**Penzar Merchant**

Test & Quality Analyst

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

---

# Playwright Interview Questions Part-1

## Q1: How do you initialize a browser?

**Answer:**

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

const browser = await chromium.launch();

const context = await browser.newContext();

const page = await context.newPage();
```

**Pro Insight:** Always use browser contexts for test isolation - fresh sessions without browser launch overhead.

**Common Mistake:** Launching new browsers for every test instead of reusing contexts.

***Story Example:*** *"I optimized our test suite by switching from browser-per-test to context-per-test, reducing execution time by 40%."*

## Q2: What are workers in Playwright?

**Answer:** Parallel threads that run tests simultaneously to reduce execution time.

```
// playwright.config.ts

export default {

  // Default: 50% of CPU cores, undefined = auto-detect

  workers: process.env.CI ? 2 : undefined

}
```

**Pro Insight:** By default, Playwright limits workers to 1/2 of CPU cores on your machine. Each worker gets its own browser instance. For CI, limit to 2-4 workers to avoid memory issues.

**Common Mistake:** Setting too many workers causing memory issues or test interference.

**Story Example:** "I leveraged workers in Jenkins to cut our 45-minute test suite to 12 minutes while maintaining stability."

## Q3: How do you handle waits and assertions?

```
// Playwright way (auto-waiting)
```

```
await expect(page.locator('#submit')).toBeVisible();


// Old way
await page.waitForTimeout(3000);
```

**Pro Insight:** Playwright has built-in auto-waiting. Use expect() with locators for smart waiting.

**Common Mistake:** Using waitForTimeout() instead of condition-based waits.

***Story Example:*** *"I eliminated 80% of flaky tests by replacing hardcoded waits with auto-waiting assertions."*

## Q4: What's your project structure?

```
📁 playwright-tests/
├── 📁 tests/
│   ├── 📁 e2e/        # End-to-end tests
│   ├── 📁 api/        # API tests
│   └── 📁 visual/     # Visual regression
├── 📁 page-objects/   # Page Object Models
├── 📁 fixtures/       # Test data & setup
├── 📁 utils/          # Helper functions
├── ⚙️ playwright.config.ts
└── 🔧 global-setup.ts
```

**Pro Insight:** Separate concerns: e2e, api, visual tests in different folders. Page objects for complex flows.

**Common Mistake:** Putting everything in one folder or mixing test types.

**Story Example:** "I restructured our monolithic test suite into this pattern, reducing maintenance overhead by 60%."

## Q5: Types of locators (Priority Order)

**Start Here (Easiest for Beginners):**

```
// 1. CSS selectors (familiar from Selenium)
page.locator('.submit-btn')
```

// 2. XPath (when CSS isn't enough)

page.locator('//button[text()="Submit"]')

**Good for Content:**

// 3. Text-based (simple and readable)

page.getByText('Welcome')

**Inbuilt Methods Given by Playwright:**

// 4. Data attributes (test-specific, stable)

page.locator('[data-testid="submit"]')

// 5. Role-based (Playwright's native approach)

page.getByRole('button', { name: 'Submit' })

**Pro Insight:** Start with CSS selectors for quick wins, then gradually adopt role-based locators for better stability and accessibility.

**Common Mistake:** Overusing CSS selectors that break with design changes.

**Story Example:** "I started my Playwright journey using familiar CSS selectors from my Selenium background, then gradually introduced role-based locators. This approach reduced our team's learning curve by 50% while still achieving 70% better test stability."

## Don't stop at definitions

**Interviewers want stories showing:**

- 🎯 Problem you solved
- 🧠 Technical decision reasoning
- 📈 Impact/results achieved
- 🔄 Lessons learned

**Follow-up Questions They'll Ask:**

- "How would you debug this failing test?"
- "What's your strategy for reducing flaky tests?"
- "How do you handle CI/CD integration?"