← Advanced Playwright

Playwright API Testing →

Playwright - Advanced Scenarios

Table of Contents

---

page.on() Event Handling

Understanding Event Listeners

Playwright provides event listeners through `page.on()` to monitor and react to various browser events in real-time.

Common Events

Available Events:

- `request` - Network requests
- `response` - Network responses
- `requestfailed` - Failed requests
- `requestfinished` - Completed requests
- `pageerror` - Uncaught exceptions
- `crash` - Page crash
- `close` - Page close
- `load` - Page load
- `domcontentloaded` - DOM ready
- `popup` - New popup/tab

Console Event Handling

Listen to console messages:

```
import { test, expect } from '@playwright/test';

test('capture console logs', async ({ page }) => {
  const logs: Array<{ type: string; text: string }> = [];

  // Listen to console events
  page.on('console', (msg) => {
    logs.push({
      type: msg.type(),
      text: msg.text(),
    });
  });

  await page.goto('https://example.com');

  // Trigger some console logs
  await page.evaluate(() => {
    console.log('This is a log');
    console.warn('This is a warning');
    console.error('This is an error');
  });

  // Verify logs were captured
```

```
  expect(logs.some(log => log.type === 'error')).toBe(true);

  // Print all logs
  console.log('Captured console messages:');
  logs.forEach(log => console.log(`[${log.type}] ${log.text}`));
});
```

Filter console messages:

```
test('filter console errors', async ({ page }) => {
  const errors: string[] = [];

  page.on('console', (msg) => {
    if (msg.type() === 'error') {
      errors.push(msg.text());
    }
  });

  await page.goto('https://example.com');

  // If page has console errors, test will capture them
  if (errors.length > 0) {
    console.log('Console errors found:');
    errors.forEach(error => console.error(error));
  }

  // Assert no errors
  expect(errors.length).toBe(0);
});
```

Complex console object handling:

```
test('handle complex console objects', async ({ page }) => {
  page.on('console', async (msg) => {
    // Get console message arguments
    const args = msg.args();

    // Process each argument
    for (const arg of args) {
      const value = await arg.jsonValue();
      console.log('Console value:', value);
    }
```

```
  await page.evaluate(() => {
    console.log({ name: 'Test', age: 30 });
    console.log(['item1', 'item2', 'item3']);
  });
});
```

Page Error Handling

Capture page errors:

```
test('capture page errors', async ({ page }) => {
  const pageErrors: Error[] = [];

  page.on('pageerror', (error) => {
    pageErrors.push(error);
    console.error('Page error:', error.message);
    console.error('Stack:', error.stack);
  });

  await page.goto('https://example.com');

  // Trigger an error
  await page.evaluate(() => {
    // @ts-ignore
    undefinedFunction(); // This will cause an error
  });

  // Wait a bit for error to be captured
  await page.waitForTimeout(1000);

  // Verify error was captured
  expect(pageErrors.length).toBeGreaterThan(0);
  expect(pageErrors[0].message).toContain('undefinedFunction');
});
```

Fail test on page errors:

```
test('fail on page errors', async ({ page }) => {
  let hasError = false;
  let errorMessage = '';
```

```
    errorMessage = error.message;
  });

  await page.goto('https://example.com');

  // Your test actions...

  // Assert no page errors occurred
  if (hasError) {
    throw new Error(`Page error occurred: ${errorMessage}`);
  }
});
```

Request Event Handling

Monitor all requests:

```
test('monitor all requests', async ({ page }) => {
  const requests: Array<{ url: string; method: string }> = [];

  page.on('request', (request) => {
    requests.push({
      url: request.url(),
      method: request.method(),
    });
  });

  await page.goto('https://example.com');

  console.log(`Total requests: ${requests.length}`);

  // Filter requests
  const apiRequests = requests.filter(r => r.url.includes('/api/'));
  console.log(`API requests: ${apiRequests.length}`);

  const postRequests = requests.filter(r => r.method === 'POST');
  console.log(`POST requests: ${postRequests.length}`);
});
```

Log request details:

```
    console.log('Headers:', request.headers());

    // Log POST data
    if (request.method() === 'POST') {
      console.log('POST data:', request.postData());
    }
  });

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

  await page.goto('https://example.com');
});
```

Response Event Handling

Monitor responses:

```
test('monitor responses', async ({ page }) => {
  const responses: Array<{
    url: string;
    status: number;
    statusText: string
  }> = [];

  page.on('response', (response) => {
    responses.push({
      url: response.url(),
      status: response.status(),
      statusText: response.statusText(),
    });
  });

  await page.goto('https://example.com');

  // Check for failed responses
  const failedResponses = responses.filter(r => r.status >= 400);

  if (failedResponses.length > 0) {
    console.log('Failed responses:');
    failedResponses.forEach(r => {
```

```
  // Assert no server errors
  const serverErrors = responses.filter(r => r.status >= 500);
  expect(serverErrors.length).toBe(0);
});
```

Get response body:

```
test('read response body', async ({ page }) => {
  page.on('response', async (response) => {
    if (response.url().includes('/api/users')) {
      const body = await response.json();
      console.log('Users API response:', body);

      // Verify response structure
      expect(body).toHaveProperty('data');
      expect(Array.isArray(body.data)).toBe(true);
    }
  });

  await page.goto('https://example.com/users');
});
```

Failed Request Handling

Track failed requests:

```
test('track failed requests', async ({ page }) => {
  const failedRequests: Array<{
    url: string;
    failure: string;
  }> = [];

  page.on('requestfailed', (request) => {
    failedRequests.push({
      url: request.url(),
      failure: request.failure()?.errorText || 'Unknown',
    });
  });

  await page.goto('https://example.com');
```

```
    failedRequests.forEach(r => {
      console.log(`${r.url} - ${r.failure}`);
    });
  }

  // Assert no critical requests failed
  const criticalFailed = failedRequests.filter(r =>
    r.url.includes('/api/critical')
  );
  expect(criticalFailed.length).toBe(0);
});
```

Combining Multiple Events

Comprehensive monitoring:

```
test('comprehensive event monitoring', async ({ page }) => {
  const monitoring = {
    requests: [] as string[],
    responses: [] as { url: string; status: number }[],
    consoleLogs: [] as string[],
    errors: [] as string[],
  };

  page.on('request', (request) => {
    monitoring.requests.push(request.url());
  });

  page.on('response', (response) => {
    monitoring.responses.push({
      url: response.url(),
      status: response.status(),
    });
  });

  page.on('console', (msg) => {
    if (msg.type() === 'log') {
      monitoring.consoleLogs.push(msg.text());
    }
  });

  page.on('pageerror', (error) => {
    monitoring.errors.push(error.message);
```

```
  // Generate report
  console.log('\n=== Test Monitoring Report ===');
  console.log(`Total Requests: ${monitoring.requests.length}`);
  console.log(`Total Responses: ${monitoring.responses.length}`);
  console.log(`Console Logs: ${monitoring.consoleLogs.length}`);
  console.log(`Errors: ${monitoring.errors.length}`);

  const failedResponses = monitoring.responses.filter(r => r.status >= 400
  console.log(`Failed Responses: ${failedResponses.length}`);

  // Assert quality gates
  expect(monitoring.errors.length).toBe(0);
  expect(failedResponses.filter(r => r.status >= 500).length).toBe(0);
});
```

Network Monitoring and Request/Response Interception

Waiting for Specific Requests/Responses

Wait for API response:

```
test('wait for API response', async ({ page }) => {
  await page.goto('https://example.com');

  // Wait for specific API call
  const [response] = await Promise.all([
    page.waitForResponse('**/api/users'),
    page.click('#load-users-button'),
  ]);

  expect(response.status()).toBe(200);

  const body = await response.json();
  expect(body.users.length).toBeGreaterThan(0);
});
```

```
test('wait for specific response', async ({ page }) => {
  await page.goto('https://example.com');

  const [response] = await Promise.all([
    page.waitForResponse(
      (response) =>
        response.url().includes('/api/') &&
        response.status() === 200
    ),
    page.click('button'),
  ]);

  console.log('Matched response:', response.url());
});
```

Wait for multiple responses:

```
test('wait for multiple API calls', async ({ page }) => {
  await page.goto('https://example.com');

  const [usersResponse, productsResponse] = await Promise.all([
    page.waitForResponse('**/api/users'),
    page.waitForResponse('**/api/products'),
    page.click('#load-data'),
  ]);

  expect(usersResponse.status()).toBe(200);
  expect(productsResponse.status()).toBe(200);
});
```

Request Interception and Mocking

Abort requests:

```
test('block images', async ({ page }) => {
  // Block image requests
  await page.route('**/*.{png,jpg,jpeg,gif,svg}', (route) => {
    route.abort();
  });

  await page.goto('https://example.com');
```

Block specific domains:

```
test('block third-party scripts', async ({ page }) => {
  // Block analytics and ads
  await page.route('**/*', (route) => {
    const url = route.request().url();

    if (
      url.includes('google-analytics.com') ||
      url.includes('facebook.com') ||
      url.includes('doubleclick.net')
    ) {
      route.abort();
    } else {
      route.continue();
    }
  });

  await page.goto('https://example.com');
});
```

Mock API responses:

```
test('mock API response', async ({ page }) => {
  // Intercept and mock API call
  await page.route('**/api/users', (route) => {
    route.fulfill({
      status: 200,
      contentType: 'application/json',
      body: JSON.stringify({
        users: [
          { id: 1, name: 'Mock User 1', email: 'mock1@example.com' },
          { id: 2, name: 'Mock User 2', email: 'mock2@example.com' },
        ],
      }),
    });
  });

  await page.goto('https://example.com/users');

  // Verify mocked data is displayed
```

Mock error responses:

```
test('mock API error', async ({ page }) => {
  await page.route('**/api/users', (route) => {
    route.fulfill({
      status: 500,
      contentType: 'application/json',
      body: JSON.stringify({
        error: 'Internal Server Error',
      }),
    });
  });

  await page.goto('https://example.com/users');

  // Verify error handling
  await expect(page.getByText('Failed to load users')).toBeVisible();
});
```

Modify request headers:

```
test('modify request headers', async ({ page }) => {
  await page.route('**/api/**', (route) => {
    const headers = {
      ...route.request().headers(),
      'X-Custom-Header': 'test-value',
      'Authorization': 'Bearer mock-token',
    };

    route.continue({ headers });
  });

  await page.goto('https://example.com');
});
```

Modify response:

```
test('modify response', async ({ page }) => {
  await page.route('**/api/config', async (route) => {
```

```
    // Modify response
    body.featureFlag = true;
    body.maxUsers = 1000;

    // Return modified response
    route.fulfill({
      response,
      body: JSON.stringify(body),
    });
  });

  await page.goto('https://example.com');

  // App uses modified config
});
```

Network Conditions Testing

Simulate slow network:

```
test('test with slow network', async ({ page, context }) => {
  // Simulate slow 3G
  await context.route('**/*', async (route) => {
    await new Promise(resolve => setTimeout(resolve, 2000)); // 2s delay
    await route.continue();
  });

  const startTime = Date.now();
  await page.goto('https://example.com');
  const loadTime = Date.now() - startTime;

  console.log(`Page loaded in ${loadTime}ms with slow network`);
  expect(loadTime).toBeGreaterThan(2000);
});
```

Simulate offline mode:

```
test('test offline behavior', async ({ page, context }) => {
  await page.goto('https://example.com');
```

```
  // Try to navigate (should fail or show offline message)
  await page.click('a[href="/other-page"]');

  // Verify offline handling
  await expect(page.getByText(/offline|no internet/i)).toBeVisible();

  // Go back online
  await context.setOffline(false);
});
```

Request Timing Analysis

Measure request timing:

```
test('analyze request timing', async ({ page }) => {
  const timings: Array<{
    url: string;
    duration: number;
  }> = [];

  page.on('requestfinished', async (request) => {
    const response = await request.response();
    if (response) {
      const timing = response.timing();
      timings.push({
        url: request.url(),
        duration: timing.responseEnd,
      });
    }
  });

  await page.goto('https://example.com');

  // Find slowest requests
  const sorted = timings.sort((a, b) => b.duration - a.duration);

  console.log('Top 5 slowest requests:');
  sorted.slice(0, 5).forEach((item, index) => {
    console.log(`${index + 1}. ${item.duration}ms - ${item.url}`);
  });

  // Assert performance
  const apiRequests = timings.filter(t => t.url.includes('/api/'));
```

```
});
```

Handling iframes and Shadow DOM

Working with iframes

Locate iframe:

```
test('access iframe content', async ({ page }) => {
  await page.goto('https://example.com/page-with-iframe');

  // Method 1: By frame locator
  const frame = page.frameLocator('#my-iframe');
  await frame.locator('input#username').fill('testuser');
  await frame.locator('button').click();

  // Method 2: By frame name
  const frameByName = page.frame({ name: 'payment-frame' });
  if (frameByName) {
    await frameByName.fill('input#card', '4111111111111111');
  }

  // Method 3: By URL
  const frameByUrl = page.frame({ url: /payment\.example\.com/ });
  if (frameByUrl) {
    await frameByUrl.click('button#submit');
  }
});
```

Wait for iframe to load:

```
test('wait for iframe', async ({ page }) => {
  await page.goto('https://example.com');

  // Wait for iframe to be attached
  const frameLocator = page.frameLocator('#dynamic-iframe');
```

```
  // Interact with iframe content
  await frameLocator.locator('input').fill('text');
  await frameLocator.locator('button').click();
});
```

Nested iframes:

```
test('nested iframes', async ({ page }) => {
  await page.goto('https://example.com');

  // Access nested iframe
  const outerFrame = page.frameLocator('#outer-frame');
  const innerFrame = outerFrame.frameLocator('#inner-frame');

  await innerFrame.locator('input').fill('text in nested iframe');
  await innerFrame.locator('button').click();
});
```

Get all frames:

```
test('list all frames', async ({ page }) => {
  await page.goto('https://example.com');

  const frames = page.frames();
  console.log(`Total frames: ${frames.length}`);

  frames.forEach((frame, index) => {
    console.log(`Frame ${index}: ${frame.url()}`);
    console.log(`Frame name: ${frame.name()}`);
  });
});
```

Complete iframe example:

```
test('payment in iframe', async ({ page }) => {
  await page.goto('https://example.com/checkout');

  // Fill main form
  await page.fill('#email', 'user@example.com');
  await page.fill('#name', 'John Doe');
```

```
  // Fill payment details in iframe
  await paymentFrame.locator('#card-number').fill('4111111111111111');
  await paymentFrame.locator('#expiry').fill('12/25');
  await paymentFrame.locator('#cvv').fill('123');

  // Submit from iframe
  await paymentFrame.locator('button[type="submit"]').click();

  // Verify success on main page
  await expect(page.getByText('Payment successful')).toBeVisible();
});
```

Working with Shadow DOM

Access Shadow DOM elements:

```
test('access shadow DOM', async ({ page }) => {
  await page.goto('https://example.com/custom-components');

  // Method 1: Using >> (deprecated but still works)
  await page.locator('custom-button >> button').click();

  // Method 2: Using piercing selector (recommended)
  await page.locator('custom-button').locator('button').click();

  // Method 3: Explicit shadow root access
  const customElement = await page.locator('custom-button').elementHandle(
  const shadowRoot = await customElement?.evaluateHandle(
    (element) => element.shadowRoot
  );
  const button = await shadowRoot?.asElement()?.waitForSelector('button');
  await button?.click();
});
```

Deep shadow DOM navigation:

```
test('deep shadow DOM', async ({ page }) => {
  await page.goto('https://example.com');

  // Access nested shadow DOM
```

```
  await button.click();
});
```

Get shadow DOM content:

```
test('extract shadow DOM content', async ({ page }) => {
  await page.goto('https://example.com');

  const text = await page.evaluate(() => {
    const element = document.querySelector('custom-element');
    const shadowRoot = element?.shadowRoot;
    const content = shadowRoot?.querySelector('.content');
    return content?.textContent;
  });

  console.log('Shadow DOM content:', text);
  expect(text).toBeDefined();
});
```

File Upload and Download Automation

File Upload

Simple file upload:

```
test('upload single file', async ({ page }) => {
  await page.goto('https://example.com/upload');

  // Select file
  await page.setInputFiles('input[type="file"]', 'test-files/document.pdf'

  // Submit
  await page.click('button[type="submit"]');

  // Verify upload success
  await expect(page.getByText('Upload successful')).toBeVisible();
```

Multiple file upload:

```
test('upload multiple files', async ({ page }) => {
  await page.goto('https://example.com/upload');

  // Upload multiple files
  await page.setInputFiles('input[type="file"]', [
    'test-files/file1.pdf',
    'test-files/file2.pdf',
    'test-files/file3.pdf',
  ]);

  await page.click('button[type="submit"]');

  // Verify all files uploaded
  await expect(page.getByText('3 files uploaded')).toBeVisible();
});
```

Upload from buffer:

```
test('upload from buffer', async ({ page }) => {
  await page.goto('https://example.com/upload');

  // Create file from buffer
  const fileContent = 'This is test file content';
  const buffer = Buffer.from(fileContent);

  await page.setInputFiles('input[type="file"]', {
    name: 'test-file.txt',
    mimeType: 'text/plain',
    buffer: buffer,
  });

  await page.click('button[type="submit"]');
});
```

Handle file chooser dialog:

```
test('file chooser dialog', async ({ page }) => {
  await page.goto('https://example.com/upload');
```

```
    page.click('#upload-button'), // Opens file chooser
  ]);

  // Set files
  await fileChooser.setFiles('test-files/document.pdf');

  // Verify file selected
  await expect(page.getByText('document.pdf')).toBeVisible();
});
```

Drag and drop upload:

```
test('drag and drop file upload', async ({ page }) => {
  await page.goto('https://example.com/upload');

  // Create DataTransfer object
  const dataTransfer = await page.evaluateHandle(() => {
    const dt = new DataTransfer();
    return dt;
  });

  // Read file
  const fileContent = await page.evaluate(async () => {
    const response = await fetch('test-files/document.pdf');
    const buffer = await response.arrayBuffer();
    return Array.from(new Uint8Array(buffer));
  });

  // Create File in browser
  await page.evaluate(
    ({ dataTransfer, fileContent }) => {
      const file = new File([new Uint8Array(fileContent)], 'document.pdf',
        type: 'application/pdf',
      });
      dataTransfer.items.add(file);
    },
    { dataTransfer, fileContent }
  );

  // Trigger drop event
  const dropZone = await page.locator('.drop-zone');
  await dropZone.dispatchEvent('drop', { dataTransfer });
});
```

```
test('clear file input', async ({ page }) => {
  await page.goto('https://example.com/upload');

  // Upload file
  await page.setInputFiles('input[type="file"]', 'test-files/document.pdf'

  // Verify file selected
  await expect(page.getByText('document.pdf')).toBeVisible();

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

  // Verify no file selected
  await expect(page.getByText('document.pdf')).not.toBeVisible();
});
```

File Download

Handle downloads:

```
test('download file', async ({ page }) => {
  await page.goto('https://example.com/downloads');

  // Wait for download
  const [download] = await Promise.all([
    page.waitForEvent('download'),
    page.click('a[download]'), // Click download link
  ]);

  // Get download info
  const fileName = download.suggestedFilename();
  console.log('Downloaded file:', fileName);

  // Save to specific path
  await download.saveAs(`downloads/${fileName}`);

  // Verify download
  const path = await download.path();
  expect(path).toBeTruthy();
});
```

Verify downloaded file content:

```
  await page.goto('https://example.com/downloads');

  const [download] = await Promise.all([
    page.waitForEvent('download'),
    page.click('#download-report'),
  ]);

  const path = await download.path();
  expect(path).toBeTruthy();

  // Read file content
  const content = fs.readFileSync(path!, 'utf8');

  // Verify content
  expect(content).toContain('Report Data');
  expect(content.length).toBeGreaterThan(0);
});
```

Download with custom name:

```
test('download with custom name', async ({ page }) => {
  await page.goto('https://example.com/downloads');

  const [download] = await Promise.all([
    page.waitForEvent('download'),
    page.click('#download-button'),
  ]);

  // Save with custom name
  const customPath = `downloads/custom-${Date.now()}.pdf`;
  await download.saveAs(customPath);

  console.log('Saved as:', customPath);
});
```

Handle multiple downloads:

```
test('multiple downloads', async ({ page }) => {
  await page.goto('https://example.com/downloads');

  const downloads: Download[] = [];
```

```
    downloads.push(download);
  });

  // Trigger multiple downloads
  await page.click('#download-all');

  // Wait for all downloads to start
  await page.waitForTimeout(2000);

  // Save all downloads
  for (const download of downloads) {
    const fileName = download.suggestedFilename();
    await download.saveAs(`downloads/${fileName}`);
    console.log('Downloaded:', fileName);
  }

  expect(downloads.length).toBeGreaterThan(0);
});
```

PDF download and verification:

```
test('download and verify PDF', async ({ page }) => {
  await page.goto('https://example.com/reports');

  const [download] = await Promise.all([
    page.waitForEvent('download'),
    page.click('#generate-pdf'),
  ]);

  const path = await download.path();
  expect(path).toBeTruthy();

  // Verify PDF file
  const stats = fs.statSync(path!);
  expect(stats.size).toBeGreaterThan(1000); // At least 1KB

  // Check file extension
  const fileName = download.suggestedFilename();
  expect(fileName).toMatch(/\.pdf$/);
});
```

JavaScript Dialogs

Handle alert dialog:

```
test('handle alert', async ({ page }) => {
  await page.goto('https://example.com');

  // Accept dialog automatically
  page.on('dialog', async (dialog) => {
    console.log('Alert message:', dialog.message());
    expect(dialog.type()).toBe('alert');
    await dialog.accept();
  });

  // Trigger alert
  await page.click('#show-alert');

  // Continue after alert is dismissed
  await expect(page.getByText('Alert dismissed')).toBeVisible();
});
```

Handle confirm dialog:

```
test('handle confirm - accept', async ({ page }) => {
  await page.goto('https://example.com');

  page.on('dialog', async (dialog) => {
    console.log('Confirm message:', dialog.message());
    expect(dialog.type()).toBe('confirm');
    await dialog.accept(); // Click OK
  });

  await page.click('#show-confirm');
  await expect(page.getByText('Confirmed')).toBeVisible();
});

test('handle confirm - dismiss', async ({ page }) => {
  await page.goto('https://example.com');

  page.on('dialog', async (dialog) => {
    await dialog.dismiss(); // Click Cancel
```

```
  await expect(page.getByText('Cancelled')).toBeVisible();
});
```

Handle prompt dialog:

```
test('handle prompt', async ({ page }) => {
  await page.goto('https://example.com');

  page.on('dialog', async (dialog) => {
    expect(dialog.type()).toBe('prompt');
    expect(dialog.defaultValue()).toBe(''); // Default input value

    // Enter text and accept
    await dialog.accept('My Response');
  });

  await page.click('#show-prompt');
  await expect(page.getByText('You entered: My Response')).toBeVisible();
});
```

Different responses to dialogs:

```
test('conditional dialog handling', async ({ page }) => {
  await page.goto('https://example.com');

  page.on('dialog', async (dialog) => {
    const message = dialog.message();

    if (message.includes('delete')) {
      await dialog.accept(); // Confirm deletion
    } else if (message.includes('name')) {
      await dialog.accept('Test User'); // Enter name
    } else {
      await dialog.dismiss(); // Cancel others
    }
  });

  await page.click('#action-button');
});
```

Test without dialog handler (throws error):

```
  // This will throw an error if dialog appears
  // because there's no handler
  await expect(async () => {
    await page.click('#show-alert');
  }).rejects.toThrow();
});
```

BeforeUnload dialog:

```
test('handle beforeunload', async ({ page, context }) => {
  await page.goto('https://example.com/form');

  // Fill form
  await page.fill('#input', 'Some data');

  // Set up dialog handler
  page.on('dialog', async (dialog) => {
    expect(dialog.type()).toBe('beforeunload');
    await dialog.accept();
  });

  // Try to navigate away (triggers beforeunload)
  await page.goto('https://example.com/other-page');
});
```

Page Modals (Non-JavaScript dialogs)

Handle custom modals:

```
test('handle custom modal', async ({ page }) => {
  await page.goto('https://example.com');

  // Open modal
  await page.click('#open-modal');

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

  // Interact with modal
  await page.fill('.modal input', 'Modal input text');
```

```
  await page.waitForSelector('.modal', { state: 'hidden' });

  // Verify modal action
  await expect(page.getByText('Modal confirmed')).toBeVisible();
});
```

Close modal with different methods:

```
test('close modal variations', async ({ page }) => {
  await page.goto('https://example.com');
  await page.click('#open-modal');

  // Method 1: Click close button
  await page.click('.modal .close-button');

  // Method 2: Click overlay
  await page.click('#open-modal');
  await page.click('.modal-overlay', { force: true });

  // Method 3: Press Escape
  await page.click('#open-modal');
  await page.keyboard.press('Escape');
});
```

Authentication and Cookie Handling

Basic Authentication

HTTP Basic Auth:

```
test('basic authentication', async ({ browser }) => {
  const context = await browser.newContext({
    httpCredentials: {
      username: 'admin',
      password: 'admin123',
    },
  });
```

```
  // Now authenticated
  await expect(page.getByText('Admin Dashboard')).toBeVisible();

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

Configure in playwright.config.ts:

```
export default defineConfig({
  use: {
    httpCredentials: {
      username: process.env.AUTH_USERNAME || '',
      password: process.env.AUTH_PASSWORD || '',
    },
  },
});
```

Cookie Management

Set cookies:

```
test('set cookies', async ({ context, page }) => {
  // Set cookies before navigation
  await context.addCookies([
    {
      name: 'session_id',
      value: 'abc123',
      domain: 'example.com',
      path: '/',
    },
    {
      name: 'user_pref',
      value: 'dark_mode',
      domain: 'example.com',
      path: '/',
    },
  ]);

  await page.goto('https://example.com');
```

Get cookies:

```
test('get cookies', async ({ context, page }) => {
  await page.goto('https://example.com/login');

  // Login
  await page.fill('#username', 'user');
  await page.fill('#password', 'pass');
  await page.click('button[type="submit"]');

  // Get all cookies
  const cookies = await context.cookies();
  console.log('All cookies:', cookies);

  // Get specific cookie
  const sessionCookie = cookies.find(c => c.name === 'session_id');
  console.log('Session cookie:', sessionCookie);

  // Verify cookie exists
  expect(sessionCookie).toBeDefined();
});
```

Get cookies for specific URL:

```
test('get cookies for URL', async ({ context, page }) => {
  await page.goto('https://example.com');

  // Get cookies for specific URL
  const cookies = await context.cookies('https://example.com');

  cookies.forEach(cookie => {
    console.log(`${cookie.name}: ${cookie.value}`);
  });
});
```

Clear cookies:

```
test('clear cookies', async ({ context, page }) => {
  await page.goto('https://example.com/login');
```

```
  await page.click('button[type="submit"]');

  // Verify logged in
  await expect(page.getByText('Dashboard')).toBeVisible();

  // Clear all cookies
  await context.clearCookies();

  // Refresh page
  await page.reload();

  // Now logged out
  await expect(page.getByText('Login')).toBeVisible();
});
```

Clear specific cookies:

```
test('clear specific cookies', async ({ context, page }) => {
  await page.goto('https://example.com');

  // Clear specific cookie
  await context.clearCookies({
    name: 'session_id',
  });

  // Or clear cookies for specific domain
  await context.clearCookies({
    domain: 'example.com',
  });
});
```

LocalStorage and SessionStorage

Set localStorage:

```
test('set localStorage', async ({ page }) => {
  await page.goto('https://example.com');

  // Set localStorage
  await page.evaluate(() => {
    localStorage.setItem('user', JSON.stringify({
```

```
    }));
    localStorage.setItem('theme', 'dark');
  });

  // Reload to use stored data
  await page.reload();

  // Verify data is used
  await expect(page.locator('body')).toHaveClass(/dark-theme/);
});
```

Get localStorage:

```
test('get localStorage', async ({ page }) => {
  await page.goto('https://example.com');

  // Get localStorage data
  const userData = await page.evaluate(() => {
    const userJson = localStorage.getItem('user');
    return userJson ? JSON.parse(userJson) : null;
  });

  console.log('User data from localStorage:', userData);
  expect(userData).toBeDefined();
});
```

Clear localStorage:

```
test('clear localStorage', async ({ page }) => {
  await page.goto('https://example.com');

  // Clear localStorage
  await page.evaluate(() => {
    localStorage.clear();
  });

  await page.reload();

  // Verify cleared
  const storageLength = await page.evaluate(() => localStorage.length);
  expect(storageLength).toBe(0);
```

SessionStorage operations:

```
test('sessionStorage operations', async ({ page }) => {
  await page.goto('https://example.com');

  // Set sessionStorage
  await page.evaluate(() => {
    sessionStorage.setItem('temp_data', 'temporary value');
  });

  // Get sessionStorage
  const tempData = await page.evaluate(() => {
    return sessionStorage.getItem('temp_data');
  });

  expect(tempData).toBe('temporary value');

  // Clear sessionStorage
  await page.evaluate(() => {
    sessionStorage.clear();
  });
});
```

Authentication Patterns

Login once and reuse:

```
// auth.setup.ts
import { test as setup } from '@playwright/test';

setup('authenticate', async ({ page }) => {
  await page.goto('https://example.com/login');
  await page.fill('#username', 'user@example.com');
  await page.fill('#password', 'password123');
  await page.click('button[type="submit"]');

  await page.waitForURL('**/dashboard');

  // Save authentication state
  await page.context().storageState({ path: 'auth.json' });
});
```

```
// playwright.config.ts
export default defineConfig({
  projects: [
    {
      name: 'setup',
      testMatch: /auth\.setup\.ts/,
    },
    {
      name: 'authenticated tests',
      dependencies: ['setup'],
      use: {
        storageState: 'auth.json',
      },
    },
  ],
});
```

Test with pre-authenticated state:

```
test('user already logged in', async ({ page }) => {
  // Storage state loaded automatically
  await page.goto('https://example.com/dashboard');

  // Already authenticated
  await expect(page.getByText('Welcome')).toBeVisible();
});
```

Practice Exercises

Exercise 1: Event Monitoring System

Task: Build comprehensive monitoring

```
// Create a monitoring utility that:
// 1. Tracks all console messages (log, warn, error)
// 2. Monitors all network requests/responses
// 3. Captures page errors
```

```
class TestMonitor {
  // Implement monitoring class
  // Include methods for: start(), stop(), getReport()
}

test('monitored test', async ({ page }) => {
  const monitor = new TestMonitor(page);
  monitor.start();

  // Your test actions

  const report = monitor.getReport();
  console.log(report);
});
```

Exercise 2: API Mocking Framework

Task: Create reusable API mocking

```
 // Create a MockAPI class that:
// 1. Mocks multiple endpoints
// 2. Supports different response codes
// 3. Allows response delays
// 4. Tracks mock usage
// 5. Resets mocks between tests

class MockAPI {
  // Implement mock API functionality
  mockEndpoint(url: string, response: any, options?: {
    status?: number;
    delay?: number;
  }): void {
    // Implementation
  }
}

test('test with mocked APIs', async ({ page }) => {
  const mockApi = new MockAPI(page);

  mockApi.mockEndpoint('/api/users', { users: [...] });
  mockApi.mockEndpoint('/api/products', { products: [...] });

  // Run test with mocked APIs
```

Exercise 3: iframe Handler Utility

Task: Create iframe helper

```
 // Build a utility for iframe handling:
// 1. Wait for iframe to load
// 2. Switch between iframes easily
// 3. Handle nested iframes
// 4. Provide iframe-scoped actions

class IframeHelper {
  async switchTo(selector: string): Promise<FrameLocator> {
    // Implementation
  }

  async fillInIframe(
    iframeSelector: string,
    inputSelector: string,
    value: string
  ): Promise<void> {
    // Implementation
  }
}

test('complex iframe interaction', async ({ page }) => {
  const iframeHelper = new IframeHelper(page);

  await iframeHelper.fillInIframe('#payment-frame', '#card', '4111');
  // More actions
});
```

Exercise 4: File Upload/Download Test Suite

Task: Comprehensive file handling tests

```
test.describe('File Operations', () => {
  test('upload single file', async ({ page }) => {
    // Implement
  });

  test('upload multiple files', async ({ page }) => {
```

```
  test('upload and verify content', async ({ page }) => {
    // Implement
  });

  test('download and verify file', async ({ page }) => {
    // Implement
  });

  test('download PDF and check size', async ({ page }) => {
    // Implement
  });
});
```

Exercise 5: Authentication Flow

Task: Complete auth implementation

```
 // Implement:
// 1. Setup script to save auth state
// 2. Login test
// 3. Tests using saved auth
// 4. Logout test
// 5. Protected route test

// auth.setup.ts
setup('save auth', async ({ page }) => {
  // Implementation
});

// tests/authenticated.spec.ts
test('use saved auth', async ({ page }) => {
  // Implementation
});
```

Exercise 6: Network Performance Testing

Task: Measure and verify performance

```
 test('page performance', async ({ page }) => {
  // 1. Measure page load time
```

```
  // 5. Generate performance report

  // Your implementation
});
```

Exercise 7: Complete E2E Scenario

Task: Build end-to-end test with all concepts

```
test('complete user journey', async ({ page }) => {
  // 1. Mock slow API responses
  // 2. Monitor all network traffic
  // 3. Handle alerts/confirms
  // 4. Upload documents in iframe
  // 5. Download confirmation PDF
  // 6. Verify all operations
  // 7. Generate comprehensive report

  // Use all Day 5 concepts
});
```

Summary

In Day 5, you mastered:

✅ page.on() Event Handling

- Console message monitoring
- Page error capturing
- Request/response tracking
- Failed request handling
- Multi-event monitoring

✅ Network Interception

- Blocking resources
- Modifying requests/responses
- Network condition testing
- Performance analysis

✅ iframes and Shadow DOM

- Accessing iframe content
- Nested iframe handling
- Shadow DOM navigation
- Multiple iframe management

✅ File Operations

- Single/multiple file uploads
- File from buffer
- Drag and drop upload
- Download handling
- Download verification

✅ Dialogs

- Alert handling
- Confirm dialogs
- Prompt dialogs
- Conditional dialog responses
- Custom modal handling

✅ Authentication

- Basic HTTP authentication
- Cookie management
- LocalStorage/SessionStorage
- Auth state persistence

Key Takeaways

- **Event listeners** provide real-time monitoring
- **Network mocking** enables reliable testing
- **iframe handling** requires special techniques
- **File operations** are straightforward with Playwright
- **Auth state** can be saved and reused
- **Comprehensive monitoring** improves debugging

Advanced Patterns Learned

1. Create monitoring utilities for better observability
2. Mock APIs for consistent test data
3. Handle complex nested iframes
4. Verify file uploads and downloads
5. Persist authentication across tests
6. Analyze network performance

Production-Ready Techniques

- Error tracking and reporting
- API mocking strategies
- Performance monitoring
- File validation
- Session management
- Comprehensive test logging

Next Steps

- Apply these patterns to real applications
- Build reusable utilities
- Create test harnesses with monitoring

Congratulations! You've completed Week 1 of the QE Automation Training Program!

Week 1 Summary:

- Day 1: QE & TypeScript Fundamentals
- Day 2: Async Programming & Node.js
- Day 3: Playwright Basics
- Day 4: Advanced Playwright (POM, Fixtures, Data-Driven)
- Day 5: Advanced Scenarios (Events, Network, Files, Auth)

What's Next: Week 2

- Day 6: Playwright API Testing Fundamentals
- Day 7: Advanced API Testing
- Day 8: Cypress Fundamentals
- Day 9: Advanced Cypress & API Testing
- Day 10: Mobile Testing with WebDriverIO

End of Day 5 Documentation

← Advanced Playwright

Playwright API Testing →