Best Practices for Playwright Framework with JavaScript, TypeScript, and Cucumber

# 1. Structure and Organization

1. Use a Layered Folder Structure  
 - Organize the project by keeping pages, step definitions, and test data separate.  
 project-root/  
 ├── pages/  
 ├── features/  
 ├── step-definitions/  
 ├── utils/  
 └── reports/  
2. Leverage TypeScript for Type Safety  
 - Use TypeScript to catch errors at compile time and enforce typing on objects.  
 let page: Page; // Enforces correct type  
3. Create Reusable Page Object Classes  
 - Abstract web elements and actions into classes following the Page Object Model (POM) pattern.  
4. Use Cucumber Tags for Test Filtering  
 - Use @tags in Cucumber to run specific test scenarios or groups.  
 npx cucumber-js --tags @smoke  
5. Separate Environment Configuration  
 - Use environment files or .env to manage different configurations (e.g., URLs, credentials).  
 baseURL = process.env.BASE\_URL || "https://staging.example.com";

# 2. Parallel Execution and Performance Optimization

6. Enable Parallel Test Execution  
 - Utilize Playwright's parallel workers and Cucumber’s parallel execution support to speed up the suite.  
7. Run Headless Mode for Faster CI Execution  
 - Use headless mode unless the test requires UI interaction feedback.  
8. Test Data Isolation with Browser Contexts  
 - Use newContext() for each scenario to ensure isolation of cookies, storage, and session data.  
9. Use Fixtures for Repetitive Setup/Teardown  
 - Use Playwright test fixtures to avoid redundant code in setup and teardown.

# 3. Reporting and Logs

10. Generate HTML Reports  
 - Use tools like Allure Reports or Cucumber HTML Reporter to visualize test results.  
11. Capture Screenshots on Failure  
 - Automatically capture screenshots whenever a test fails for easier debugging.  
 await page.screenshot({ path: `screenshots/error.png`, fullPage: true });  
12. Use Video Recording for Debugging  
 - Enable video recording for CI builds to diagnose flaky test behavior.  
13. Log Important Actions during Execution  
 - Log user actions and API calls to troubleshoot test failures quickly.

# 4. Error Handling and Flakiness Reduction

14. Use Explicit Waits Instead of Implicit Waits  
 - Avoid setTimeout() and rely on Playwright's waitForSelector() to handle asynchronous events.  
 await page.waitForSelector('button#submit');  
15. Retry Mechanism for Flaky Tests  
 - Use retry logic in Cucumber for scenarios that occasionally fail.  
 npx cucumber-js --retry 2  
16. Handle Network Interceptions and Mocking  
 - Mock APIs to control dependencies during tests using page.route().  
17. Avoid Global State Across Tests  
 - Ensure that test data isn’t shared between scenarios to prevent test interference.

# 5. Test Design and Scenario Writing

18. Follow the Gherkin Syntax Correctly  
 - Use Given-When-Then properly to describe test scenarios clearly.  
19. Use Backgrounds for Repetitive Steps  
 - Move common setup steps to Background sections to avoid duplication.  
20. Limit Scenario Complexity  
 - Keep each scenario focused on a single feature or behavior for better maintainability.  
21. Avoid Overusing Tags  
 - Use meaningful tags, but avoid over-cluttering scenarios with too many tags.  
22. Use Data Tables for Parameterization  
 - Use Cucumber data tables to run scenarios with multiple data sets.  
 When I fill the form with:  
 | field | value |  
 | username | admin |  
 | password | pass123 |

# 6. Code Quality and Maintainability

23. Follow Linting and Formatting Rules  
 - Use tools like ESLint and Prettier to maintain code consistency.  
24. Avoid Hardcoding Values  
 - Store values like URLs and credentials in environment variables or JSON files.  
25. Modularize Step Definitions  
 - Keep step definitions small and focused by modularizing reusable steps.  
26. Use Custom Commands for Common Actions  
 - Abstract frequently used actions (e.g., login) into helper functions or utilities.  
27. Version Control with Git  
 - Use Git and branching strategies to manage your Playwright framework.

# 7. CI/CD Integration

28. Integrate with CI Tools  
 - Set up your framework with CI/CD tools like Jenkins, GitHub Actions, or GitLab for automated builds.  
29. Run Tests in Docker Containers for Consistency  
 - Use Docker to ensure consistent test environments across machines.  
30. Notify Teams on Test Failures via Slack or Email  
 - Integrate CI pipelines with Slack or email notifications to alert the team on test failures.

# 8. Best Practices for Locators

31. Use Unique Selectors  
 - Always prefer unique IDs or data-test attributes to locate elements reliably.  
 Example: await page.locator('#submit-button').click();  
  
32. Avoid Using XPath Selectors  
 - Use CSS selectors over XPath as they are faster and easier to read.  
 Example: page.locator('button[type="submit"]');  
  
33. Leverage Playwright’s Strict Mode for Locators  
 - Use strict mode to ensure only one element matches the locator.  
 Example: page.locator('button', { hasText: 'Submit' }).click();  
  
34. Use Role-based Locators for Accessibility  
 - Use locators like getByRole() to target elements based on ARIA roles.  
 Example: page.getByRole('button', { name: 'Login' }).click();  
  
35. Chain Locators for Better Precision  
 - Use .locator() chaining to narrow down elements within a parent container.  
 Example: page.locator('.form-container').locator('input[name="email"]');  
  
36. Wait for Elements Before Interaction  
 - Always wait for elements to be visible or enabled before interacting.  
 Example: await page.locator('input#username').waitFor();  
  
37. Avoid Hardcoded Waits  
 - Use smart waits like waitForSelector() instead of static timeouts.  
 Example: await page.waitForSelector('.loading', { state: 'hidden' });  
  
38. Use nth-of-type() or :nth-child() for Multiple Elements  
 - Use CSS pseudo-classes to select specific elements from a list.  
 Example: page.locator('ul > li:nth-of-type(3)');  
  
39. Handle Dynamic Elements Gracefully  
 - Use text-based or attribute-based locators for elements with changing IDs.  
 Example: page.locator('div[data-status="active"]');  
  
40. Validate Element Presence with Assertions  
 - Use assertions to confirm element visibility and correctness.  
 Example: expect(await page.locator('#logout-button').isVisible()).toBeTruthy();

# 9. Do's and Don'ts for Playwright Framework Development

* ### Do's:
* - Do use meaningful test names and descriptions to make test reports readable.  
  - Do group related tests logically into feature files and step definitions.  
  - Do use reusable functions for repetitive actions like login, navigation, etc.  
  - Do prefer headless mode for CI pipelines to reduce resource usage.  
  - Do handle pop-ups and alerts explicitly to prevent unexpected failures.  
  - Do leverage Playwright’s built-in fixtures and test hooks for setup and teardown.  
  - Do mock network requests where necessary to avoid dependency on external services.
* ### Don'ts:
* - Don’t use hardcoded waits (like setTimeout) in tests as it can introduce flakiness.  
  - Don’t rely on CSS classes or attributes that are likely to change frequently.  
  - Don’t use shared global variables across tests; prefer context isolation.  
  - Don’t run multiple assertions in a single test without clear separation.  
  - Don’t mix UI and API tests in the same suite; keep them separate.  
  - Don’t ignore test failures; always investigate and fix them promptly.  
  - Don’t use Playwright's default browser context for all tests—create new contexts.