Playwright Interview Questions and Answers

A comprehensive guide covering beginner to advanced Playwright interview questions.

Basic Questions

1. What is Playwright and why would you use it?

Answer:

Playwright is a modern open-source automation framework developed by Microsoft for end-to-end testing of web applications. It supports multiple browsers (Chromium, Firefox, WebKit) and provides a single API to automate across all of them.

Key advantages:

• Cross-browser support (Chrome, Firefox, Safari via WebKit)

• Auto-wait mechanism - no need for explicit waits

• Powerful selector engine with multiple strategies

• Network interception and mocking capabilities

• Mobile emulation support

• Parallel test execution out of the box

• Built-in screenshot and video recording

• Strong support for modern web apps (SPAs, PWAs)

2. What are the different types of locators in Playwright?

Answer:

Playwright provides several locator strategies:

```
// CSS Selector
page.locator('#element-id')
page.locator('.class-name')
page.locator('div > button')

// Text-based
page.getByText('Submit')
page.getByText(/submit/i) // regex

// Role-based (recommended)
page.getByRole('button', { name: 'Submit' })
page.getByRole('textbox', { name: 'Username' })

// Label-based
page.getByLabel('Email address')

// Placeholder
page.getByPlaceholder('Enter your email')

// Alt text (for images)
page.getByAltText('Logo')

// Title attribute
page.getByTitle('Close dialog')

// Test ID
page.getByTestId('submit-btn')
```

Best Practice: Prefer user-facing attributes (role, label, text) over implementation details (CSS selectors, XPath).

3. What is the difference between `page.locator()` and `page.$()` or `page.$$()` ?

Answer:

page.locator() (Recommended):

• Returns a Locator object (lazy evaluation)

• Auto-waits for element to be actionable

• Strict mode by default (fails if multiple elements match)

• Supports chaining and filtering

• Better error messages

```
const button = page.locator('button.submit');
await button.click(); // Auto-waits
```

page.$() and page.$$():

• Returns ElementHandle (immediate evaluation)

• No auto-waiting

• Can become stale if DOM changes

• Legacy API from Puppeteer

```
const button = await page.$('button.submit'); // Can be null
if (button) {
  await button.click(); // Manual null check needed
}
```

4. How do you handle multiple browser contexts in Playwright?

Answer:

Browser contexts are isolated browser sessions that don't share cookies, cache, or local storage.

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

// Launch browser
const browser = await chromium.launch();

// Create multiple isolated contexts
const context1 = await browser.newContext();
const context2 = await browser.newContext();

// Each context has its own pages
const page1 = await context1.newPage();
const page2 = await context2.newPage();

// Different sessions - no data sharing
await page1.goto('https://example.com');
await page2.goto('https://example.com');

// Cleanup
await context1.close();
await context2.close();
await browser.close();
```

Use cases:

• Testing multi-user scenarios

• Parallel test execution

• Testing with different permissions/states

• Isolating test data

5. What is auto-waiting in Playwright?

Answer:

Playwright automatically waits for elements to be actionable before performing actions. This eliminates the need for explicit waits in most cases.

Auto-wait checks:

• Element is attached to DOM

• Element is visible

• Element is stable (not animating)

• Element receives events (not obscured)

• Element is enabled (for inputs/buttons)

```
// No explicit wait needed
await page.locator('#button').click();

// Playwright automatically waits for:
```

```
// 1. Button to exist
// 2. Button to be visible
// 3. Button to be enabled
// 4. Button to stop animating
```

Timeout: Default is 30 seconds, configurable:

```
// Set timeout for specific action
await page.locator('#button').click({ timeout: 5000 });

// Set default timeout
page.setDefaultTimeout(60000);
```

Intermediate Questions

6. How do you handle file uploads in Playwright?

Answer:

```
// Single file upload
await page.locator('input[type="file"]').setInputFiles('path/to/file.pdf');

// Multiple files
await page.locator('input[type="file"]').setInputFiles([
  'file1.pdf',
  'file2.jpg'
]);

// Upload from buffer (in-memory)
await page.locator('input[type="file"]').setInputFiles({
  name: 'report.pdf',
  mimeType: 'application/pdf',
  buffer: Buffer.from('file content')
});

// Clear file input
await page.locator('input[type="file"]').setInputFiles([]);

// Handle file chooser dialog
const fileChooserPromise = page.waitForEvent('filechooser');
await page.locator('#upload-btn').click();
const fileChooser = await fileChooserPromise;
await fileChooser.setFiles('file.pdf');
```

7. How do you handle downloads in Playwright?

Answer:

```
// Method 1: Wait for download event
const downloadPromise = page.waitForEvent('download');
await page.locator('#download-link').click();
const download = await downloadPromise;

// Save to specific path
await download.saveAs('/path/to/save/file.pdf');

// Get download info
console.log(download.suggestedFilename());
const path = await download.path(); // temporary path

// Method 2: Handle multiple downloads
page.on('download', async download => {
  const fileName = download.suggestedFilename();
  await download.saveAs(`./downloads/${fileName}`);
});
```

8. How do you handle alerts, prompts, and confirmations?

Answer:

```
// Handle alert - accept
page.on('dialog', async dialog => {
  console.log(dialog.message());
```

```
  await dialog.accept();
});

// Handle prompt - provide input
page.on('dialog', async dialog => {
  if (dialog.type() === 'prompt') {
    await dialog.accept('User Input');
  }
});

// Handle confirmation - dismiss
page.on('dialog', async dialog => {
  if (dialog.type() === 'confirm') {
    await dialog.dismiss();
  }
});

// Conditional handling
page.on('dialog', async dialog => {
  if (dialog.message().includes('Are you sure?')) {
    await dialog.accept();
  } else {
    await dialog.dismiss();
  }
});

// One-time dialog handling
page.once('dialog', dialog => dialog.accept());
await page.locator('#delete-btn').click();
```

9. How do you perform network interception and mocking?

Answer:

```
// Mock API responses
await page.route('**/api/users', async route => {
  await route.fulfill({
    status: 200,
    contentType: 'application/json',
    body: JSON.stringify([
      { id: 1, name: 'John' },
      { id: 2, name: 'Jane' }
    ])
  });
});

// Abort requests (block images, fonts)
await page.route('**/*.{png,jpg,jpeg}', route => route.abort());

// Modify requests
await page.route('**/api/**', async route => {
  const headers = route.request().headers();
  headers['Authorization'] = 'Bearer token123';
  await route.continue({ headers });
});

// Intercept and log network calls
page.on('request', request => {
  console.log('Request:', request.url());
});

page.on('response', response => {
  console.log('Response:', response.url(), response.status());
});

// Wait for specific API call
await page.waitForResponse('**/api/users');
```

10. How do you handle frames and iframes?

Answer:

```
// Get frame by name or URL
const frame = page.frame('frame-name');
const frame = page.frame({ url: /.*frame.html/ });

// Get frame by selector
const frameElement = page.frameLocator('iframe#my-frame');

// Work with frame content
await frameElement.locator('#button').click();

// Get all frames
const frames = page.frames();

// Main frame
const mainFrame = page.mainFrame();

// Wait for frame
await page.waitForSelector('iframe#my-frame');
const frame = page.frame({ name: 'my-frame' });

// Frame locator (recommended for nested frames)
const frameLocator = page.frameLocator('iframe').first();
await frameLocator.locator('#submit').click();
```

11. How do you handle new pages/tabs?

Answer:

```
// Method 1: Wait for new page
const newPagePromise = context.waitForEvent('page');
await page.locator('a[target="_blank"]').click();
const newPage = await newPagePromise;
await newPage.waitForLoadState();
console.log(await newPage.title());

// Method 2: Using page.on event
context.on('page', async page => {
  await page.waitForLoadState();
  console.log('New page:', await page.title());
});

// Get all pages
const pages = context.pages();

// Switch between pages
await pages[0].bringToFront();
await pages[1].bringToFront();

// Close specific page
await newPage.close();
```

12. What are the different wait strategies in Playwright?

Answer:

```
// Wait for element state
await page.locator('#element').waitFor({ state: 'visible' });
await page.locator('#element').waitFor({ state: 'hidden' });
await page.locator('#element').waitFor({ state: 'attached' });
await page.locator('#element').waitFor({ state: 'detached' });
```

```
// Wait for load states
await page.waitForLoadState('load'); // full page load
await page.waitForLoadState('domcontentloaded'); // DOM ready
await page.waitForLoadState('networkidle'); // no network activity

// Wait for selector
await page.waitForSelector('#element');
await page.waitForSelector('#element', { state: 'visible' });

// Wait for function
await page.waitForFunction(() => {
  return document.querySelectorAll('.item').length > 5;
});

// Wait for URL
await page.waitForURL('**/dashboard');
await page.waitForURL(/.*profile.*/);

// Wait for response
await page.waitForResponse('**/api/users');
await page.waitForResponse(response =>
  response.url().includes('/api/') && response.status() === 200
);

// Wait for event
await page.waitForEvent('console');
await page.waitForEvent('download');

// Wait for timeout (use sparingly)
await page.waitForTimeout(3000);
```

Advanced Questions

13. How do you implement Page Object Model (POM) in Playwright?

Answer:

```
// pages/LoginPage.js
class LoginPage {
  constructor(page) {
    this.page = page;
    this.usernameInput = page.getByLabel('Username');
    this.passwordInput = page.getByLabel('Password');
    this.loginButton = page.getByRole('button', { name: 'Login' });
    this.errorMessage = page.locator('.error-message');
  }

  async navigate() {
    await this.page.goto('/login');
  }

  async login(username, password) {
    await this.usernameInput.fill(username);
    await this.passwordInput.fill(password);
    await this.loginButton.click();
  }

  async getErrorMessage() {
    return await this.errorMessage.textContent();
  }

  async isLoggedIn() {
    return await this.page.url().includes('/dashboard');
  }
}

module.exports = { LoginPage };

// test file
const { test, expect } = require('@playwright/test');
const { LoginPage } = require('./pages/LoginPage');

test('user can login', async ({ page }) => {
  const loginPage = new LoginPage(page);
  await loginPage.navigate();
  await loginPage.login('admin', 'password123');
  expect(await loginPage.isLoggedIn()).toBeTruthy();
});
```

14. How do you handle authentication and session management?

Answer:

```
// Method 1: Save authentication state
const { chromium } = require('playwright');

const browser = await chromium.launch();
const context = await browser.newContext();
const page = await context.newPage();

// Perform login
await page.goto('https://example.com/login');
await page.fill('#username', 'admin');
await page.fill('#password', 'password');
await page.click('#login');
```

```
// Save authenticated state
await context.storageState({ path: 'auth.json' });
await browser.close();

// Method 2: Reuse authentication state
const browser = await chromium.launch();
const context = await browser.newContext({
  storageState: 'auth.json'
});
const page = await context.newPage();
await page.goto('https://example.com/dashboard'); // Already logged in

// Method 3: API-based authentication
test.use({
  storageState: async ({}, use) => {
    // Get token via API
    const response = await fetch('https://api.example.com/login', {
      method: 'POST',
      body: JSON.stringify({ user: 'admin', pass: 'password' })
    });
    const { token } = await response.json();

    // Create storage state
    await use({
      cookies: [],
      origins: [
        {
          origin: 'https://example.com',
          localStorage: [
            { name: 'authToken', value: token }
          ]
        }
      ]
    });
  }
});

// Method 4: Setup in beforeEach
test.beforeEach(async ({ page }) => {
  await page.goto('/login');
  await page.fill('#username', 'admin');
  await page.fill('#password', 'password');
  await page.click('#login');
  await page.waitForURL('**/dashboard');
});
```

15. How do you implement custom fixtures in Playwright?

Answer:

```
// fixtures.js
const base = require('@playwright/test');

exports.test = base.test.extend({
  // Custom page fixture with logged-in state
  authenticatedPage: async ({ page }, use) => {
    await page.goto('/login');
    await page.fill('#username', 'testuser');
    await page.fill('#password', 'password');
    await page.click('#login');
    await page.waitForURL('**/dashboard');
    await use(page);
  },

  // Custom API client fixture
  apiClient: async ({}, use) => {
    const client = {
      async get(url) {
        const response = await fetch(url);
        return response.json();
      },
```

```
      async post(url, data) {
        const response = await fetch(url, {
          method: 'POST',
          headers: { 'Content-Type': 'application/json' },
          body: JSON.stringify(data)
        });
        return response.json();
      }
    };
    await use(client);
  },

  // Database fixture
  db: async ({}, use) => {
    // Setup
    const db = await connectDatabase();
    await db.query('TRUNCATE users');

    // Use
    await use(db);

    // Teardown
    await db.close();
  }
});

// test file
const { test, expect } = require('./fixtures');

test('user can view dashboard', async ({ authenticatedPage }) => {
  // Page is already authenticated
  await expect(authenticatedPage.locator('h1')).toHaveText('Dashboard');
});

test('API returns correct data', async ({ apiClient }) => {
  const data = await apiClient.get('/api/users');
  expect(data).toHaveLength(5);
});
```

16. How do you implement parallel testing and test sharding?

Answer:

```
// playwright.config.js
module.exports = {
  // Run tests in parallel
  fullyParallel: true,

  // Number of workers
  workers: process.env.CI ? 2 : undefined, // 2 in CI, all cores locally

  // Projects for different browsers
  projects: [
    { name: 'chromium', use: { browserName: 'chromium' } },
    { name: 'firefox', use: { browserName: 'firefox' } },
    { name: 'webkit', use: { browserName: 'webkit' } }
  ]
};

// Run specific number of workers
// npx playwright test --workers=4

// Test sharding (split tests across machines)
// Machine 1:
// npx playwright test --shard=1/3

// Machine 2:
// npx playwright test --shard=2/3

// Machine 3:
// npx playwright test --shard=3/3
```

```
// Disable parallel for specific test
test.describe.configure({ mode: 'serial' });
test.describe('checkout flow', () => {
  test('add to cart', async ({ page }) => {});
  test('proceed to checkout', async ({ page }) => {});
  test('complete payment', async ({ page }) => {});
});

// Serial mode for file
test.describe.serial('dependent tests', () => {
  let orderId;

  test('create order', async ({ page }) => {
    // Test creates order
    orderId = '12345';
  });

  test('verify order', async ({ page }) => {
    // This runs only if previous test passed
    await page.goto(`/orders/${orderId}`);
  });
});
```

17. How do you implement visual regression testing?

Answer:

```
const { test, expect } = require('@playwright/test');

test('visual regression - homepage', async ({ page }) => {
  await page.goto('/');

  // Full page screenshot
  await expect(page).toHaveScreenshot('homepage.png');

  // Element screenshot
  await expect(page.locator('.header')).toHaveScreenshot('header.png');

  // With custom threshold
  await expect(page).toHaveScreenshot('homepage.png', {
    maxDiffPixels: 100 // Allow 100 pixels difference
  });

  // With custom threshold percentage
  await expect(page).toHaveScreenshot('homepage.png', {
    maxDiffPixelRatio: 0.2 // Allow 20% difference
  });

  // Mask dynamic content
  await expect(page).toHaveScreenshot('homepage.png', {
    mask: [page.locator('.timestamp'), page.locator('.ad-banner')]
  });

  // Update snapshots
  // npx playwright test --update-snapshots
});

// Advanced: Custom visual testing
test('custom visual test', async ({ page }) => {
  await page.goto('/dashboard');

  // Take screenshot
  const screenshot = await page.screenshot();

  // Compare using external library (e.g., pixelmatch)
  const diff = compareImages(screenshot, baselineImage);
  expect(diff.percentDifference).toBeLessThan(5);
});

// Cross-browser visual testing
test('visual consistency across browsers', async ({ page, browserName }) => {
```

```
  await page.goto('/');
  await expect(page).toHaveScreenshot(`homepage-${browserName}.png`);
});
```

18. How do you debug tests in Playwright?

Answer:

```
// Method 1: Playwright Inspector
// npx playwright test --debug

// Method 2: Debug specific test
// npx playwright test example.spec.js:10 --debug

// Method 3: Headed mode
// npx playwright test --headed

// Method 4: Slow motion
const browser = await chromium.launch({
  headless: false,
  slowMo: 1000 // 1 second delay between actions
});

// Method 5: Pause execution
test('debug test', async ({ page }) => {
  await page.goto('/');
  await page.pause(); // Opens inspector
  await page.click('#button');
});

// Method 6: Screenshots on failure
// playwright.config.js
module.exports = {
  use: {
    screenshot: 'only-on-failure',
    video: 'retain-on-failure',
    trace: 'retain-on-failure'
  }
};

// Method 7: Console logs
page.on('console', msg => console.log('Browser log:', msg.text()));
page.on('pageerror', error => console.log('Page error:', error));

// Method 8: Trace viewer
// playwright.config.js
module.exports = {
  use: {
    trace: 'on-first-retry'
  }
};
// View trace: npx playwright show-trace trace.zip

// Method 9: VS Code debugger
// Add breakpoint in VS Code
// Run: Debug Test in VS Code

// Method 10: Verbose logging
// npx playwright test --reporter=list --verbose
```

Scenario-Based Questions

19. How would you test a dynamic table with infinite scrolling?

Answer:

```
test('infinite scroll table', async ({ page }) => {
  await page.goto('/users');

  let previousRowCount = 0;
  let currentRowCount = 0;
  let scrollAttempts = 0;
  const maxScrolls = 10;

  while (scrollAttempts < maxScrolls) {
    // Get current row count
    currentRowCount = await page.locator('table tbody tr').count();

    // If no new rows loaded, we've reached the end
    if (currentRowCount === previousRowCount) {
      break;
    }

    // Scroll to last row
    const lastRow = page.locator('table tbody tr').last();
    await lastRow.scrollIntoViewIfNeeded();

    // Wait for new content
    await page.waitForTimeout(1000); // Or better: wait for network idle

    previousRowCount = currentRowCount;
    scrollAttempts++;
  }

  console.log(`Total rows loaded: ${currentRowCount}`);

  // Verify specific row
  const rows = page.locator('table tbody tr');
  const fifthRow = rows.nth(4);
  await expect(fifthRow).toBeVisible();
});

// Alternative: Wait for network response
test('infinite scroll with network wait', async ({ page }) => {
  await page.goto('/users');

  for (let i = 0; i < 5; i++) {
    const responsePromise = page.waitForResponse('**/api/users*');

    // Scroll to bottom
    await page.evaluate(() => window.scrollTo(0, document.body.scrollHeight));

    // Wait for API response
    await responsePromise;
  }
});
```

20. How would you handle a multi-step form with validation?

Answer:

```
test('multi-step form submission', async ({ page }) => {
  await page.goto('/signup');

  // Step 1: Personal Information
```

```
  await page.getByLabel('First Name').fill('John');
  await page.getByLabel('Last Name').fill('Doe');
  await page.getByLabel('Email').fill('john@example.com');

  // Test validation
  await page.getByRole('button', { name: 'Next' }).click();
  await expect(page.locator('.error')).toHaveCount(0); // No errors

  // Step 2: Address
  await page.getByLabel('Street').fill('123 Main St');
  await page.getByLabel('City').fill('New York');
  await page.getByLabel('Zip').fill('10001');

  // Test back navigation
  await page.getByRole('button', { name: 'Back' }).click();
  await expect(page.getByLabel('First Name')).toHaveValue('John'); // Data persisted

  // Continue forward
  await page.getByRole('button', { name: 'Next' }).click();
  await page.getByRole('button', { name: 'Next' }).click();

  // Step 3: Review and Submit
  await expect(page.locator('.review-name')).toHaveText('John Doe');
  await expect(page.locator('.review-email')).toHaveText('john@example.com');

  // Submit
  const responsePromise = page.waitForResponse('**/api/signup');
  await page.getByRole('button', { name: 'Submit' }).click();
  const response = await responsePromise;

  expect(response.status()).toBe(201);
  await expect(page).toHaveURL('**/success');
});

// Test field validation
test('form validation errors', async ({ page }) => {
  await page.goto('/signup');

  // Try to proceed without filling required fields
  await page.getByRole('button', { name: 'Next' }).click();

  await expect(page.locator('.error-firstname')).toBeVisible();
  await expect(page.locator('.error-email')).toBeVisible();

  // Fill with invalid email
  await page.getByLabel('Email').fill('invalid-email');
  await page.getByRole('button', { name: 'Next' }).click();

  await expect(page.locator('.error-email')).toContainText('valid email');
});
```

21. How would you test a drag-and-drop Kanban board?

Answer:

```
test('drag and drop task between columns', async ({ page }) => {
  await page.goto('/kanban');

  // Verify initial state
  const todoColumn = page.locator('[data-column="todo"]');
  const inProgressColumn = page.locator('[data-column="in-progress"]');

  const task = todoColumn.locator('[data-task="task-1"]');
  await expect(task).toBeVisible();

  // Drag task from TODO to IN PROGRESS
  await task.dragTo(inProgressColumn);

  // Verify task moved
  await expect(todoColumn.locator('[data-task="task-1"]')).toHaveCount(0);
  await expect(inProgressColumn.locator('[data-task="task-1"]')).toBeVisible();
```

```
  // Verify API call was made
  const response = await page.waitForResponse('**/api/tasks/*/move');
  expect(response.status()).toBe(200);

  // Verify task order
  const tasks = await inProgressColumn.locator('[data-task]').all();
  const taskIds = await Promise.all(
    tasks.map(t => t.getAttribute('data-task'))
  );
  expect(taskIds).toEqual(['task-5', 'task-1', 'task-6']);
});

// Alternative: Using mouse
test('drag and drop with mouse coordinates', async ({ page }) => {
  await page.goto('/kanban');

  const source = page.locator('[data-task="task-1"]');
  const target = page.locator('[data-column="done"]');

  // Get bounding boxes
  const sourceBox = await source.boundingBox();
  const targetBox = await target.boundingBox();

  // Perform drag
  await page.mouse.move(
    sourceBox.x + sourceBox.width / 2,
    sourceBox.y + sourceBox.height / 2
  );
  await page.mouse.down();
  await page.mouse.move(
    targetBox.x + targetBox.width / 2,
    targetBox.y + targetBox.height / 2
  );
  await page.mouse.up();

  // Verify
  await expect(target.locator('[data-task="task-1"]')).toBeVisible();
});
```

22. How would you test a real-time chat application?

Answer:

```
test('real-time chat messaging', async ({ browser }) => {
  // Create two browser contexts (two users)
  const context1 = await browser.newContext();
  const context2 = await browser.newContext();

  const page1 = await context1.newPage();
  const page2 = await context2.newPage();

  // User 1 login
  await page1.goto('/chat');
  await page1.fill('#username', 'Alice');
  await page1.click('#join');

  // User 2 login
  await page2.goto('/chat');
  await page2.fill('#username', 'Bob');
  await page2.click('#join');

  // User 1 sends message
  await page1.fill('#message-input', 'Hello Bob!');
  await page1.click('#send');

  // Verify message appears for both users
  await expect(page1.locator('.message').last()).toContainText('Hello Bob!');
  await expect(page2.locator('.message').last()).toContainText('Hello Bob!');
  await expect(page2.locator('.message').last()).toContainText('Alice');

  // User 2 replies
```

```
  await page2.fill('#message-input', 'Hi Alice!');
  await page2.click('#send');

  // Verify reply
  await expect(page1.locator('.message').last()).toContainText('Hi Alice!');
  await expect(page2.locator('.message').last()).toContainText('Hi Alice!');

  // Test typing indicator
  await page1.fill('#message-input', 'Typing...');
  await expect(page2.locator('.typing-indicator')).toContainText('Alice is typing');

  // Cleanup
  await context1.close();
  await context2.close();
});

// Test WebSocket connection
test('WebSocket connection and reconnection', async ({ page }) => {
  await page.goto('/chat');

  // Monitor WebSocket
  page.on('websocket', ws => {
    console.log('WebSocket opened:', ws.url());
    ws.on('close', () => console.log('WebSocket closed'));
    ws.on('framereceived', event => {
      console.log('Received:', event.payload);
    });
  });

  // Simulate network disruption
  await page.context().setOffline(true);
  await expect(page.locator('.disconnected-banner')).toBeVisible();

  // Restore connection
  await page.context().setOffline(false);
  await expect(page.locator('.connected-banner')).toBeVisible();
});
```

Best Practices

23. What are Playwright testing best practices?

Answer:

1. Use User-Facing Locators

```
// Good
await page.getByRole('button', { name: 'Submit' });
await page.getByLabel('Email');
await page.getByText('Welcome');

// Avoid
await page.locator('#btn-submit-form-id-123');
await page.locator('div > form > button:nth-child(3)');
```

2. Leverage Auto-Waiting

```
// Good - auto-waits
await page.locator('#button').click();

// Avoid - unnecessary explicit waits
await page.waitForTimeout(3000);
await page.locator('#button').click();
```

3. Use Page Object Model

```
// Encapsulate page logic
class LoginPage {
  constructor(page) {
    this.page = page;
  }

  async login(user, pass) {
    await this.page.getByLabel('Username').fill(user);
    await this.page.getByLabel('Password').fill(pass);
    await this.page.getByRole('button', { name: 'Login' }).click();
  }
}
```

4. Parallelize Tests

```
// playwright.config.js
module.exports = {
  fullyParallel: true,
  workers: 4
};
```

5. Use Fixtures for Setup

```
test.beforeEach(async ({ page }) => {
  await page.goto('/');
});
```

6. Implement Proper Assertions

```
// Good
await expect(page.locator('h1')).toHaveText('Dashboard');
await expect(page).toHaveURL(/dashboard/);

// Avoid
```

```
const text = await page.locator('h1').textContent();
expect(text).toBe('Dashboard'); // No auto-retry
```

7. Handle Network Properly

```
// Wait for specific API calls
await Promise.all([
  page.waitForResponse('**/api/users'),
  page.click('#load-users')
]);
```

8. Use Descriptive Test Names

```
// Good
test('user can successfully complete checkout with valid payment', async ({ page }) => {});

// Avoid
test('test1', async ({ page }) => {});
```

9. Clean Test Data

```
test.afterEach(async ({ page }) => {
  // Clean up test data
  await page.request.delete('/api/test-data');
});
```

10. Use Trace on Failures

```
// playwright.config.js
module.exports = {
  use: {
    trace: 'on-first-retry',
    screenshot: 'only-on-failure',
    video: 'retain-on-failure'
  }
};
```

Common Pitfalls to Avoid

Don't use fixed timeouts - Use auto-waiting instead

Avoid element handles - Use locators for better reliability

Don't ignore network state - Wait for API responses when needed

Avoid brittle selectors - Use semantic selectors (role, label, text)

Don't skip error handling - Handle expected failures gracefully

Avoid test interdependence - Each test should be independent

Don't hardcode test data - Use fixtures or data generators

Avoid excessive mocking - Test against real APIs when possible

Don't ignore accessibility - Use role-based selectors for better tests

Avoid skipping CI/CD integration - Run tests in pipeline