ChatGPT

Boas Práticas de Uso do Playwright

Sumário

Estratégias eficazes para testes automatizados

Nesta seção abordamos boas práticas para criar testes automatizados estáveis e eficientes utilizando Playwright. Isso inclui técnicas de autenticação, reutilização de sessões, execução multiplataforma, integração contínua e uso adequado de fixtures no Playwright.

Login e reutilização de sessões

Evite refazer o processo de login antes de cada teste. Refazer a autenticação repetidamente consome tempo e pode tornar o teste instável, principalmente se depender de elementos de UI ou CAPTCHAs do fluxo de login [1] [2]. Em vez disso, prefira realizar o login uma única vez e reutilizar a sessão autenticada em todos os testes. O Playwright suporta o salvamento do estado de autenticação (cookies, localStorage etc.) através da função `browserContext.storageState()`, permitindo carregar esse estado nos testes subsequentes [3] [4]. Assim, elimina-se a necessidade de autenticar em cada teste, acelerando a suíte e evitando instabilidades [4] [5].

Uma prática recomendada é separar a lógica de login do restante dos testes. Por exemplo, pode-se criar um script ou teste especial de *setup* que realiza o login e salva o estado em um arquivo (por exemplo, `authState.json`). Em seguida, configurar o Playwright para usar esse estado em todos os testes, seja via configuração global ou via fixtures. A própria documentação do Playwright sugere esse approach: autenticar uma vez em um projeto de setup e reutilizar o estado em todos os testes, que já iniciam autenticados [6] [7]. O trecho de código abaixo ilustra esse conceito em TypeScript, realizando login e salvando o estado de sessão:

```
import { chromium } from '@playwright/test';
```

```
(async () => {
  const browser = await chromium.launch();
  const context = await browser.newContext();
  const page = await context.newPage();
  // Acessa a página de login e realiza a autenticação
  await page.goto('https://meuapp.com/login');
  await page.fill('#usuario', 'meu-usuario');
  await page.fill('#senha', 'minha-senha');
  await page.click('button[type="submit"]');
  await page.waitForURL('**/dashboard'); // espera redirecionar pós-login
  // Salva estado de autenticação (cookies, localStorage) em arquivo
  await context.storageState({ path: 'authState.json' });
  await browser.close();
})();
```

Em seguida, nos testes, é possível carregar este estado. Por exemplo, no `playwright.config.ts`:

```
import { defineConfig, devices } from '@playwright/test';
export default defineConfig({
  use: {
    storageState: 'authState.json' // Aplica estado logado em todos os testes
  }
});
```

Dessa forma, todos os testes iniciarão com a sessão já autenticada, pulando etapas de login. Essa abordagem traz múltiplos benefícios: **ganho de desempenho** (menos etapas repetitivas), **redução de flakiness** (menos dependência de elementos voláteis do login, como animações ou captchas) e **código de teste mais limpo**, já que os testes focam no que realmente importa (funcionalidade da aplicação) em vez de detalhes de autenticação [8] [9]. Estudos de caso mostram redução significativa no tempo de execução e na taxa de falhas falsos-positivas ao adotar a reutilização de sessão no Playwright [10] [11].

Algumas dicas adicionais ao lidar com estado de autenticação:

- **Estados por perfil ou papel de usuário:** se sua aplicação tem diferentes perfis (admin, usuário comum, etc.), considere gerar e armazenar múltiplos arquivos de estado (ex: `authState.admin.json`, `authState.user.json`) para cobrir cenários de autorização [12].
- **Validade da sessão:** tokens podem expirar. Automatize a recriação do arquivo de estado se ele estiver ausente ou expirado (por exemplo, renovando no início do pipeline CI) [13].
- **Segurança:** nunca commitar arquivos de sessão contendo cookies/tokens válidos. Adicione-os no `.gitignore` e trate-os como sensíveis [14]. Eles podem permitir a personificação do usuário de teste [15].
- **Execução paralela:** se executar testes em paralelo, assegure que não haja interferência entre testes devido ao compartilhamento de conta. Caso os testes modifiquem dados no servidor, talvez seja necessário usar estados de login distintos por worker (cada thread de teste com um usuário diferente) [16] [17]. Mas se os testes são só leitura (não alteram estado compartilhado), todos podem usar a mesma sessão sem problemas.

Em resumo, **mantenha o estado de autenticação entre os testes ao invés de logar a cada vez**. O Playwright oferece mecanismos nativos para isso (Storage State), e seguir essa prática melhora muito a eficiência e estabilidade dos testes [18] [19].

Testes multiplataforma (cross-browser e cross-device)

Um dos pontos fortes do Playwright é a capacidade de executar um mesmo teste em diferentes navegadores e ambientes de forma consistente. Ele suporta todos os principais motores de renderização: Chromium (Chrome/Edge), WebKit (Safari) e Firefox, com uma única API unificada [20]. Isso significa que você pode escrever seu script de teste uma vez e executá-lo em *todas* essas plataformas para garantir compatibilidade. De fato, a automação *cross-browser* é um dos pilares do Playwright [20].

Na prática, recomenda-se configurar *projetos* múltiplos no Playwright Test para cada navegador ou dispositivo alvo. Por exemplo, no arquivo de configuração é possível definir projetos para Desktop Chrome, Desktop Firefox e Mobile Safari (emulado via WebKit) utilizando os *device descriptors* fornecidos pelo Playwright. Cada projeto pode herdar os testes escritos, mudando apenas o contexto (navegador/viewport) em que rodam. Assim, você obtém facilmente cobertura multiplataforma sem duplicar código de teste.

Além de navegadores distintos, o Playwright permite testar em diferentes sistemas operacionais (Windows, Linux, macOS) e até em dispositivos móveis simulados. Os testes podem rodar em qualquer OS suportado, localmente ou no CI, garantindo que sua aplicação funcione em ambientes variados. Por exemplo, é possível emular um iPhone usando `...devices['iPhone 13']` no contexto de teste para verificar responsividade móvel.

Por que testar em múltiplas plataformas? Bugs específicos de navegador são uma realidade (diferenças de engine, suporte a recursos, tempos de resposta). Executar **testes em todos os navegadores relevantes** ajuda a pegar regressões que poderiam passar despercebidas se testássemos apenas em um browser [21]. Com Playwright isso é bastante simples, então aproveite essa facilidade. Certifique-se de incluir no seu pipeline ao menos os três motores (Chromium, WebKit, Firefox) a não ser que sua aplicação não os suporte oficialmente.

No contexto de projetos *cross-browser*, vale lembrar de algumas práticas:

- **Use seletores e asserts resilientes:** Pequenas diferenças de renderização ou temporização entre navegadores podem afetar testes. Utilize seletores robustos (ex: atributos data-test-id ou texto visível) e tire proveito do *auto-wait* do Playwright para sincronização (ele já aguarda elemento estar visível e estável antes de interagir [22]).
- **Considere diferenças de UI/UX:** Um componente pode ser renderizado diferentemente no Safari vs Chrome. Se necessário, adapte as expectativas no teste ou use condicional por `browserName` quando um comportamento for exclusivo.
- **Emulação de dispositivos:** Ao testar versões mobile via emulação, lembre de possíveis diferenças de viewport ou eventos de toque. Playwright provê dispositivos predefinidos com user-agent e dimensões correspondentes.

Em suma, **garanta cobertura multi-browser** nos seus testes automatizados. O Playwright torna isso simples e eficaz, permitindo entregar aplicações mais confiáveis em qualquer ambiente de usuário final [20].

Integração de testes em CI/CD

Incorporar a suíte de testes Playwright no processo de *Continuous Integration/Continuous Deployment (CI/CD)* é fundamental para detectar regressões o mais cedo possível. **Teste automatizado deve fazer parte de todo pipeline de deploy**, barrando a promoção de código com falhas [23]. Assim, ao cada nova alteração de código (commit ou pull request), seu conjunto de testes roda automaticamente, garantindo que novas funcionalidades não quebrem as existentes.

Para integrar no CI (como GitHub Actions, GitLab CI, Jenkins, Azure Pipelines, etc.), seguem algumas práticas:

- **Ambiente de execução consistente:** Utilize a imagem Docker oficial do Playwright ou instale as dependências de navegadores na máquina de CI com `npx playwright install --with-deps`. Isso garante que os navegadores (Chromium, Firefox, WebKit) estejam disponíveis mesmo em um ambiente de container Linux headless [24].
- **Executar em modo headless:** Em servidores CI geralmente não há interface gráfica, então por padrão o Playwright executará headless. Isso também torna os testes mais rápidos e adequados ao CI [25].
- **Configuração de workers:** Por padrão, o Playwright pode executar testes em paralelo usando múltiplos workers. Entretanto, para priorizar estabilidade no CI (que costuma ter recursos limitados e variáveis), recomenda-se configurar `workers: 1` quando `process.env.CI` estiver definido [26]. Executar testes sequencialmente evita condições de corrida por recursos no sistema e facilita reproduzir erros. Se você dispõe de infraestrutura robusta (self-hosted runners potentes), pode usar paralelismo, mas de forma controlada. Outra opção para acelerar sem instabilidade é usar **sharding**: distribuir diferentes testes em máquinas separadas no CI [27] [28].
- **Reports e artefatos:** Configure o pipeline para armazenar os relatórios de teste e outras evidências (logs, screenshots, vídeos de falha, traces). Por exemplo, no GitHub Actions você pode usar `actions/upload-artifact` para guardar a pasta `playwright-report` gerada pelo Playwright [29]. Assim, em caso de falha, você pode baixar o relatório HTML ou vídeos para investigar o que houve.
- **Passo obrigatório no pipeline:** Certifique-se de que a etapa de testes seja um *gate* obrigatório antes do deploy em produção. Idealmente, os testes rodam em cada commit/pull request em branch principal [30]. Se algum teste falha, o pipeline deve marcar erro e impedir merge/deploy até a correção.
- **Recursos de debug no CI:** Habilite captura de trace (`--trace on` ou on-first-retry), screenshots em falhas e eventualmente vídeos no contexto de teste para ajudar a diagnosticar flakiness no CI. O impacto de performance é pequeno e pode poupar muito tempo na depuração de problemas que ocorrem só no ambiente CI.
- **Estratégias para flakiness:** No CI, se um teste falhar esporadicamente (flaky), o Playwright tem opção de reexecutar automaticamente (`retries`) apenas naquela run, para distinguir falha real de intermitente. Use com moderação – idealmente, investigue e corrija a causa raiz em vez de ocultá-la com múltiplas tentativas [31] [32].

Integrar o Playwright ao pipeline traz **consistência nos resultados** em diversos ambientes e evita o clássico "funciona na minha máquina, mas quebrou em produção" [23]. Ao seguir essas práticas, seus testes rodarão de forma confiável no CI, tornando-se um componente valioso da qualidade do seu software. Lembre-se: **testes automatizados devem ser tratados como código de primeira classe**, versionados e executados continuamente para entregar valor.

Uso de fixtures no Playwright

O Playwright Test (framework de testes embutido no Playwright para Node.js/TS) introduz o conceito de **fixtures**, que são estruturas reutilizáveis de configuração e teardown para os testes. Fixtures funcionam de forma semelhante a ganchos *beforeEach/afterEach*, porém de maneira muito mais poderosa e declarativa [33] . Com fixtures, você pode definir recursos que seus testes precisam (como um contexto autenticado, dados de entrada, páginas preparadas) e o Playwright garante que esses recursos sejam provisionados antes de cada teste e limpos depois.

Alguns pontos-chave sobre fixtures no Playwright:

- O Playwright já fornece **fixtures embutidas** como `page`, `context` e `browser` para cada teste. Ou seja, por padrão cada caso de teste recebe um novo `page` (página/navegador) isolado para usar. Isso previne interferência entre testes (isolamento por contexto) e permite paralelização segura.
- Podemos **estender** as fixtures padrão criando nossas próprias. Por exemplo, é comum criar uma fixture de página autenticada. Em TypeScript, pode-se fazer algo como:

```
import { test as baseTest, chromium, Page } from '@playwright/test';

type MyFixtures = {
  loggedPage: Page;
};

export const test = baseTest.extend<MyFixtures>({
  loggedPage: async ({}, use) => {
    // Inicia navegador e contexto
    const browser = await chromium.launch();
    const context = await browser.newContext({ storageState:
'authState.json' });
    const page = await context.newPage();
    // Navega para a página inicial já autenticado
    await page.goto('https://meuapp.com/dashboard');
    // Disponibiliza a página autenticada para o teste
    await use(page);
    // Cleanup após o teste
    await context.close();
    await browser.close();
  }
});
```

No exemplo acima, definimos uma fixture `loggedPage` que fornece ao teste uma página já logada (usando um `storageState` previamente salvo). Dentro de cada teste, podemos usar essa página autenticada sem precisar repetir código de login. O Playwright cuida de rodar o setup da fixture antes do teste e o teardown (fechando contexto/navegador) após o teste automaticamente [34] [35] .

- **Vantagens de fixtures:** seu código de teste fica mais limpo e declarativo, pois a configuração pesada fica centralizada na definição das fixtures. Como comentou um autor, fixtures trazem consistência e evitam duplicação, fornecendo pontos de extensão limpos para preparar contexto

compartilhado entre testes [33]. É uma melhoria enorme comparado a chamar funções de setup manualmente em cada teste.

- **Reutilização e composição:** Fixtures podem depender de outras fixtures. Por exemplo, sua fixture customizada pode receber o `browser` ou `context` padrão do Playwright se preferir, ou até outra fixture custom (via parâmetros no factory async). Isso favorece a composição de setups complexos.
- **Setup global vs fixtures por teste:** O Playwright também suporta um arquivo de *global setup* (executado uma vez antes de todos os testes, fora do contexto de cada teste). Isso é útil, por exemplo, para criar estado de autenticação compartilhado (como vimos na seção de login). Já as fixtures operam no nível dos testes ou dos workers. Avalie o escopo correto: se algo é caro e pode ser reutilizado por todos os testes, use global setup (ex: gerar um arquivo de cookies); se precisa de isolamento por teste ou por thread, use fixtures.
- **Evitar código repetitivo nos testes:** Um caso comum é evitar repetir a visita a páginas de login em todo teste. Conforme discutido, podemos usar um fixture que já fornece a página pós-login e assim **pular telas iniciais em todos os casos de teste** [36] [37]. Isso economiza tempo (menos navegações e cliques) e previne efeitos colaterais (por exemplo, bater no endpoint `/login` em cada teste poderia sobrecarregar algo ou produzir logs indesejados).

Em resumo, **fixtures são ferramentas essenciais para estruturar testes Playwright de forma escalável e manutenível**. Elas garantem que cada teste comece em um estado conhecido e termine liberando recursos, facilitando execução paralela e evitando "vazamentos" de estado entre casos. Ao adotar fixtures, busque definir aquelas que fazem sentido para seu domínio (ex: página autenticada, dados de teste configurados, mocks ativos) e assim seus testes ficarão mais concisos e robustos.

(Dica: A documentação do Playwright oferece vários exemplos de fixtures, e há bibliotecas e artigos com padrões úteis. Vale a pena investir tempo para dominar esse recurso.)

Técnicas e precauções para scraping com Playwright

O Playwright não é apenas para testes; ele também é bastante utilizado para **web scraping**, pois permite automatizar navegadores reais e extrair dados de páginas web, inclusive as dinâmicas. Nesta seção, discutimos boas práticas para extrair dados de forma eficaz e evitar armadilhas comuns (como conteúdo dinâmico não carregado, detecção de bot, CAPTCHAs etc.).

Manipulação de páginas dinâmicas

Diferentemente de um simples HTTP request, usar o Playwright para scraping envolve carregar a página num navegador real e possivelmente interagir com ela (clicar, rolar, etc.) para obter todo o conteúdo desejado. Muitos sites hoje são **Single Page Applications (SPA)** ou carregam dados via requisições assíncronas, então é crucial saber **quando** os dados que queremos estão presentes no DOM.

Por padrão, `page.goto(url)` aguardará o evento "load" da página, que nem sempre garante que todo conteúdo dinâmico foi renderizado. Uma estratégia melhor é aguardar por sinais específicos de que o conteúdo foi carregado. O Playwright oferece mecanismos como:

- `waitForSelector(selector)`: espera até que um determinado seletor apareça na página. Por exemplo, após navegar até uma lista de produtos, você pode esperar `page.waitForSelector('.produto-card')` para garantir que ao menos um elemento de produto foi inserido no DOM [38]. Você pode ajustar um `timeout` adequado caso a página demore (ex: 10 segundos).

- Estados de carga da página: `waitForLoadState('domcontentloaded')` aguarda o HTML inicial ser carregado e parseado, enquanto `waitForLoadState('networkidle')` pode ser útil para esperar que não haja mais requisições de rede em andamento (indicando fim de carregamentos adicionais) [38]. Em sites SPA, após interações que disparam requisições XHR/fetch, `networkidle` é valioso para saber quando as respostas chegaram.
- Em alguns casos, pode ser necessário inserir **atrasos manuais** (`waitForTimeout`) se o site tem animações ou períodos conhecidos de espera. Porém, evite usar timeouts fixos desnecessariamente – prefira os waits inteligentes acima, que são mais resilientes. O uso de *explicit wait* só deve ocorrer se não houver um gatilho determinístico para aguardar.
- **Monitoramento de rede ou eventos:** Para casos avançados, você pode usar `page.on('response')` para vigiar respostas XHR específicas e só prosseguir quando determinada resposta chegar, ou usar `page.waitForEvent('response', {...})`. Outra técnica é avaliar periodicamente uma condição com `page.waitForFunction`, por exemplo esperar até que `document.querySelectorAll('.produto').length > 0`.

Exemplo de abordagem em pseudo-código para extrair títulos de produtos de uma página dinâmica:

```
await page.goto('https://loja.com/produtos', { waitUntil:
'domcontentloaded' });
// Espera até que os elementos de produto estejam no DOM (até 10s)
await page.waitForSelector('.produto-item', { timeout: 10000 });
const titulos = await page.$$eval('.produto-item .titulo', els => els.map(e
=> e.textContent));
console.log(titulos);
```

No exemplo, garantimos que `.produto-item` existe antes de extrair os títulos, evitando coletar um array vazio devido a carregamento lento.

Também atente para **páginas com rolagem infinita ou paginação via scroll** (ex: feed de redes sociais). Nesses casos, você precisará simular a rolagem com algo como:

```
await page.evaluate(() => window.scrollBy(0, window.innerHeight));
```

E possivelmente chamar isso repetidas vezes, ou usar `page.mouse.wheel(0, valor)` para disparar o scroll, verificando se novos itens carregaram entre scrolls. Uma abordagem comum é: enquanto o seletor do último item novo continuar aparecendo, continuar scrollando.

Resumindo, para scraping de páginas dinâmicas use as ferramentas de espera do Playwright a seu favor. **Não tente adivinhar tempos**; espere por condições concretas (elementos ou inatividade de rede) para obter dados confiáveis [38]. Isso torna seu scraper mais robusto a variações de desempenho do site e evita capturar informações incompletas.

Modos headless vs headful

O Playwright permite rodar navegadores em modo **headless** (sem interface gráfica) ou **headful** (com interface visível). Para web scraping, entender as implicações de cada modo é importante:

- **Headless:** É o modo sem cabeça, ideal para automação em servidores. Ele consome menos recursos (não renderiza UI na tela) e tende a ser mais rápido [25]. Em ambientes CI e na maioria dos scrapers em produção, este é o modo usado por padrão. No Playwright, ao usar `browserType.launch()`, se não especificado, o padrão é headless=true.
- *Vantagem:* desempenho e capacidade de paralelizar muitas instâncias sem precisar de placas de vídeo ou servidores X. Além disso, evita janelas se abrindo na sua máquina local durante o desenvolvimento quando você não precisa ver a UI.

- *Desvantagem:* headless browsers podem ser mais facilmente detectados por contramedidas anti-bot. Historicamente, propriedades como `navigator.webdriver` ficam true em headless, a string do User-Agent às vezes indica headless, e certas APIs como WebGL ou WebRTC podem se comportar diferente ou ausentes, denunciando a automação [39] [40]. Muitos sites usam essas pistas para mostrar CAPTCHAs ou bloquear acesso automatizado.

- **Headful (headed):** É rodar o navegador com interface visível (como um usuário normal). No Playwright, basta chamar `launch({ headless: false })`.

- *Vantagem:* maior fidelidade ao comportamento real de um usuário. Como o ambiente headful carrega extensões/plugins padrões do navegador e exibe tudo normalmente, há menos *fingerprints* indicando que é um robô. Isso pode **reduzir a detecção** – alguns sites não aplicam CAPTCHA quando percebem um navegador real com janela e comportamento humano. Além disso, para o desenvolvedor, ver o que acontece facilita o debug durante a criação do script.
- *Desvantagem:* consumo de recursos e velocidade. Renderizar a interface gasta CPU/GPU; um navegador headful é mais pesado. Para scrapers em larga escala, headful é pouco prático. Também, mesmo headful, bots ainda podem ser detectados por outras heurísticas (tempo de reação impossível para um humano, padrões de acesso, fingerprint de canvas, etc.).

Podemos resumir as diferenças principais em uma tabela:

Modo	Vantagens	Desvantagens
Headless	- Execução mais rápida (sem carregar interface gráfica) - Ideal para ambientes CI e alto paralelismo - Menor consumo de memória e CPU 📉	- Mais sujeito a detecção automática (ex: `navigator.webdriver=true`) ⚠ [39] - Depuração menos intuitiva (não dá para "ver" o que o bot vê facilmente)
Headful	- Simula mais fielmente um usuário real (menos flags de bot) 🕵 - Permite assistir as interações ao vivo (útil para debug) 📱	- Mais lento e pesado (renderização completa da página) - Exige ambiente gráfico (problemático em servidores Linux sem X) 🖱 - Escalabilidade limitada para muitas instâncias simultâneas

Em geral, ao desenvolver seu scraper, você pode **começar em headful para montar e depurar o fluxo**, e depois rodar em headless em produção. Se você enfrentar bloqueios ou CAPTCHAs frequentes

somente em headless, pode ser um indicativo de detecção; nesse caso avalie técnicas anti-detecção (ver próxima seção) ou em último caso rode headful (com Xvfb em servidores Linux para emular display, por exemplo).

Importante destacar: as capacidades do Playwright (e.g. quais APIs Web são suportadas) são as mesmas em ambos os modos, pois o engine do navegador é o mesmo. A diferença está em como o ambiente é exposto ao site. Por exemplo, **headless Chrome pode ser imediatamente identificado** via scripts que checam propriedades específicas do agente [40]. Felizmente, existem soluções para amenizar isso sem abrir mão do headless, como veremos adiante (ex: plugin *stealth*).

Dica: mantenha uma opção configurável no seu código para alternar headless/headful facilmente (por variável de ambiente ou parâmetro). Assim, se precisar investigar o comportamento do scraper, pode rodá-lo com interface visível para entender o que ocorre.

Gerenciamento de cookies e sessão em scraping

Cookies são dados cruciais em scraping, especialmente se envolvemos sessões autenticadas ou queremos evitar mecanismos de bloqueio baseados em repetição de acessos. Boas práticas sobre cookies no contexto do Playwright:

- **Reutilize sessões quando possível:** Similar aos testes, se você precisa fazer login para extrair dados (por exemplo, scraping de um painel interno), é muito útil salvar os cookies da sessão uma vez e reutilizar nos próximos scrapes. Com Playwright, você pode obter os cookies atuais com `context.cookies()` e salvar em arquivo JSON [41]; depois, inicializar um novo contexto já carregando esses cookies via `browser.newContext({ storageState: 'cookies.json' })` ou usando `context.addCookies([...])` com os valores lidos. Isso economiza o overhead de logar toda vez e pode também evitar gatilhos de segurança por múltiplos logins sucessivos.
- **Persistência entre execuções:** Diferente de testes que geralmente começam limpos a cada suite, em scraping talvez você queira **continuar de onde parou** entre execuções do script. Por exemplo, preservar cookies de sessão ou preferências. Playwright permite que você *exporte* todo o estado de armazenamento (cookies + localStorage) facilmente e depois *importe* em outra execução. Use isso a seu favor para manter sessões vivas.
- **Isolamento de contexto:** Ao coletar dados de diversos sites, crie contextos separados para cada site, assim os cookies e cache de um alvo não vazam para outro. Cada `browser.newContext()` é como um perfil novo de navegador, isolado. Isso evita interferências e também permite configurar user-agent ou proxies distintos por contexto sem conflito.
- **Limpeza e renovação:** Esteja preparado para cenários em que cookies expiram ou ficam inválidos (ex: após logout, ou expiração natural). Seu código deve detectar uma página que redirecionou para login inesperadamente e então refazer a autenticação. Alternativamente, automatize a renovação de cookies antes da expiração conhecida (se você controla essa informação). Também, se usar cookies persistidos em arquivo, tenha alguma política de limpeza para não usar cookies muito antigos que possam gerar comportamentos estranhos.
- **Cookies de consentimento:** Muitos sites exibem banners de "aceitar cookies". Isso pode atrapalhar o scrape se o banner cobrir partes da página. Você pode lidar com isso de algumas formas: clicar no botão "Aceitar" via script (detectando o seletor do banner), ou já iniciar o contexto com um cookie de consentimento definido (se souber o nome e formato do cookie de consentimento após aceitar uma vez manualmente). A segunda opção é interessante: por exemplo, após aceitar cookies manualmente, extraia esse cookie e adicione em todos os novos contextos para aquele domínio – assim o banner nem aparece.

- **Bloqueio de recursos indesejados:** Não diretamente cookies, mas relacionado: ao raspar, considere bloquear recursos que não impactam os dados (anúncios, trackers) para reduzir carga. Você pode usar `page.route('**/*.{png,jpg}', route => route.abort())` etc., mas cuidado para não bloquear algo essencial. Isso melhora desempenho mas teste bem para não impedir carregamento de conteúdo dinâmico necessário.

Em resumo, **trate os cookies de forma estratégica no scraping**. Eles podem ser aliados (mantendo sessão, pulando etapas) ou vilões (acúmulo de estado sujo). O Playwright dá controle total sobre cookies, então use métodos como `context.cookies()`, `context.addCookies()` e `context.clearCookies()` para gerenciá-los conforme suas necessidades. Uma abordagem atenciosa ao manejo de sessão resultará em scrapers mais resilientes e eficientes.

Lidando com CAPTCHAs e anti-bots

Uma das maiores dificuldades no web scraping moderno são as defesas anti-bot: CAPTCHAs, bloqueios por IP, desafios como Cloudflare IUAM, checagem de fingerprint, etc. Como o Playwright dirige um navegador real, você já supera muitas limitações de scrapers simples, mas ainda assim **pode ser detectado** se não tomar cuidado [39]. Vamos pontuar estratégias e precauções para contornar esses obstáculos:

1. Entenda os gatilhos de detecção: Sites detectam bots procurando comportamentos ou assinaturas incomuns. Alguns sinais comuns que **Playwright/Chrome headless expõem por padrão** incluem: o cabeçalho `User-Agent` indicando headless, a propriedade `navigator.webdriver = true`, ausência de plugins instalados, fontes ou canvas com características padrão, timings de execução super-humanos, entre outros [39]. Ou seja, *sem nenhuma contramedida*, seu scraper pode apresentar vários *sinais de robô* que disparam CAPTCHAs.

2. Use técnicas de *stealth* (ofuscação): A comunidade criou plugins para Playwright com intuito de esconder ou falsificar essas características. Por exemplo, o **Playwright Stealth** (plugin do pacote `playwright-extra`) aplica diversas modificações no browser para torná-lo mais similar a um usuário real. Ele ajusta o User-Agent para não mostrar "HeadlessChrome", insere objetos de browser para simular plugins e WebGL, desativa o WebRTC leak de IP, e define `navigator.webdriver = false`, dentre outras táticas [42] [43]. Isso tudo ajuda a evitar gatilhos automáticos de bot-detection. Para usá-lo, em Node.js você instalaria `playwright-extra` e `playwright-extra-plugin-stealth`, ou em Python o pacote `playwright-stealth` (conforme exemplo ZenRows) e aplica antes de navegar [44] [45]. Vale notar: nenhuma solução é 100% infalível – por exemplo, o Stealth plugin atual não consegue driblar CAPTCHAs mais avançados como o Cloudflare Turnstile [46] – mas certamente **reduz bastante a frequência de desafios** em muitos sites.

3. Rotacione proxies e IPs: Muitas proteções contam a frequência de acessos por IP. Usar IPs residenciais ou móveis em rotação é uma prática quase obrigatória se você raspa em escala ou sites protegidos [47]. Proxies de datacenter tendem a ser bloqueados mais rápido; já IPs domésticos se misturam ao tráfego comum. Combine o Playwright com uma boa estratégia de proxies: alterne IP periodicamente, use sessões persistentes (sticky) quando precisar manter login por várias páginas (para que o cookie de sessão permaneça válido no mesmo IP) [47]. Muitos serviços oferecem APIs com rede de proxies rotativos + anti-bot integrados. Mesmo sem serviços, evite saturar um único IP com muitas requisições em curto período.

4. Imite comportamento humano: Pequenas aleatoriedades podem ajudar: adicionar `await page.waitForTimeout(50 + Math.random()*100)` entre ações para não parecer um robozinho clicando instantâneo; ou movimentar o mouse um pouco (`mouse.move(x, y)`) antes de clicar em

algo, etc. O Playwright em si já aguarda os elementos estarem prontos e executa ações muito rápido; introduzir delays aleatórios pode evitar sistemas que detectam cliques muito rápidos para um humano. Entretanto, isso sacrifica desempenho – então aplique apenas se necessário em alvos específicos conhecidos por bloqueios sofisticados.

5. Resolver ou contornar CAPTCHAs: Se, apesar de tudo, você se depara com CAPTCHAs, tem poucas saídas: ou **resolve-os** ou **previne que apareçam**. Resolver pode ser feito via serviços externos como 2Captcha, Anti-Captcha, etc., que usam humanos ou modelos para resolver o desafio e retornar o token. O Playwright pode integrar-se a esses serviços – por exemplo, usando a API deles para enviar a imagem ou chave do reCAPTCHA e inserir a resposta quando pronta [48] [49]. Há até bibliotecas que facilitam isso (como `2captcha-python` mostrado por ZenRows). Contudo, essa abordagem **custa dinheiro e tempo** (cada captcha pode levar ~10-20 segundos para ser solucionado). Em larga escala, torna-se inviável. A melhor abordagem é realmente **evitar acionar o CAPTCHA** através das técnicas anteriores (stealth, proxies, timmings). Isso inclui também evitar interações suspeitas – por exemplo, algumas páginas disparam CAPTCHA se você faz muitas navegações muito rápido; mitigue reduzindo a velocidade ou usando múltiplas sessões em paralelo com baixo throughput individual.

6. Utilize serviços especializados se necessário: Caso você não queira lidar manualmente com tudo isso, existem serviços (como o próprio ZenRows, ScrapingAnt, Browserless, etc.) que oferecem APIs onde você faz uma requisição e eles retornam o HTML já pós-desbloqueio. Esses serviços internamente usam navegação com anti-block, resolvem CAPTCHAs, gerenciam proxies, etc. [50] [51]. É uma solução paga, mas que pode economizar tempo de desenvolvimento se scraping for atividade crítica para você. Uma alternativa self-hosted é usar o **Browserless** com BQL (Browser Query Language) que automatiza muitas dessas melhores práticas (stealth, rotação, solve) por você [52] [53].

Resumindo, **raspar a web de forma robusta exige driblar mecanismos anti-robô**. Com Playwright, você já sai na frente por se passar por um navegador completo. Mas para alvos mais protegidos, combine as estratégias: *stealth headless*, proxies de qualidade, e comportamento humanizado. Lembre-se que qualquer brecha na detecção pode levar a bloqueios (temporários ou permanentes). Então, monitore as respostas do site – se receber códigos 429/403 ou páginas com desafios, adapte seu scraper. Às vezes, a solução é tão simples quanto diminuir a frequência de acessos.

Por fim, seja ético no scraping: respeite o `robots.txt` quando aplicável, não sobrecarregue sites com muitas threads e intervalos zero, e cumpra termos de uso. Evitar bloqueios também passa por não chamar atenção indevida

Aplicações do MCP com Playwright e testes por IA

Uma novidade empolgante no ecossistema de testes é a junção de **Playwright + IA** para gerar e executar casos de teste automaticamente. O **MCP (Model Context Protocol)** é a peça central dessa integração, permitindo que agentes de IA (como modelos de linguagem tipo GPT-4, Claude etc.) controlem o navegador via Playwright e, com isso, tenham contexto real para escrever testes. Nesta seção, explicamos o que é o MCP, como ele se integra ao Playwright e como possibilita testes gerados/dirigidos por Inteligência Artificial, além de discutir benefícios e limitações dessa abordagem.

O que é o Model Context Protocol (MCP)?

O Model Context Protocol, criado originalmente pela Anthropic, é um **protocolo padrão para conectar modelos de IA a ferramentas externas** de forma estruturada. Em essência, ele define como um modelo (ex: um chatbot) pode requisitar ações a um "servidor" que executa essas ações no mundo real

(como navegar em um site, ler um arquivo, enviar um email) e retorna o resultado para o modelo continuar o raciocínio. Pense no MCP como a linguagem que permite ao modelo dizer: "Abra o navegador nessa URL e me diga o que você viu lá."

No contexto do Playwright, o MCP assume a forma de um servidor especializado que expõe comandos do navegador. **O Playwright MCP Server** é uma implementação oficial que **permite à IA controlar um navegador real usando as APIs do Playwright** [54]. Ou seja, ao conectar um modelo de linguagem a esse servidor, ele ganha "mãos e olhos": pode navegar para páginas, clicar em elementos, ler textos, tudo via Playwright, e então receber de volta dados do estado atual da página (HTML, capturas de tela, URL atual, valores de cookies, etc.). Com isso, supera-se a limitação dos LLMs de não terem conhecimento em tempo real da interface sob teste.

Em termos práticos, o MCP define um conjunto de **ferramentas/comandos** que o modelo pode chamar. Por exemplo, comandos como `browser_newContext`, `browser_newPage`, `browser_navigate`, `page_click`, `page_input`, até comandos para obter o HTML (`page_content`) ou tirar screenshot. Cada vez que o modelo invoca um desses, o servidor Playwright executa no navegador e retorna o resultado (por exemplo: após um `navigate`, retorna o HTML da página carregada, ou após um `click`, informa que a ação foi feita e talvez dá um novo HTML resultante).

O importante é: sem esse protocolo, se você simplesmente pedir para ChatGPT "escreva um teste para meu site", ele **inventará** seletores e passos baseados no que conhece genericamente (muitas vezes alucinando coisas que não existem) [55]. Com o MCP, o modelo pode realmente inspecionar o site e tomar decisões embasadas no conteúdo real, eliminando adivinhações. Fornecer **contexto valioso é o "molho secreto" para a geração de testes por IA dar certo** [56] – e o MCP é o meio de obter esse contexto diretamente da aplicação em execução.

Integração do MCP com o Playwright

Para usar o MCP com Playwright, você precisa rodar um **servidor MCP do Playwright** e conectá-lo a um cliente/ambiente de IA que suporte MCP. Atualmente, alguns assistentes de código e plataformas de chat avançadas já suportam esse protocolo (ex: o GitHub Copilot Chat, Cursor AI, Claude com tools, etc., estão adotando padrões de tool use como o MCP) [57]. Em um cenário típico:

- Você inicializa o servidor MCP do Playwright (por exemplo, `npx @playwright/mcp@latest`), eventualmente indicando configurações como porta, etc. Esse servidor aguardará comandos.
- No lado do modelo de IA, você prepara um *prompt* e credenciais para ele se conectar a esse servidor. Muitas ferramentas usam um arquivo `.mcp.json` ou similar para listar servidores disponíveis (como um chamado "playwright" apontando para seu servidor).
- Uma vez conectado, o modelo "sabe" que tem a ferramenta `playwright` disponível e quais ações pode executar através dela.

A **magia** acontece quando a IA decide usar as ferramentas do MCP para alcançar um objetivo. Por exemplo, se pedimos: "Teste se ao buscar 'Star Wars' no meu site de filmes aparece o resultado correto", o agente de IA pode: chamar `browser_newPage`, depois `browser_navigate` com a URL inicial, em seguida localizar o campo de busca e usar `page_fill` e `page_click` no botão, depois `page_content` ou `page_querySelector` para extrair o resultado e verificar. Cada chamada retorna informação que o modelo incorpora no seu contexto para a próxima etapa [58] [59]. O Protocolo MCP cuida de serializar esses resultados de forma que o modelo possa lê-los (geralmente como trechos textuais truncados se for HTML grande, possivelmente acompanhados de metadados como URL atual, etc.).

Em termos de implementação de teste, existem dois modos gerais mencionados:

- **Modo *agent* exploratório:** Aqui dá-se autonomia para o modelo explorar o aplicativo em busca de funcionalidades e problemas. Você fornece um objetivo genérico ("Explore o site X e gere testes para funcionalidades-chave") e o agente vai navegando em estilo livre. O exemplo da Debbie O'Brien demonstra isso: ela rodou o agente em modo autônomo no app de filmes, e ele por conta própria clicou em busca, temas, navegação, etc., descobrindo inclusive um bug no caminho [60] [61]. Esse modo é interessante para descobrir cenários de teste automaticamente, quase como um QA exploratório realizado pela IA.
- **Modo dirigido por cenário:** Aqui você passa um cenário específico ou caso de uso, e a IA utiliza o MCP para implementar aquele fluxo. Por exemplo: "Faça login no site, vá até a página de perfil, edite o nome e salve; depois verifique se aparece uma mensagem de sucesso." A IA seguiria esses passos chamando as ações necessárias no browser via MCP e, ao final, produziria um código de teste automatizado correspondente.

Uma vez que o modelo tenha coletado contexto suficiente (estruturas de página, textos, etc.), ele pode então **gerar o código do teste Playwright (em TypeScript, por exemplo)**. Ferramentas como Copilot Chat conseguem não só gerar o código como também inseri-lo no seu editor diretamente. Alguns setups vão além: fazem o agente *salvar* o arquivo de teste e até executá-lo em seguida, iterando caso falhe, até conseguir um teste que passe [62] [63]. Isso é possível porque o agente pode invocar a execução dos testes (via comando do MCP ou via integração com o ambiente dev) e reagir ao resultado (se falhou, ler o erro e ajustar o código).

Resumindo, a integração MCP+Playwright **coloca o modelo de IA "dentro do loop" do teste end-to-end**. Em vez de trabalhar só com texto estático ou código, ele interage com a aplicação real enquanto escreve o teste. É um novo paradigma de automação assistida por IA que promete acelerar a criação de testes e aumentar a cobertura.

Geração de testes automatizados via IA

Vamos entender como esse casamento de IA e Playwright se traduz na prática da geração de testes:

Imagine que você tem uma aplicação web e quer criar testes automaticamente. Sem IA, você poderia usar o Codegen do Playwright para gravar interações manuais e gerar código. Com IA + MCP, a proposta é que o **próprio modelo seja capaz de navegar e gerar os testes**, potencialmente encontrando cenários que você não cobriu.

No caso do *agent mode*, o agente de IA pode agir quase como um usuário exploratório. O relato de Debbie O'Brien mostra o agente navegando em um app de filmes: ele procurou um filme, encontrou um bug de resultado incorreto (um filme aparecendo com título errado) – algo que não estava previsto – e reportou isso no contexto [60]. Depois, ele testou o botão de tema escuro/claro e outras navegações [64]. Ou seja, **o agente conseguiu identificar comportamentos inesperados e converter em casos de teste**, tudo sozinho. No fim do processo, ele gerou um arquivo de teste Playwright com os passos descobertos e asserções para verificar aqueles comportamentos, e executou para confirmar que passava [62].

Já no modo direcionado, você poderia fornecer à IA uma descrição formal de um caso ("cenário: usuário faz X, espera Y") e ela usará o MCP para realizar X no browser e ver se Y acontece, daí produz o teste. A vantagem sobre simplesmente *gerar o código direto do prompt* é enorme: aqui o teste é baseado na verdade da aplicação, não em suposições. Por exemplo, sem MCP, um modelo pode *alucinar* um seletor `#loginButton` porque é comum, mas talvez seu site use `.btn-primary` – e o teste gerado falharia

imediatamente [55]. Com MCP, o modelo veria o HTML real, encontraria o seletor correto e então geraria o código usando ele. Assim, os testes gerados têm alta chance de já funcionarem de primeira [65].

Benefícios observados:

- **Velocidade na criação de testes:** Uma tarefa que poderia levar horas (escrever vários testes cobrindo fluxos) a IA faz em minutos, após configurada. Ela pode literalmente "navegar" em velocidades sobre-humanas, coletando infos e cuspindo código.
- **Cobertura aumentada:** O agente pode pensar em casos que o desenvolvedor não pensou. Por exemplo, testar input com valor inválido, ou checar a presença de um elemento de erro. Claro que depende de como pedimos, mas um bom agente poderia ter certa criatividade guiada.
- **Atualização de testes facilitada:** Se sua UI mudar, você poderia rodar o agente novamente para atualizar seletores ou passos. Ou mesmo apontar para uma página e dizer "ajuste o teste para a nova UI", e ele via MCP entende as mudanças.
- **Descoberta de bugs em tempo real:** Como visto, durante a geração o agente já está validando a aplicação. Ele pode detectar bugs ou inconsistências antes mesmo de terminar de escrever os testes, servindo como um bot de QA exploratório. No exemplo, encontrou uma falha na busca que passara despercebida [66].

Desafios e considerações:

Apesar do potencial, ainda não estamos no estágio de eliminar o humano do loop. Modelos de IA podem se confundir em aplicações muito complexas ou com muitas possibilidades. É preciso fornecer prompts bem feitos, talvez limitar o escopo ("explore apenas a seção X") para não gastar contexto em demasia. Além disso, a IA pode gerar testes que fazem sentido logicamente mas não são os mais elegantes ou eficientes – o código pode precisar revisão.

Outro ponto: execução de MCP no momento é principalmente via ferramentas específicas (VSCode + Copilot Labs, Cursor IDE, etc.). Ainda não é trivial integrar um fluxo totalmente automatizado de "IA gera testes no CI sozinha". Geralmente há uma interação do desenvolvedor para revisar/aceitar os testes gerados.

Também, um modelo pode **alucinar conclusões erradas** se interpretar mal o resultado de uma ação. Ex: ele clica num botão e espera um texto "Sucesso" que não aparece, e pode seguir tentando outras coisas ou escrever um teste que falha. Ou pode não compreender a lógica de negócio (por ex, talvez confunda nomes semelhantes). Portanto, a supervisão humana e a validação dos testes gerados continuam importantes.

Benefícios e considerações finais

A integração de Playwright com MCP e IA é **uma fronteira promissora na automação de testes**. Recapitulando os benefícios:

- **Aumento de produtividade:** Testes básicos podem ser gerados rapidamente pela IA, liberando QA/devs para tarefas mais complexas. Em vez de escrever boilerplate, o humano pode revisar e aprimorar testes gerados.
- **Manutenção simplificada:** Com a IA "conhecendo" a aplicação real, atualizar testes após mudanças na UI pode ser semiautomático. A IA pode revalidar seletores e fluxos periodicamente.

- **Cobertura exploratória:** IA pode executar cenários aleatórios ou pouco comuns e revelar problemas que casos de teste escritos à mão não cobririam. Essa espécie de *fuzz testing guiado por IA* é um novo paradigma.
- **Aprendizado e documentação:** Os testes gerados pela IA, acompanhados de explicações que ela mesma pode produzir, servem também como documentação viva dos comportamentos do sistema.

Mas é importante temperar o entusiasmo com realidade:

- **IA não substitui estratégia de testes:** Ela é uma ferramenta. É preciso saber formular bons prompts e objetivos. A curadoria dos casos de teste continua necessária – a IA não sabe intrinsecamente quais casos são críticos para o negócio.
- **Cuidado com falsas seguranças:** Um teste gerado pela IA passando não significa que tudo foi testado. Podemos cair numa armadilha de confiar cegamente. Portanto, use a IA para complementar a cobertura, não para assinar embaixo sozinha.
- **Custos e infraestrutura:** Rodar um MCP server e acionar modelos avançados (como GPT-4) pode ter custo financeiro e de setup. Precisa avaliar ROI para cada equipe/projeto.
- **Segurança dos dados:** Ao dar acesso da IA ao seu app, mesmo localmente, pense que informações podem acabar indo para a nuvem (dependendo do modelo usado). Ferramentas offline ou self-hosted mitigam isso, mas é um ponto a se considerar ao testar aplicações confidenciais.

Em conclusão, **Playwright + MCP + IA representa o estado da arte em automação inteligente de testes**. Como visto nos exemplos práticos, já é possível testes "escreverem a si próprios" a partir do uso real da aplicação [67]. Empresas e projetos que adotarem essa abordagem podem ganhar velocidade e robustez no QA. Ainda há espaço para evolução, mas é provável que veremos cada vez mais a IA atuando como co-piloto dos QAs (trocadilho intencional).

Mantendo as boas práticas tradicionais de engenharia de testes junto com essas novas técnicas, obtemos o melhor dos dois mundos: testes sólidos, abrangentes e gerados/atualizados com uma ajuda bem-vinda da inteligência artificial.

Referências: Para se aprofundar, confira o anúncio oficial do Playwright MCP [68], artigos demonstrando geração de testes via IA [67] [60] e guias de melhores práticas de Playwright citados ao longo do texto [5] [39], entre outros. Aproveite o futuro dos testes automatizados!

---

[1] [2] [3] [8] [9] [10] [11] [12] [13] [14] Handling Authentication in Playwright: Login Once, Reuse Across Tests | by Testrig Technologies | Jul, 2025 | Medium
https://testrig.medium.com/handling-authentication-in-playwright-login-once-reuse-across-tests-5b7e9c5d8d71

[4] [6] [7] [15] [16] [17] Authentication | Playwright
https://playwright.dev/docs/auth

[5] [18] [19] [22] [23] [25] [30] [31] [32] 11 Pivotal Best Practices for Playwright
https://autify.com/blog/playwright-best-practices

[20] Playwright Web Scraping Tutorial for 2025
https://oxylabs.io/blog/playwright-web-scraping

[21] 9 Playwright Best Practices and Pitfalls to Avoid - Better Stack
https://betterstack.com/community/guides/testing/playwright-best-practices/

24 26 27 28 29 Continuous Integration | Playwright
https://playwright.dev/docs/ci

33 34 35 36 37 Using Playwright fixtures to skip login pages - DEV Community
https://dev.to/philipfong/using-playwright-fixtures-to-skip-login-pages-3g01

38 40 41 47 52 53 Scalable Web Scraping with Playwright and Browserless (2025 Guide)
https://www.browserless.io/blog/scraping-with-playwright-a-developer-s-guide-to-scalable-undetectable-data-extraction

39 42 43 44 45 46 48 49 50 51 How to Bypass CAPTCHA Using Playwright - ZenRows
https://www.zenrows.com/blog/playwright-captcha

54 55 56 57 58 59 65 68 Generating end-to-end tests with AI and Playwright MCP
https://www.checklyhq.com/blog/generate-end-to-end-tests-with-ai-and-playwright/

60 61 62 63 64 66 67 Letting Playwright MCP Explore your site and Write your Tests - DEV Community
https://dev.to/debs_obrien/letting-playwright-mcp-explore-your-site-and-write-your-tests-mf1