual, and diff screenshots before updating the baseline. The cause may be a real layout issue, but it may also be unstable data, different rendering, missing fonts, or an early screenshot.

## Common Mistake

comparing screenshots before the page reaches a stable final state, which causes failures due to loading, animation, dynamic data, or environment differences instead of real visual regressions.



---

## **6. Playwright Vs Selenium**

---

## **Part I - Core Questions**

## Question 6.4

### How is Playwright `Locator` different from Selenium `WebElement`?

#### Interview-Style Answer

A Playwright `Locator` represents a way to find an element, while Selenium `WebElement` represents a specific element reference that has already been found in the DOM.

This difference is important for modern dynamic applications. In Selenium, a stored `WebElement` can become stale if the DOM is re-rendered. In Playwright Java, a `Locator` is re-evaluated when an action or assertion is performed, so it works better with dynamic UI updates, auto-waiting, and web-first assertions.

#### Detailed Explanation

In Selenium Java, when you call `findElement()`, Selenium returns a `WebElement` that points to a specific DOM node at that moment. If the page re-renders, replaces the element, or updates part of the DOM, that stored `WebElement` may no longer be valid and can cause a stale element problem.

In Playwright Java, a `Locator` is different. It does not permanently store one DOM node. It stores the strategy for finding the element. When you perform an action like `click()` or an assertion like `isVisible()`, Playwright resolves the locator at that time and applies its auto-waiting and actionability checks.

Example:

```
Locator saveButton = page.getByRole(  
    AriaRole.BUTTON,  
    new Page.GetByRoleOptions().setName("Save")  
);  
  
saveButton.click();  
  
PlaywrightAssertions.assertThat(saveButton).isEnabled();
```

Here, `saveButton` is not a fixed DOM element reference like a Selenium `WebElement`. It is a reusable locator query. When `click()` runs, Playwright finds the matching button and waits until it is actionable. When the assertion runs, Playwright checks the current state of the matching button.

---

This improves reliability in SPAs where components are frequently re-rendered after API calls, state changes, validation updates, or route changes.

Good locator design still matters. The locator should represent user intent and target a unique, stable element. Prefer role locators, labels, accessible names, visible text, scoped locators, or stable test IDs where appropriate:

```
Locator submitButton = page.getByRole(  
    AriaRole.BUTTON,  
    new Page.GetByRoleOptions().setName("Submit order")  
);
```

If a locator matches multiple elements, Playwright strict mode can fail actions that require a single element. In that case, improve the locator, scope it to a section, or use `filter()`, `first()`, `last()`, or `nth()` only when the choice is intentional.

## Common Mistake

treating Playwright `Locator` like a stored Selenium `WebElement` and adding unnecessary re-find logic, manual stale-element handling, or hard waits instead of relying on locator re-resolution, auto-waiting, and web-first assertions.

---

## Question 6.8

### How should Selenium XPath-heavy tests be migrated to Playwright Java?

#### Interview-Style Answer

Selenium XPath-heavy tests should not be migrated by simply copying the same XPath into Playwright Java. They should be reviewed and rewritten using Playwright's user-facing locator strategy wherever possible, such as role locators, labels, text, placeholder, accessible names, scoped locators, and stable test IDs.

The goal is to make the test express user intent. For example, instead of locating the second button inside the third `div`, the test should locate the Approve button inside the row for invoice `INV-1001`. XPath can still be used when there is no better option, but it should not be the default migration strategy.

#### Detailed Explanation

A direct XPath-to-Playwright migration may technically work, but it carries old Selenium weaknesses into the new framework. Many Selenium suites contain fragile XPath selectors based on DOM position, nested `div` structure, indexes, CSS classes, or generated attributes. These selectors break easily when the UI is refactored, even if the user-visible behavior has not changed.

Weak migrated locator:

```
page.locator("//div[3]/button[2]").click();
```

This locator does not explain the user action. It depends on page structure and position, so it can break when another button, wrapper, or layout container is added.

A better Playwright Java locator is:

```
page.getByRole(
    AriaRole.BUTTON,
    new Page.GetByRoleOptions().setName("Approve")
).click();
```

This is more readable because it matches how the user understands the page: click the button named Approve.

---

For repeated elements such as tables, cards, and search results, the locator should be scoped by business identity:

```
Locator row = page.getByRole(AriaRole.ROW)
    .filter(new Locator.FilterOptions().setHasText("INV-1001"));

row.getByRole(
    AriaRole.BUTTON,
    new Locator.GetByRoleOptions().setName("Approve")
).click();
```

This is better than using row indexes because it says: find the invoice row for **INV-1001**, then click its Approve button.

Good migration rules include:

1. Replace positional XPath with role, label, text, placeholder, or test-id locators.
2. Use accessible names for buttons, links, inputs, and headings.
3. Scope locators to a section, row, card, dialog, or form before acting.
4. Use filter() when selecting from repeated UI elements.
5. Use nth(), first(), or last() only when the order itself is meaningful.
6. Keep XPath only for rare cases where user-facing locators are not available.
7. Improve the application markup if poor accessibility prevents stable locators.

XPath may still be acceptable for legacy pages, SVGs, complex DOM relationships, or areas without accessible markup. But even then, the XPath should be stable and intentional, not based on random hierarchy or indexes.

## Common Mistake

mechanically converting Selenium XPath selectors into `page.locator("//...")` without checking whether the locator represents user intent, uniqueness, accessibility, and long-term stability after UI refactoring.

---

---

## **7. Emulation (Mobile)**

---

## **Part I - Core Questions**



---

## Question 7.8

**What are the limitations of Playwright mobile emulation compared with real-device testing?**

### Interview-Style Answer

Playwright mobile emulation is useful for validating responsive layout, viewport behavior, touch support, device scale factor, user agent, locale, timezone, geolocation, and permissions. However, it does not fully replace real-device testing because it cannot completely reproduce physical hardware, mobile operating system behavior, real browser-device combinations, real network conditions, sensors, battery impact, performance, native keyboard behavior, or complex gesture-heavy workflows.

In Playwright Java, mobile emulation is configured through `browser.newContext()` options such as `setViewportSize()`, `setScreenSize()`, `setDeviceScaleFactor()`, `setIsMobile()`, `setHasTouch()`, `setUserAgent()`, permissions, locale, and timezone. Emulation should be treated as fast CI-friendly coverage, while real devices or cloud-device testing should be used for high-risk mobile scenarios.

### Detailed Explanation

Playwright mobile emulation is excellent for catching many mobile and responsive defects early. It helps verify whether the mobile menu appears, desktop sidebar is hidden, cards stack correctly, buttons remain accessible, and layout behaves correctly across selected breakpoints.

Example mobile-like context in Playwright Java:

```
BrowserContext mobileContext = browser.newContext(  
    new Browser.NewContextOptions()  
        .setViewportSize(390, 844)  
        .setScreenSize(390, 844)  
        .setDeviceScaleFactor(3)  
        .setIsMobile(true)  
        .setHasTouch(true)  
        .setUserAgent("Mozilla/5.0 Mobile")  
);  
  
Page page = mobileContext.newPage();  
page.navigate(baseUrl + "/dashboard");  
  
PlaywrightAssertions.assertThat(  
    page.getByRole(  
        AriaRole.BUTTON,  
        new Page.GetByRoleOptions().setName("Menu")  
    )  
).isVisible();
```

Use Playwright emulation for:

1. CI-friendly responsive regression
2. Mobile navigation checks
3. Basic touch-enabled flows
4. Viewport and breakpoint testing
5. Locale, timezone, geolocation, and permission scenarios
6. Early detection of mobile layout defects

But real devices may still reveal issues that emulation cannot fully reproduce. These include actual device performance, mobile browser quirks, native keyboard behavior, physical touch gestures, camera behavior, biometric flows, sensor behavior, battery impact, device memory limitations, and real network variation.

Use real devices or cloud-device testing for:

1. High-risk mobile releases
2. Gesture-heavy features
3. Performance-sensitive mobile flows
4. Camera, biometric, sensor, or file-upload scenarios
5. Device/browser-specific production defects
6. Final confidence before major mobile releases

A good mobile test strategy does not depend only on one emulated device or one viewport. It uses a small responsive matrix in Playwright Java for fast automated feedback and adds real-device validation for the flows where actual hardware, browser, OS, performance, or touch behavior can affect the user experience.

## Common Mistake

Assuming Playwright mobile emulation gives 100% real-device confidence, instead of using it for fast responsive coverage and adding real-device or cloud-device testing for hardware, OS, performance,

---

native keyboard, browser-specific, and gesture-heavy scenarios.

---

## Question 7.15

**When should you prefer `browser.newContext()` viewport configuration over `page.setViewportSize()`?**

### Interview-Style Answer

Prefer `browser.newContext()` viewport configuration when the test should start in a specific desktop, tablet, or mobile profile from the beginning of the page lifecycle. This is better for most responsive and mobile tests because the application loads directly with the intended viewport, mobile mode, touch support, user agent, locale, timezone, permissions, and storage state.

Use `page.setViewportSize()` when the test specifically needs to verify runtime resize behavior, such as how the UI reacts when the browser window changes size after the page is already loaded.

### Detailed Explanation

Context-level viewport configuration is usually the cleaner and more realistic option for responsive testing. Many applications make layout or feature decisions during the initial page load. For example, the application may decide which navigation component to render, which API payload to request, which feature flags to enable, or whether to load mobile-specific JavaScript based on the initial viewport, user agent, or mobile context settings.

Example:

```
BrowserContext mobileContext = browser.newContext(  
    new Browser.NewContextOptions()  
        .setViewportSize(390, 844)  
        .setIsMobile(true)  
        .setHasTouch(true)  
);  
  
Page mobilePage = mobileContext.newPage();  
mobilePage.navigate("https://example.com");  
  
PlaywrightAssertions.assertThat(  
    mobilePage.getByRole(  
        AriaRole.BUTTON,  
        new Page.GetByRoleOptions().setName("Menu")  
    )  
).isVisible();
```

This is better than loading the page as desktop and resizing later because the page starts directly in the intended mobile-like profile.

---

`page.setViewportSize()` is useful for a different purpose. It helps when the requirement is to test dynamic resizing after the page has already loaded.

Example:

```
Page page = context.newPage();
page.navigate("https://example.com");

page.setViewportSize(390, 844);

PlaywrightAssertions.assertThat(
    page.getByRole(
        AriaRole.BUTTON,
        new Page.GetByRoleOptions().setName("Menu")
    )
).isVisible();
```

This checks whether the application responds correctly to runtime viewport changes. That is useful for validating resize listeners, responsive CSS changes, collapsible layouts, or desktop-to-mobile transitions. But it may not be identical to a fresh mobile page load because some application behavior may be initialized only once during startup.

In a framework, I would normally create separate contexts for desktop, tablet, and mobile profiles using `browser.newContext()`. I would use `page.setViewportSize()` only for tests whose purpose is specifically to validate resize behavior.

## Common Mistake

Loading the page in desktop mode, resizing it to mobile with `page.setViewportSize()`, and assuming it is identical to opening the page directly in a mobile-configured `BrowserContext`. This can miss startup-time differences such as mobile navigation rendering, feature flags, API behavior, user-agent logic, and touch-specific behavior.

---

---

## 8. CI

---

## **Part I - Core Questions**

## Question 8.6

### How would you make Playwright Java CI execution reliable across different build agents?

#### Interview-Style Answer

I would make Playwright Java CI execution reliable by standardizing the build environment across all agents. That includes Java version, Maven or Gradle version, Playwright Java version, browser binaries, OS image, required system packages, environment variables, timezone, locale, secrets, test data setup, and artifact collection.

For parallel CI execution, I would also ensure that each test uses isolated `BrowserContext` and `Page`, separate users where needed, parallel-safe test data, worker-specific files, and safe storage-state handling. The goal is that the same commit behaves the same way on every build agent.

#### Detailed Explanation

CI reliability depends heavily on reproducibility. If one build agent uses a different Java version, browser binary, OS package, timezone, locale, environment variable, or file path structure, the same Playwright Java test can pass on one agent and fail on another.

A reliable setup starts with a controlled CI image or agent configuration. The team should define approved versions for Java, Maven or Gradle, Playwright Java, and browser binaries instead of allowing each build agent to drift independently.

Recommended controls:

1. Use approved CI images or centrally managed build agents.
2. Pin Java, Maven, Gradle, and Playwright Java versions.
3. Install or cache Playwright browser binaries consistently.
4. Ensure required OS dependencies are available for browser launch.
5. Set explicit timezone and locale where tests depend on dates, currency, or formatting.
6. Use environment-specific configuration through variables, not hardcoded values.
7. Validate secrets and environment URLs before test execution.
8. Run a browser launch smoke check before starting the full suite.
9. Capture environment metadata in reports.
10. Store traces, screenshots, videos, logs, and reports with build number and commit ID.

For example, the CI report should make it easy to identify where the test ran:



```
Build: 1842
Commit: a7f9c21
Environment: QA
Browser: chromium
Java: 17
Playwright Java: 1.x
Agent: linux-agent-03
Timezone: Asia/Kolkata
```

Parallel execution also needs strict isolation. Browser isolation alone is not enough if tests share backend users, orders, carts, files, or storage-state files. Each test should use a fresh `BrowserContext` and `Page`, and each worker should use safe data and file locations.

Example:

```
BrowserContext context = browser.newContext(
    new Browser.NewContextOptions()
        .setStorageStatePath(Paths.get("auth/buyer-ci.json"))
);

Page page = context.newPage();
```

For worker-specific data:

```
String workerId = System.getenv().getOrDefault("CI_NODE_INDEX", "local");
String orderId = "ORD-" + workerId + "-" + UUID.randomUUID();
```

For worker-specific files:

```
Path downloadDir = Paths.get("target/downloads", workerId,
    UUID.randomUUID().toString());
Files.createDirectories(downloadDir);
```

This prevents one build agent or parallel worker from overwriting another worker's downloads, uploads, traces, screenshots, temporary files, or test data.

The pipeline should also publish useful evidence when failures occur. If a test fails only on one agent, traces, screenshots, videos, console logs, network logs, and environment metadata help determine whether the issue is product behavior, test code, data collision, missing dependency, browser mismatch, or CI environment drift.

## Common Mistake

Debugging the test code repeatedly while ignoring CI environment drift, such as different browser binaries, Java versions, OS packages, timezone, locale, secrets, shared users, shared test data, or shared download folders across build agents.

### Question 8.13

## How would you configure headed and headless execution in Playwright Java?

### Interview-Style Answer

I would configure headed or headless execution using `BrowserType.LaunchOptions().setHeadless()` and control it through a system property, environment variable, or framework configuration instead of changing source code.

In practice, headless mode is usually used in CI for speed and repeatability, while headed mode is useful for local debugging. Even when switching between headed and headless modes, each test should still use isolated `BrowserContext`, clean test data, separate users where needed, and worker-specific files for downloads, uploads, traces, and screenshots.

### Detailed Explanation

Playwright Java supports headed and headless browser execution through launch options. The framework should make this configurable so the same test code can run locally in headed mode and in CI in headless mode.

Example:

```
boolean headless = Boolean.parseBoolean(
    System.getProperty("headless", "true")
);

Browser browser = playwright.chromium().launch(
    new BrowserType.LaunchOptions().setHeadless(headless)
);
```

Run in headless mode:

```
mvn test -Dheadless=true
```

Run in headed mode:

```
mvn test -Dheadless=false
```

A framework can also combine this with browser selection:

```
String browserName = System.getProperty("browser", "chromium");
boolean headless = Boolean.parseBoolean(
    System.getProperty("headless", "true")
);

BrowserType.LaunchOptions launchOptions =
    new BrowserType.LaunchOptions().setHeadless(headless);

Browser browser;

switch (browserName.toLowerCase()) {
    case "firefox":
        browser = playwright.firefox().launch(launchOptions);
        break;
    case "webkit":
        browser = playwright.webkit().launch(launchOptions);
        break;
    default:
        browser = playwright.chromium().launch(launchOptions);
}
```

Headed mode is useful when debugging locator issues, popups, visual behavior, animations, or responsive layout differences. Headless mode is better for CI because it is faster, easier to run on build agents, and does not require a visible desktop session.

However, changing headed/headless mode should not change the test design. The suite should still use:

- Fresh BrowserContext and Page per test
- Parallel-safe test data
- Separate users or role-specific storage state where needed
- Worker-specific downloads, uploads, screenshots, traces, and temp files
- Proper teardown after execution
- Useful artifacts for CI failures

If a test passes only in headed mode but fails in headless mode, the issue should be investigated using traces, screenshots, videos, browser/version comparison, viewport settings, timing, and environment differences. The solution should not be to permanently force headed mode in CI unless there is a very specific reason.

## Common Mistake

Hardcoding `setHeadless(false)` or editing source code every time debugging is needed, instead of controlling headed/headless mode through Maven, Gradle, CI variables, or framework configuration.

## Question 8.16

### How would you prevent timezone and locale drift between local and CI Playwright runs?

#### Interview-Style Answer

I would prevent timezone and locale drift by explicitly setting `timezoneId` and `locale` in `Browser.NewContextOptions` for tests where date, time, currency, number formatting, sorting, or localized text matters.

Local machines may run in `Asia/Kolkata`, while CI agents may run in UTC or another region. If the browser context is not controlled, the same test can display different dates, timestamps, currencies, or translated text in local and CI runs.

#### Detailed Explanation

Timezone and locale drift is a common CI issue in applications that display dates, calendars, reports, currency, invoices, timestamps, or region-specific formats. A test may pass locally because the developer machine uses one timezone and locale, but fail in CI because the build agent uses UTC, US locale, or a different system configuration.

In Playwright Java, the safer approach is to define timezone and locale explicitly when creating the `BrowserContext`:

```
BrowserContext context = browser.newContext(  
    new Browser.NewContextOptions()  
        .setTimezoneId("Asia/Kolkata")  
        .setLocale("en-IN")  
);  
  
Page page = context.newPage();
```

This makes browser behavior more predictable. For example, if the application shows Indian date and currency formats, the test should not depend on whatever locale the CI runner happens to use.

Use explicit timezone and locale settings when validating:

- Calendar workflows
- Date pickers
- Report timestamps
- Time-based messages
- Booking or expiry dates
- Currency formatting
- Number formatting
- Locale-specific sorting
- Localized labels or messages

The expected values should also be calculated using the same intended timezone and locale. Otherwise, the browser may display **07/06/2026**, while the Java-side expected value is calculated as **06/06/2026** because the machine timezone differs.

Example:

```
ZonedDateTime nowInIndia = ZonedDateTime.now(
    ZoneId.of("Asia/Kolkata")
);

DateTimeFormatter formatter = DateTimeFormatter
    .ofPattern("dd MMM yyyy")
    .withLocale(new Locale("en", "IN"));

String expectedDate = nowInIndia.format(formatter);

PlaywrightAssertions.assertThat(
    page.getByTestId("report-date")
).hasText(expectedDate);
```

For CI reliability, the framework can centralize these settings in context creation instead of repeating them in every test. Different test suites can also define different locale profiles if the product supports multiple regions.

Timezone and locale settings are browser-context controls. They improve consistency for web application behavior, but they do not replace testing on real devices or real regional environments when the product has device-specific, OS-specific, or native-app behavior.

## Common Mistake

Setting timezone or locale only on the CI machine or calculating expected dates using the local system timezone, while the Playwright browser context displays dates, times, currency, or localized text using a different timezone or locale.

---

## 9. AI

---

## **Part I - Core Questions**

### Question 9.3

## What principles would you follow for using AI in a Playwright Java automation program?

### Interview-Style Answer

I would use AI to improve speed, consistency, failure analysis, coverage insight, test review, and knowledge sharing, but I would keep evidence, human review, security, and framework standards as mandatory controls.

AI should assist Playwright Java automation engineers; it should not become the final authority for code, locators, assertions, mocks, failure classification, or release decisions. Every AI-generated test or recommendation should be reviewed, validated through execution, checked against framework standards, and supported by real evidence from traces, screenshots, logs, network data, CI results, or product behavior.

### Detailed Explanation

AI can add value in a Playwright Java automation program by helping generate test ideas, review Page Objects, summarize failures, detect duplicate tests, identify flaky patterns, improve reports, and suggest locator or assertion improvements. But uncontrolled AI usage can create fake APIs, weak tests, unsafe data exposure, duplicated coverage, false confidence, and poor release decisions.

Important principles:

1. Use evidence-first recommendations.
2. Require human approval for code and framework changes.
3. Never share secrets, tokens, cookies, storage-state files, or sensitive artifacts carelessly.
4. Enforce Playwright Java framework standards.
5. Review all AI-generated tests before merge.
6. Validate AI output through compilation, execution, and CI.
7. Use AI for analysis and assistance, not blind authority.
8. Track AI recommendation accuracy over time.
9. Protect traces, screenshots, videos, logs, reports, and network payloads.
10. Use AI to reduce noise, improve confidence, and support better engineering decisions.

For example, if AI suggests fixing a locator failure, the recommendation should include evidence:



---

**Evidence:**  
Trace shows two visible Approve buttons.

**Risk:**  
A broad locator may click the wrong approval button.

**Recommended fix:**  
Scope the locator to the row containing the target invoice ID.

**Validation:**  
Run invoice approval tests and related table interaction tests.

A safer Playwright Java locator would be:

```
Locator row = page.getByRole(
    AriaRole.ROW,
    new Page.GetByRoleOptions().setName(Pattern.compile(invoiceId))
);

row.getByRole(
    AriaRole.BUTTON,
    new Locator.GetByRoleOptions().setName("Approve")
).click();
```

The same principle applies to AI-generated tests. A generated test should not be accepted just because it compiles. It must use stable locators, meaningful user-visible assertions, safe test data, isolated [BrowserContext](#) and [Page](#) usage, secure credentials, scoped mocks, clear cleanup, and CI-compatible execution.

I would also define governance: approved AI tools, prompt templates, prohibited data, artifact redaction rules, review checklists, CI validation, audit expectations, and escalation rules for risky recommendations.

## Common Mistake

using AI as a replacement for automation engineering judgment, instead of using it as a controlled assistant that improves Playwright test quality, debugging speed, review consistency, and release confidence.

---

## Question 9.8

### How would you train a Playwright Java team to use AI responsibly?

#### Interview-Style Answer

I would train the team to use AI as an assistant for drafting, reviewing, debugging, and learning, not as an authority that can replace engineering judgment.

The training should cover safe prompt writing, artifact redaction, Playwright Java framework standards, review of AI-generated code, CI validation, secret protection, and the limits of AI recommendations. Engineers should know how to detect hallucinated APIs, weak locators, missing assertions, unsafe data handling, duplicate scenarios, and misleading failure analysis before accepting AI output.

#### Detailed Explanation

Responsible AI usage requires both technical skill and judgment. A Playwright Java team should understand where AI is helpful and where it can be risky.

Training topics should include:

1. What AI can help with
2. What data must not be shared
3. How to redact traces, screenshots, logs, and network payloads
4. How to ask structured failure-analysis questions
5. How to review AI-generated Playwright Java code
6. How to validate locator and assertion quality
7. How to detect hallucinated APIs or TypeScript-style code in Java
8. How to use traces, screenshots, videos, logs, and network evidence
9. How to document AI-assisted decisions
10. When to escalate to senior engineers

For Playwright Java code generation, the team should be trained to check whether the output uses valid APIs, avoids `Thread.sleep()`, uses stable locators, respects `BrowserContext` and `Page` isolation, protects credentials, creates safe test data, and includes meaningful user-visible assertions.

For example, AI may generate a test that clicks a button and stops:

```
page.getByRole(  
    AriaRole.BUTTON,  
    new Page.GetByRoleOptions().setName("Submit")  
)<code>.click();
```

---

The team should know this is incomplete unless the final business outcome is validated:

```
page.getByRole(  
    AriaRole.BUTTON,  
    new Page.GetByRoleOptions().setName("Submit")  
) .click();  
  
PlaywrightAssertions.assertThat(  
    page.getByText("Request submitted successfully")  
) .isVisible();
```

The team should also be trained on data safety. They should not paste credentials, tokens, cookies, storage-state files, customer data, payment details, confidential screenshots, or full production traces into AI tools unless the tool and process are approved for that data.

Responsible usage also means documenting AI-assisted decisions when needed. If AI recommends a locator fix, mock change, or failure classification, the engineer should verify it with trace evidence, run the impacted tests, and keep human ownership of the final decision.

## Common Mistake

giving teams AI tools without teaching safe prompt writing, artifact redaction, Playwright Java API validation, locator standards, assertion quality, secret protection, CI validation, and the limits of AI-generated recommendations.

---

## Question 9.17

### How would you prevent AI usage from weakening engineering skills in a Playwright team?

#### Interview-Style Answer

I would position AI as a learning, review, and productivity assistant rather than a replacement for engineering understanding. Engineers should be expected to understand, explain, and validate AI-generated recommendations before using them.

The team should continue developing skills in Playwright Java fundamentals such as locator design, actionability, web-first assertions, Trace Viewer analysis, BrowserContext management, test data strategy, network debugging, and framework design. AI should accelerate learning and problem-solving, not become a substitute for technical competence.

#### Detailed Explanation

One of the biggest risks of AI adoption is that engineers may begin accepting recommendations without understanding why they are correct. Over time, this can weaken debugging skills, framework design capability, code review quality, and technical ownership.

Good practices include:

1. Require engineers to explain AI-suggested fixes before merging them.
2. Review why a locator strategy is correct.
3. Review why a wait or assertion strategy is correct.
4. Use AI outputs as learning material during team discussions.
5. Keep manual debugging skills active.
6. Require engineers to analyze traces and failure evidence.
7. Keep framework design decisions human-led.
8. Pair junior engineers with experienced reviewers.
9. Encourage engineers to challenge AI recommendations.
10. Measure understanding, not just implementation speed.

For example, if AI suggests replacing:

```
Thread.sleep(5000);
```

with:

```
PlaywrightAssertions.assertThat(  
    page.getByTestId("order-status")  
) .hasText("Submitted");
```

the engineer should be able to explain:

- 
- Why hard waits are unreliable.
  - Why web-first assertions are preferred.
  - What condition is actually being validated.
  - How the assertion improves reliability.
  - How it behaves in CI and parallel execution.

Similarly, when AI recommends a locator change, engineers should understand why the new locator is more stable, whether it relies on accessible names, whether it is unique, and whether it will remain maintainable after UI changes.

Organizations should also continue conducting framework reviews, debugging workshops, failure-analysis sessions, and architecture discussions where engineers solve problems using traces, screenshots, videos, network logs, and Playwright behavior rather than relying solely on AI-generated answers.

A healthy AI-enabled team becomes stronger over time because AI handles repetitive analysis while engineers focus on deeper reasoning, framework design, risk assessment, and quality decisions.

## Common Mistake

allowing engineers to copy AI-generated Playwright Java code, locators, waits, assertions, or fixes into the framework without understanding why they work, how they affect reliability, or what trade-offs they introduce.

---

### Question 9.34

## How would AI help detect weak Page Object design?

### Interview-Style Answer

AI can detect weak Page Object design by reviewing Playwright Java page classes for unclear responsibility boundaries, generic methods, duplicated locators, hidden assertions, mixed API setup, utility logic inside page classes, and missing component extraction.

A strong Page Object should represent user-facing actions and page-specific validations. Reusable widgets should become component objects, while API setup, data generation, file handling, and framework utilities should stay outside Page Objects. This keeps ownership clear, reduces duplication, improves review quality, and helps the suite scale in CI/CD.

### Detailed Explanation

A weak Page Object often becomes a dumping ground for locators, business flows, assertions, API calls, test data setup, and utility logic. It may work initially, but over time it becomes hard to review, reuse, debug, and maintain.

AI can detect design issues such as:

1. Generic methods such as `clickButton()`, `enterText()`, or `verifyText()`
2. Missing business-readable methods such as `submitOrder()` or `approveInvoice()`
3. Test data creation inside Page Object classes
4. API calls mixed into UI page actions
5. Hidden assertions inside action methods
6. Same locator repeated across many classes
7. Very large page class representing multiple screens
8. Repeated table, card, modal, or menu logic not extracted into components
9. Framework utilities mixed with product page behavior
10. Page Objects owned by the wrong team or shared without clear rules

Better responsibility separation:

---

#### Page Object:

- User-facing page actions
- Page-specific validations
- Page-specific locators

#### Component Object:

- Reusable tables, modals, cards, menus, filters, and widgets

#### Utility or Service:

- API setup
- Test data generation
- File parsing
- Date handling
- Environment configuration
- Cleanup logic

### Example of weak design:

```
class OrdersPage {
    void clickButton(String text) { }
    void createOrderThroughApi() { }
    void parseDownloadedFile() { }
    void assertOrderCreated() { }
}
```

### Better design:

```
class OrdersPage {
    void submitOrder(String orderName) { }
    void shouldShowOrderStatus(String orderId, String status) { }
}

class OrderApiClient {
    String createDraftOrder(OrderRequest request) { return "ORD-1001"; }
}

class DownloadFileVerifier {
    void shouldContainOrderId(Path file, String orderId) { }
}
```

AI can also help during PR review by flagging Page Objects that are growing too large, duplicating component logic, hiding assertions inside action methods, or mixing setup and UI behavior. These findings help teams maintain readable code, clear ownership, reusable components, and scalable automation architecture.

## Common Mistake

calling any class with locators a Page Object even when it mixes UI actions, assertions, API setup, test data, file utilities, and multiple screen responsibilities without clear design boundaries.

---

## Question 9.45

### How would you design confidence scoring for AI Playwright failure recommendations?

#### Interview-Style Answer

I would design confidence scoring based on evidence strength, artifact completeness, historical pattern match, reproduction consistency, and whether multiple signals point to the same root cause.

For Playwright Java failure analysis, a high-confidence recommendation should be backed by trace evidence, screenshot or video state, console or network logs, CI history, retry behavior, and a known failure pattern. If the evidence is incomplete or multiple causes are possible, the AI should clearly mark the recommendation as medium or low confidence.

#### Detailed Explanation

Confidence scoring should not be a random percentage. It should explain why the AI believes a recommendation is strong, uncertain, or weak.

Useful confidence factors include:

1. Trace Viewer confirms the failing page state
2. Screenshot or video supports the same conclusion
3. Network response or console error matches the suspected cause
4. Same failure pattern occurred previously
5. Failure is reproducible in retry or rerun
6. Test data setup confirms or rejects data-related issues
7. Browser and environment behavior are consistent
8. Recent code changes are linked to the failing module
9. Known issue or known fix exists
10. No conflicting evidence is found

A simple confidence model can be:



High confidence:

- Same error pattern occurred before
- Trace confirms the same page state
- Network or console evidence matches
- Failure is reproducible
- Known fix or known owner exists

Medium confidence:

- Error pattern is similar to previous failures
- Some artifacts support the suspected cause
- Artifacts are incomplete
- Root cause is likely but not fully proven

Low confidence:

- Only stack trace or timeout message is available
- No trace, screenshot, video, or network evidence
- Multiple root causes are possible
- Failure is new, inconsistent, or not reproducible

## Example:

Recommendation:

Likely backend issue in invoice loading.

Confidence:

High

Evidence:

- Trace shows invoice table stayed in loading state.
- Network log shows /api/invoices returned 504.
- Screenshot confirms the loading spinner remained visible.
- Same pattern occurred five times in staging this week.
- Retry passed after the API returned 200.

## Another example:

Recommendation:

Possible locator issue.

Confidence:

Low

Evidence:

- Only timeout error is available.
- No trace or screenshot was captured.
- The locator may be wrong, but page state is unknown.
- Test data and network behavior were not available.

This makes AI recommendations safer because engineers can see the level of uncertainty before acting. High-confidence recommendations may be triaged faster, while medium and low-confidence cases should require deeper engineer review before changing test code, quarantining tests, or raising defects.

## Common Mistake

showing AI failure recommendations without confidence level, supporting evidence, uncertainty, or explanation of why the

---

recommendation should be trusted.

---

---

# Final Interview Preparation Checklist

Use this checklist before attending a Playwright Java interview.

## Concept Readiness

- I can explain what Playwright is and why it is useful for modern web automation.
- I can explain the Playwright Java object model: [Playwright](#), [Browser](#), [BrowserContext](#), [Page](#), [Locator](#), and [FrameLocator](#).
- I can explain browser context isolation, storage state, authentication reuse, and role-based sessions clearly.
- I can explain auto-waiting, actionability checks, locator re-resolution, and web-first assertions.
- I can choose reliable locators using role, label, text, test id, scoping, filtering, and chaining.
- I can explain actions such as click, fill, check, select, upload, download, drag-and-drop, hover, keyboard, and mouse actions.
- I can handle frames, popups, dialogs, downloads, uploads, events, multiple tabs, and Shadow DOM scenarios.
- I can explain network interception, API mocking, HAR usage, API response validation, and UI/API combined testing.
- I can debug failures using Trace Viewer, screenshots, videos, Inspector, console logs, network evidence, and CI artifacts.
- I can explain flaky test root causes and prevention strategies, including locator design, test data, isolation, timing, and CI environment issues.
- I can explain mobile emulation, viewport, device profile, touch events, responsive testing, and mobile limitations.
- I can explain visual testing, snapshot testing, accessibility testing, cross-browser testing, WebView2, Selenium Grid integration, and advanced Playwright topics.
- I can explain test design, Page Object Model, reusable components, framework design, CI execution, reporting, test data strategy, and parallel-safe execution.
- I can compare Playwright Java with Selenium Java honestly and practically.
- I can answer architect-level and leadership questions with trade-offs, governance, maintainability, ROI, ownership, quality gates, release

---

readiness, and long-term automation strategy.

## 5-Book Revision Checklist

Use the five books in this order for final interview preparation:

### 1. Book 1 - Playwright

Revise Playwright Java foundations, setup, browser execution, contexts, pages, locators, actions, assertions, frames, network, screenshots, videos, traces, and core APIs.

### 2. Book 2 - Automation Scenarios & Debugging

Practice real project scenarios, workflow validation, UI/API behavior, troubleshooting, debugging evidence, and failure analysis.

### 3. Book 3 - Framework & Test Design

Strengthen test design, Page Object Model, reusable components, framework structure, maintainability, and parallel-safe test data.

### 4. Book 4 - Architect/Leadership

Prepare for senior, lead, and architect-level discussions on strategy, governance, ROI, ownership, release readiness, reliability, and leadership decisions.

### 5. Book 5 - Advanced Playwright Topics

Revise advanced topics such as Selenium Grid, snapshot testing, accessibility, cross-browser and visual testing, mobile emulation, CI, AI, touch events, and WebView2.

## Answering Strategy in Interviews

For every answer, follow this pattern:

1. Direct answer
2. Why it matters in real projects
3. Playwright Java API or example
4. Risk, limitation, or trade-off
5. Common mistake
6. Best practice or decision rule

## Final Advice

Do not memorize only API names. Interviewers usually look for practical understanding:

- Why this feature exists
- When to use it

- 
- When not to use it
  - What can go wrong
  - How to debug it
  - How to design it in a maintainable framework
  - How it behaves in CI, parallel execution, and real project conditions
  - How to explain trade-offs like a senior automation engineer

If you can explain the reasoning behind each answer, you can handle beginner, project-level, senior, architect, and leadership-level Playwright Java interviews confidently.

---

---

## About Crack Playwright Interviews

Crack Playwright Interviews is a focused learning platform for automation engineers who want to master Playwright Java interviews through concept clarity, practical scenarios, debugging, framework design, advanced Playwright topics, and architect-level preparation.

Website: [www.crackplaywrightinterviews.com](http://www.crackplaywrightinterviews.com)

---

---

## About the Author

R. Rajamanthiram is a Quality Engineering leader and Automation Test Architect with 22+ years of experience in software quality engineering, test automation, framework design, and automation strategy.

He has worked extensively on automation architecture, enterprise test frameworks, UI automation, API testing, CI/CD quality integration, and interview preparation for QA and automation engineers.

He has taken hundreds of interviews and has worked in premium product companies, bringing practical hiring and real-project automation experience into this ebook.

Website: [www.crackplaywrightinterviews.com](http://www.crackplaywrightinterviews.com)

LinkedIn: <https://www.linkedin.com/in/rajamanthiram/>

---

---

## Copyright and Disclaimer

This ebook is intended for educational and interview preparation purposes. Playwright is a trademark of Microsoft. This ebook is independently created and is not an official Microsoft Playwright publication.

All examples are provided for learning purposes and should be adapted to your project standards, application architecture, security policies, test data policies, compliance requirements, and CI/CD environment.

The questions and answers are designed to help readers prepare for interviews and improve practical understanding. They should not be treated as a substitute for official documentation, project-specific coding standards, security review, accessibility review, or production engineering judgment.





THE ULTIMATE GUIDE TO **PLAYWRIGHT (JAVA)** INTERVIEW SUCCESS



# SAMPLE BOOK

# 100+

## QUESTIONS



Crack **Playwright (Java) Interviews** is a premium interview-preparation guide for automation engineers who want deep, structured, and practical Playwright Java interview readiness. It covers foundations, real project scenarios, mobile automation, framework design, debugging, CI, and architect-level thinking in a highly organized question-and-answer format.

## INSIDE THIS EBOOK



**1800+** questions



**250+** automation scenario questions



**280+** mobile questions



**150+** framework and test design questions



**250+** architect questions



Structured into **5 books**: Playwright, Automation, Scenarios & Debugging, Framework & Test Design, Architect/Leadership, and Advanced Playwright Topics.



*Learn  
every important  
Playwright Java  
concept with  
confidence.*



## ABOUT THE AUTHOR

**R. Rajamanthiram** is a Quality Engineering leader and Automation Test Architect with 22+ years of experience in software quality engineering, automation architecture, framework design, and interview preparation. He has taken hundreds of interviews and worked in premium product companies.



ISBN 978-93-XXXX-XXXX-X



NOT FOR RESALE - SAMPLE BOOK



[www.crackplaywrightinterviews.com](http://www.crackplaywrightinterviews.com)