The 7 Playwright Features You're Not Using (But Should Be)

I've reviewed dozens of Playwright codebases over the past year, and I keep seeing the same pattern: teams are using about 30% of what Playwright offers.

They've got the basics down. Locators, assertions, page objects. But they're missing features that could make their tests faster, more reliable, and way easier to maintain.

These aren't obscure tricks buried in documentation. They're practical features that solve real problems you're probably dealing with right now.

Let's fix that.

---

1. Trace Viewer for Debugging

What it is: Playwright's built-in time-travel debugger that records everything your test does, including screenshots, network requests, console logs, and DOM snapshots at each step.

Why you're missing out: Most people add console.log statements or run tests with headful mode to debug failures. That works, but it's slow and doesn't give you the full picture. You're guessing at what went wrong based on incomplete information.

How to use it: Enable tracing in your playwright.config.js:

```
use: {
  trace: 'retain-on-failure'
}
```

When a test fails, Playwright saves a trace file. Open it with `npx playwright show-trace trace.zip` and you get a complete recording of everything that happened. Click through each action, see the exact DOM state, check network calls, review console logs. It's like having a DVR for your test execution.

The impact: What used to take 20 minutes of re-running tests with different debug flags now takes 2 minutes of clicking through a trace file. You'll wonder how you ever debugged without it.

---

2. Auto-Waiting (and trusting it)

What it is: Playwright automatically waits for elements to be actionable before interacting with them. No manual waits needed.

Why you're missing out: I see codebases full of `await page.waitForTimeout(2000)` or `await page.waitForSelector('#element', { timeout: 5000 })` before every action. You're fighting Playwright's built-in intelligence instead of trusting it.

How to use it: Just call actions directly:

```
// Instead of this:
await page.waitForSelector('#submit-button')
await page.click('#submit-button')

// Do this:
await page.click('#submit-button')
```

Playwright waits for the element to be visible, enabled, and stable before clicking. It handles animations, loading states, and dynamic content automatically.

The impact: Your tests get faster (no unnecessary waits) and more reliable (Playwright's waiting logic is smarter than hardcoded timeouts). Your code gets cleaner too.

---

3. API Testing Within UI Tests

What it is: Playwright has a built-in API client that lets you make HTTP requests directly in your test code.

Why you're missing out: Teams often separate API tests into different tools (Postman, REST Assured) or skip them entirely. But some of the best test strategies combine UI and API calls in the same test.

How to use it:

```
const response = await request.post('https://api.example.com/users', {
  data: {
    name: 'Test User',
    email: 'test@example.com'
  }
})

const userId = (await response.json()).id

// Now use that user in your UI test
await page.goto(`/users/${userId}`)
```

Use API calls to set up test data quickly, then verify it in the UI. Or test the UI flow and verify backend state with API assertions.

The impact: Tests run faster because you're not clicking through entire setup flows. You have more control over test data. And you can verify both frontend and backend behavior in one test.

---

4. Network Interception and Mocking

What it is: The ability to intercept, modify, or mock network requests during test execution.

Why you're missing out: When third-party APIs are slow, flaky, or unavailable in test environments, people either skip those tests or deal with constant failures. Or they build elaborate test doubles in their backend code.

How to use it:

```
// Mock an API response
await page.route('**/api/products', route => {
  route.fulfill({
    status: 200,
    body: JSON.stringify([
      { id: 1, name: 'Test Product', price: 29.99 }
    ])
  })
})

// Or modify a real request
await page.route('**/api/checkout', route => {
  const request = route.request()
  route.continue({
    headers: {
      ...request.headers(),
      'Authorization': 'Bearer test-token'
    }
  })
})
```

The impact: Test flaky third-party integrations reliably. Simulate error conditions without breaking your backend. Test edge cases that are hard to reproduce with real APIs. Speed up tests by mocking slow endpoints.

---

5. Multiple Contexts for Isolation

What it is: Browser contexts are isolated browser sessions that share the same browser instance but have separate cookies, storage, and cache.

Why you're missing out: Most people create a new browser instance for every test, which is slow. Or they reuse the same context and deal with state bleeding between tests.

How to use it:

```
test.describe('User workflows', () => {
  test('admin user flow', async ({ browser }) => {
    const adminContext = await browser.newContext()
    const adminPage = await adminContext.newPage()
    // Test admin features
    await adminContext.close()
  })

  test('regular user flow', async ({ browser }) => {
    const userContext = await browser.newContext()
    const userPage = await userContext.newPage()
    // Test regular user features
    await userContext.close()
  })
})
```

Or test multi-user scenarios in the same test:

```
const adminContext = await browser.newContext()
const userContext = await browser.newContext()

const adminPage = await adminContext.newPage()
const userPage = await userContext.newPage()

// Admin creates something
await adminPage.goto('/admin/create-post')
await adminPage.fill('#title', 'New Post')
await adminPage.click('#publish')

// User sees it immediately
await userPage.goto('/posts')
await expect(userPage.locator('text=New Post')).toBeVisible()
```

The impact: Tests run faster (contexts are cheaper than browsers). You can test multi-user scenarios easily. Each test stays isolated without the overhead of spinning up multiple browsers.

---

6. Built-in Accessibility Testing

What it is: Playwright has native support for accessibility testing through its integration with axe-core.

Why you're missing out: Accessibility often gets tested manually or in separate tools. But catching accessibility issues in your automated tests means you find them before they reach production.

How to use it:

```
const { test, expect } = require('@playwright/test')
const AxeBuilder = require('@axe-core/playwright').default

test('homepage should not have accessibility violations', async ({ page }) => {
  await page.goto('/')

  const accessibilityScanResults = await new AxeBuilder({ page }).analyze()

  expect(accessibilityScanResults.violations).toEqual([])
})
```

You can also scan specific components or exclude certain rules if needed.

The impact: You catch accessibility issues automatically in every test run. Your application becomes more inclusive. And you satisfy compliance requirements without manual testing overhead.

---

7. Codegen for Quick Script Generation

What it is: A built-in tool that records your browser interactions and generates Playwright test code automatically.

Why you're missing out: People spend time writing boilerplate code for every new test, manually figuring out selectors, and debugging why their locators don't work.

How to use it: Run `npx playwright codegen https://your-site.com` and Playwright opens a browser and a code generator. Click around your application and watch it write the test code for you. Copy what you need, refine it, and you've got a working test in minutes.

Use it to:

- Get started quickly on a new test
- Figure out complex selectors
- Explore how Playwright handles tricky interactions
- Generate code for repetitive flows

The impact: You write new tests faster. You learn better locator strategies by seeing what Playwright chooses. And you spend less time debugging basic interaction code.

Start Using These This Week

You don't need to adopt all seven at once. Pick one that solves your biggest pain point right now:

- Spending too much time debugging? Start with Trace Viewer.
- Tests full of waits and timeouts? Trust auto-waiting.
- Dealing with flaky third-party APIs? Try network mocking.
- Need faster test execution? Use contexts instead of multiple browsers.
- Writing lots of similar tests? Use Codegen to speed up the initial draft.

These features are already in Playwright. You're already paying the cost of including the framework. You might as well get the full value.