Penzar Merchant

Test & Quality Analyst

Follow: http://www.linkedin.com/in/penzar

Playwright Interview Questions Part-2

Q1: Your test is failing. Walk me through your debugging approach.

Answer:

Systematic Debugging Approach:

1. **Analyze Error Messages:** Read the complete error stack trace first
2. **Reproduce Locally:** Run the exact same test in your local environment
3. **Use Trace Viewer:** Best tool for post-execution analysis in CI environments
4. **Interactive Debug:** Use --debug flag for step-by-step execution
5. **Visual Evidence:** Check screenshots/videos for UI issues
6. **Network Analysis:** Inspect API calls and response times
7. **Environment Comparison:** Compare local vs CI differences

Debug Commands Arsenal:

npx playwright test --debug # Interactive debugging npx playwright test --headed # Visual browser execution npx playwright test --ui # UI mode for investigation npx playwright show-report # HTML report with failures npx playwright show-trace trace.zip # Open specific trace file

Pro Insight:
Trace Viewer lets you go back and forward through each action visually - it's like having a time machine for your test execution.

Common Mistake:
Jumping straight to code changes without analyzing the actual failure evidence first.

Story Example:
"I discovered a flaky test was failing due to a 3rd-party widget loading inconsistently. Using trace viewer analysis, I identified the race condition and implemented proper wait conditions, reducing flakiness from 15% to 0%."

Q2: How do you handle flaky tests?

Answer:

Root Cause Categories:

- **Timing Issues:** Race conditions, async operations

- **Environment Dependencies:** Network latency, resource availability
- **Test Data Conflicts:** Shared state between tests
- **External Dependencies:** 3rd party services, APIs

Progressive Solutions (Start → Advanced):

// ❌ Bad: Hard waits (Selenium approach) await page.waitForTimeout(5000); // ✅ Good: Condition-based waits await expect(page.locator('#data-table')).toContainText('Loaded'); //

✅ Better: Multiple conditions with smart waits await Promise.all([page.waitForLoadState('networkidle'), expect(page.locator('#spinner')).toBeHidden(), expect(page.locator('#content')).toBeVisible()]); // 🔥 Advanced: Custom retry with backoff async function waitForDataLoad(page, maxAttempts = 3) { for (let i = 0; i < maxAttempts; i++) { try { await expect(page.locator('[data-loaded="true"]')).toBeVisible({ timeout: 10000 }); return; } catch (error) { if (i === maxAttempts - 1) throw error; await page.reload(); } } }

Pro Insight:
Track flaky test metrics with a dashboard. If a test fails more than 5% of runs, it needs immediate attention. Use Playwright's built-in retry mechanisms before writing custom solutions.

Common Mistake:
Adding retries without fixing the root cause - this just masks the problem and wastes CI resources.

Story Example:
"I implemented a flaky test monitoring dashboard using our CI metrics. We went from 23% flaky tests to 3% by systematically fixing the worst offenders first, saving 2 hours of daily CI runtime."

Q3: Debugging tools comparison - when to use what?

Answer:

Tool/Feature	When to Use	Best For	Command/Usage
--debug	Step-by-step analysis	Complex logic issues	npx playwright test --debug
--headed	Visual verification	UI interaction problems	npx playwright test --headed
--ui	Test exploration	Understanding test flow	npx playwright test --ui
Trace Viewer	Post-failure analysis	CI/CD debugging	npx playwright show-trace

Tool/Feature	When to Use	Best For	Command/Usage
Screenshots	Automated evidence	Visual regression	await page.screenshot()

Pro Insight:
Traces are normally run in CI environment since locally you can use UI Mode for developing and debugging tests. Each tool serves a specific debugging scenario.

Advanced Tip: Combine trace viewer with HTML reports for comprehensive failure analysis. Trace shows what happened, HTML report shows patterns across test runs.

Q4: How do you debug tests that pass locally but fail in CI?

Answer:

// CI-optimized debugging configuration // playwright.config.ts export default { use: { // Evidence collection for CI failures screenshot: 'only-on-failure', video: 'retain-on-failure', trace: 'on-first-retry' // More efficient than 'retain-on-failure' }, // CI-specific optimizations ...(process.env.CI && { workers: '50%', // Official recommendation for CI fullyParallel: true, // Faster execution forbidOnly: true, // Prevent accidental test.only retries: 2, // Retry failed tests reporter: [['html'], ['github']], // CI-friendly reporters timeout: 30000 // Longer timeout for slower CI }) };

CI Debugging Systematic Checklist:

1. **Environment Differences:** OS, browser versions, screen resolution, timezone
2. **Resource Constraints:** Memory limits, CPU throttling, slower I/O
3. **Network Behavior:** Slower connections, request timeouts, blocked domains
4. **Timing Variations:** Different execution speeds affecting race conditions
5. **Parallel Execution:** Tests interfering with shared resources
6. **Permissions:** File system access, browser capabilities

Pro Insight:
Use trace: 'on-first-retry' instead of 'retain-on-failure' - it's more efficient and only captures traces when tests actually retry, reducing storage costs.

Common Mistake:
Setting trace to 'on' for all tests in CI - this creates massive artifacts and slows down execution significantly.

Story Example:
"I solved a mysterious CI failure by discovering our Docker container had a different timezone. Tests expecting specific date formats were failing. Added TZ=UTC to our CI environment variables."

Q5: How do you use hooks, annotations, and tags in Playwright?

Test Hooks (Setup & Cleanup):

// Setup and teardown at different levels test.beforeAll(async ({ browser }) => { // Run once before all tests in the file console.log('Starting test suite'); }); test.beforeEach(async ({ page }) => { // Run before each test await page.goto('/dashboard'); await page.evaluate(() => localStorage.clear()); }); test.afterEach(async ({ page }) => { // Cleanup after each test await page.close(); }); test.afterAll(async () => { // Run once after all tests console.log('Test suite completed'); });

🏷️ **Annotations (Test Metadata):**

test('user login functionality', async ({ page }) => { test.info().annotations.push({ type: 'issue', description: 'Bug #123' }); test.slow(); // Mark test as slow (3x timeout) // Test logic here }); // Skip tests conditionally test('mobile-only feature', async ({ page, isMobile }) => { test.skip(!isMobile, 'Test only for mobile devices'); // Mobile test logic }); // Mark tests as experimental test('new feature @experimental', async ({ page }) => { test.info().annotations.push({ type: 'experimental' }); // Feature test logic });

Tags (Test Organization & Selection):

// Tag tests for organized execution test('user registration @smoke @auth', async ({ page }) => { // Smoke test for authentication }); test('complex workflow @regression @slow', async ({ page }) => { // Full regression test }); test('payment processing @critical @payment', async ({ page }) => { // Critical payment functionality });

Running Tagged Tests:

Run specific tagged tests npx playwright test --grep "@smoke" # Only smoke tests npx playwright test --grep "@auth|@payment" # Auth OR payment tests npx playwright test --grep "@critical" --grep-invert "@slow" # Critical but not slow # Tag combinations npx playwright test --grep "(?=.*@smoke)(?=.*@auth)" # Smoke AND auth tests

Pro Insight:
Use hooks for consistent setup/teardown, annotations for test metadata and conditional execution, and tags for organized test runs. Combine them for powerful test organization strategies.

Common Mistake:
Putting too much logic in beforeEach hooks instead of using fixtures, or not using tags to organize test execution in CI pipelines.

Story Example:
"I organized our 500+ test suite using tags: @smoke for quick validation, @critical for

deployment gates, and @regression for full coverage. This reduced our CI feedback time from 45 minutes to 8 minutes for smoke tests."

Pro Tips: Debugging Mastery

Essential Debugging Mindset:

- 🔍 **Systematic Approach:** Follow consistent debugging steps every time
- 📊 **Data-Driven:** Use metrics to prioritize which flaky tests to fix first
- 🛠️ **Preventive:** Build debugging tools into your test framework
- 📚 **Knowledge Base:** Document common issues and their solutions

Interview Follow-ups You Should Prepare For:

- *"How do you prevent flaky tests in the first place?"*
- *"What metrics do you track for test stability?"*
- *"How do you handle test failures during production deployments?"*
- *"Describe the most challenging debugging scenario you've faced."*