Chapter 3

Explorando Testes End-to-End com Playwright: Um Convite à Automação de Qualidade

Matusalen Costa Alves, Iallen Gábio de Santos Sousa, Mayllon Veras da Silva

Abstract

This chapter explores the discipline of software testing and the importance of End-to-End testing in ensuring the quality of modern web applications. The study introduces the Playwright framework, which provides a modern and robust solution for creating reliable automated tests. It includes a guide to setting up the environment, writing tests using actions and assertions, and analyzing execution reports. Furthermore, it demonstrates the practical application of these concepts through the automation of a "ToDo List" application, highlighting the use of the Page Object Model pattern, code generation with CodeGen, and debugging techniques with the Trace Viewer.

Resumo

Este capítulo explora a disciplina de testes de software e a importância dos testes End-to-End na garantia da qualidade de aplicações web modernas. O estudo apresenta o framework Playwright, que oferece uma solução moderna e robusta para a criação de testes automatizados confiáveis. Inclui um guia para a configuração do ambiente, a escrita de testes com o uso de ações e asserções, e a análise de relatórios de execução. Além disso, demonstra a aplicação prática desses conceitos por meio da automação de uma aplicação "ToDo List", evidenciando o uso do padrão Page Object Model, geração de código com o CodeGen e de técnicas de depuração com o Trace Viewer.

3.1. Introdução

No cenário contemporâneo do desenvolvimento de software, a entrega de produtos digitais de alta qualidade transcendeu o status de diferencial competitivo para se tornar um requisito fundamental para a relevância e o sucesso de qualquer projeto. A complexidade crescente das aplicações, caracterizadas por arquiteturas distribuídas, interfaces interativas e a necessidade de compatibilidade com uma vasta gama de dispositivos e navegadores

impõe desafios às equipes de desenvolvimento. Nesse contexto, a garantia da qualidade deixa de ser uma fase isolada no final do ciclo de vida para se consolidar como uma tarefa contínua e integrada a todas as etapas da produção [Sommerville 2019].

No âmbito dos testes, um teste automatizado é um processo de verificação e validação de software que utiliza ferramentas e scripts para executar rotinas de checagem sem intervenção humana [Sommerville 2019]. Esta abordagem substitui tarefas repetitivas e manuais, o que permite a execução de um grande volume de testes de forma rápida e consistente.

A qualidade é um pilar em Engenharia de Software. Conforme preconiza Robert C. Martin, a conduta de um programador profissional exige a certeza de que o código entregue funciona como esperado, e o conjunto de testes (também conhecido como suíte de testes) automatizados é um dos principais mecanismo para prover essa garantia. A ausência de testes compromete a funcionalidade do produto e introduz o conhecido "débito técnico" que dificulta a manutenção e a evolução do sistema a longo prazo [Martin 2012].

A prática da automação de testes é, portanto, a fundação sobre a qual a agilidade e a sustentabilidade de projetos modernos estão calcadas. Martin Fowler argumenta que a capacidade de refatorar o código (aperfeiçoar seu design interno sem alterar seu comportamento externo) é diretamente dependente da existência de uma rede de testes confiáveis fornecida pela suíte de testes automatizados [Fowler 2020]. Sem essa rede, qualquer alteração se torna arriscada. Essa condição dificulta a melhoria contínua e a capacidade da equipe de responder rapidamente a novas demandas.

A relevância dessa disciplina é ainda mais acentuada pelo advento de novas tecnologias, como os Modelos de Linguagem de Grande Escala (LLMs), que têm sido cada vez mais utilizados para a geração automática de código. Embora essas ferramentas possam acelerar o desenvolvimento, elas também introduzem a necessidade de uma verificação rigorosa, visto que a revisão manual de todo o código gerado é, muitas vezes, impraticável [Vaithilingam et al. 2022]. Nesse novo paradigma, os testes automatizados tornam-se essenciais para validar o software.

Para organizar as estratégias de validação, a indústria adota modelos como a pirâmide de testes. Esta é organizada em uma base larga de testes unitários, uma camada intermediária de testes de integração e, no topo, uma camada mais seleta de testes End-to-End. Estes últimos são cruciais por simularem a jornada completa do usuário; eles validam fluxos do início ao fim e garantem que todos os componentes do sistema funcionem de maneira coesa.

Neste minicurso, materializado na forma deste capítulo, nos concentraremos na criação de testes End-to-End com o uso do Playwright, uma ferramenta moderna mantida pela Microsoft. O Playwright se destaca por oferecer uma solução eficiente, rápida e confiável para a automação de interações em navegadores, o que o torna uma escolha adequada para enfrentar os desafios discutidos.

O restante deste capítulo está organizado da seguinte maneira: a Seção 3.2 aprofunda os conceitos da disciplina de testes de software; a Seção 3.3 apresenta a arquitetura e os diferenciais do ecossistema Playwright; a Seção 3.4 detalha os passos práticos para a configuração e utilização da ferramenta; a Seção 3.5 demonstra, através de um estudo de

caso, a automação de uma aplicação real; e, por fim, a Seção 3.6 conclui o trabalho com uma síntese dos aprendizados e sugestões para estudos futuros.

3.2. A Disciplina de Testes de Software

A verificação e validação são disciplinas fundamentais da Engenharia de Software, responsáveis por assegurar que um sistema computacional atenda às suas especificações e satisfaça as necessidades dos seus usuários. Dentro deste escopo, a prática de testes de software se estabelece como o principal mecanismo para identificar defeitos e avaliar a qualidade de um produto. Esta seção explora os conceitos essenciais desta disciplina, a começar por uma análise da evolução histórica dos testes automatizados. Em seguida, apresentaremos a Pirâmide de Testes, o modelo estratégico mais influente para a organização de suítes de testes, e, por fim, detalharemos o papel crítico dos testes End-to-End, que são o foco deste capítulo.

3.2.1. História e Evolução dos Testes Automatizados

A prática de automatizar testes de software evoluiu em paralelo com as próprias metodologias de desenvolvimento. Nas abordagens mais tradicionais, como o modelo em cascata, os testes eram frequentemente relegados a uma fase final e executados de forma predominantemente manual, um processo lento, repetitivo e suscetível a falhas humanas. A necessidade de otimizar essa etapa impulsionou o surgimento das primeiras ferramentas de automação, muitas baseadas em scripts simples ou em técnicas de captura e repetição de interações do usuário.

A Figura 3.1 ilustra os marcos dessa progressão. A linha do tempo demonstra a transição de um modelo sequencial e manual, associado ao modelo cascata, para ciclos iterativos e ágeis, que culminaram nas práticas de testes contínuos e na exploração de testes autônomos com o avanço da inteligência artificial.

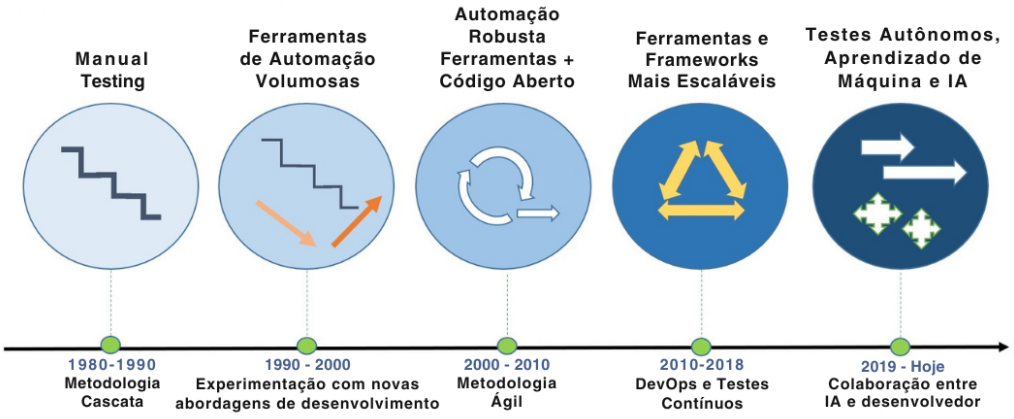

Figure 3.1. Marcos da evolução da automação de testes, da metodologia cascata aos testes contínuos e autônomos.

Uma mudança de paradigma ocorreu com a ascensão das metodologias ágeis no final da década de 1990. A partir de então, os testes passaram a ser vistos não apenas

como uma atividade de verificação final, mas como uma parte intrínseca do processo de desenvolvimento e design. A criação de frameworks de teste, como os da família xUnit, foi fundamental para essa transformação, pois forneceu aos desenvolvedores as ferramentas para escrever testes de forma sistemática e integrar a automação ao processo de codificação.

Essa evolução foi acelerada pela consolidação da cultura DevOps e das esteiras de Integração e Entrega Contínua (CI/CD), que tornaram a automação de testes um requisito essencial. Em um ciclo de vida onde novas versões do software são liberadas com alta frequência, a execução manual de testes de regressão se torna impraticável. A automação passou a ser, portanto, o pilar que garante a segurança e a agilidade necessárias para a inovação contínua [Sommerville 2019].

3.2.2. A Pirâmide de Testes: Estratégias e Níveis

Com a proliferação dos testes automatizados, tornou-se necessária a criação de um modelo estratégico para orientar sua implementação de forma eficiente. O modelo mais amplamente adotado pela indústria é a Pirâmide de Testes, um conceito originalmente proposto por Mike Cohn [Cohn 2009]. A pirâmide é uma heurística visual que descreve a proporção ideal entre diferentes tipos de testes em uma suíte de automação.

A Figura 3.2 representa visualmente este modelo. A largura de cada camada sugere o volume ideal de testes, enquanto os ícones laterais ilustram as características de cada nível: a base é a mais rápida e de menor custo, enquanto o topo é o mais lento e de maior custo.

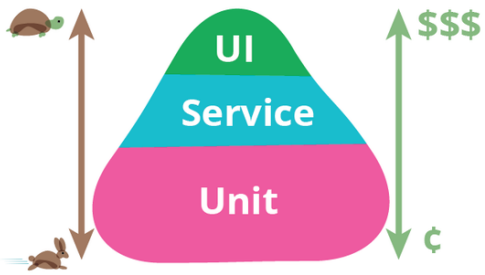

Figure 3.2. Representação da Pirâmide de Testes e suas características de velocidade e custo.

A base da pirâmide é composta pelos Testes de Unidade (*Unit Tests*). Estes testes verificam os menores componentes do sistema, como uma função ou uma classe, de forma isolada. São caracterizados por sua alta velocidade de execução e baixo custo de manutenção. Por testarem a lógica de negócio em seu nível mais granular, eles devem constituir a maior parte da suíte de testes.

A camada intermediária é composta pelos Testes de Integração, por vezes chamados de Testes de Serviço (*Service Tests*). O objetivo destes testes é verificar a interação entre dois ou mais componentes do sistema, como a comunicação entre um serviço de aplicação e o banco de dados. São mais lentos e complexos que os testes de unidade, pois envolvem múltiplos componentes, e, por isso, devem existir em menor número.

No topo da pirâmide, encontram-se os Testes End-to-End. Estes testes validam um fluxo completo do sistema sob a perspectiva do usuário final, o que geralmente envolve a interação com a interface gráfica. Conforme detalhado por Martin Fowler, embora os testes End-to-End ofereçam a maior confiança sobre o funcionamento do sistema, eles também são os mais lentos, frágeis e caros para manter. Portanto, a estratégia da pirâmide recomenda que eles sejam utilizados de forma seletiva, focados nos fluxos mais críticos do negócio [Fowler 2012].

3.2.3. O Papel Crítico dos Testes End-to-End

Um teste End-to-End (E2E) é uma técnica de teste que simula um cenário de usuário real do início ao fim. Diferentemente dos testes de unidade e integração, que operam em camadas mais baixas e com partes isoladas do código, um teste E2E interage com o sistema através da sua interface de usuário, da mesma forma que um cliente faria. O seu escopo abrange todas as camadas da arquitetura da aplicação, desde a interface gráfica no navegador até os serviços de backend e o banco de dados.

A Figura 3.3 ilustra de forma esquemática este processo. A interação do usuário ocorre na camada mais externa, a interface (representada pelo formulário), e desencadeia uma série de operações que atravessam os diversos componentes da arquitetura do sistema (representados pelos cubos), como serviços de aplicação e bancos de dados. Um teste E2E bem-sucedido valida a integridade de todo esse percurso.

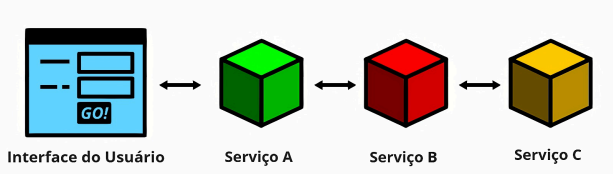

Figure 3.3. Exemplo do fluxo de um teste End-to-End, da interface aos componentes do sistema.

O principal valor dos testes E2E reside na sua capacidade de fornecer um alto grau de confiança de que a aplicação, como um todo, está funcionando corretamente e atendendo aos requisitos de negócio. Ao validar fluxos completos, como um processo de cadastro de usuário ou a finalização de uma compra em um e-commerce, os testes E2E garantem que a integração entre os diversos componentes do sistema está operando de forma coesa.

Contudo, a implementação de testes E2E apresenta desafios. A sua natureza integrada os torna inerentemente mais lentos, pois dependem de operações de rede, renderização de interface e acesso a banco de dados. Eles também são mais frágeis, ou seja, podem falhar devido a pequenas alterações na interface do usuário que não necessariamente representam um defeito na lógica de negócio. Por essa razão, a sua criação e manutenção exigem um planejamento cuidadoso e a aplicação de padrões de projeto específicos, como discute Gerard Meszaros em sua obra sobre padrões de teste [Meszaros 2007]. Apesar desses desafios, os testes E2E são uma camada indispensável em uma estratégia de qualidade, pois estão entre os poucos capazes de validar a experiência completa do usuário.

3.3. O Ecossistema Playwright

Após a fundamentação teórica sobre a disciplina de testes de software, esta seção direciona o foco para a ferramenta central deste capítulo: o Playwright. O objetivo é realizar uma imersão técnica em seu ecossistema, a fim de demonstrar por que ele se estabelece como uma solução moderna e eficaz para os desafios da automação de testes E2E.

Iniciaremos com uma análise detalhada do conceito e da arquitetura que garantem a velocidade e a confiabilidade da ferramenta. Em seguida, exploraremos suas funcionalidades-chave e os diferenciais que otimizam a experiência de desenvolvimento. Por fim, faremos uma análise comparativa que posiciona o Playwright em relação a outras ferramentas consolidadas no mercado.

3.3.1. Conceito e Arquitetura

O Playwright é um framework de automação de código aberto, mantido pela Microsoft, projetado para atender às demandas do desenvolvimento de aplicações web modernas. Seu objetivo é fornecer uma API única, coesa e poderosa para a automação dos três principais motores de renderização de navegadores: Chromium (utilizado por Google Chrome e Microsoft Edge), WebKit (utilizado pelo Apple Safari) e Firefox. A filosofia do projeto se concentra em três pilares: velocidade, capacidade e, principalmente, confiabilidade, para eliminar a instabilidade que historicamente afeta os testes de interface de usuário.

A Figura 3.4 ilustra a arquitetura da ferrameta. No lado do cliente, os testes podem ser escritos em diversas linguagens, como TypeScript, JavaScript e Python. Esses scripts enviam instruções ao servidor Playwright, que as traduz em comandos específicos para o protocolo de depuração do navegador. A conexão WebSocket garante uma comunicação bidirecional e eficiente, permitindo que o Playwright controle o navegador com precisão e receba eventos em tempo real. Essa estrutura é a base para muitas das funcionalidades avançadas da ferramenta, como a capacidade de interceptar requisições de rede e a execução de testes em múltiplos contextos de forma isolada.

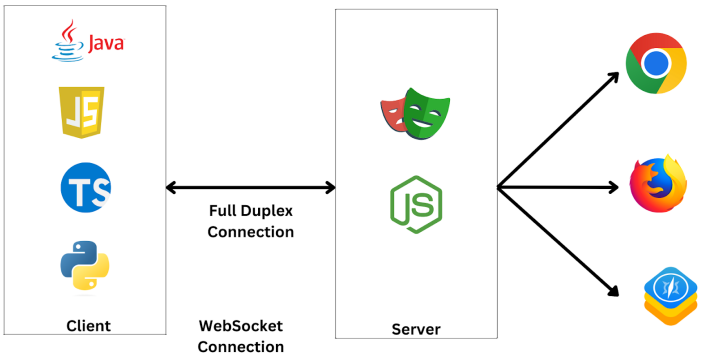

Figure 3.4. Visão geral da arquitetura de comunicação do Playwright.

O principal diferencial técnico do Playwright reside em sua arquitetura. Diferentemente de soluções mais antigas que dependem de protocolos baseados em HTTP para a

comunicação entre o script de teste e o navegador, o Playwright opera em um modelo fora do processo. Nele, o script de teste se comunica com um servidor Node.js que, por sua vez, envia comandos aos navegadores por meio de uma conexão WebSocket persistente. Essa comunicação direta e de baixa latência evita pontos de falha e gargalos de desempenho, o que resulta em uma execução de testes mais rápida e estável [Microsoft 2025].

3.3.2. Funcionalidades-Chave e Diferenciais

O Playwright se distingue por um conjunto de funcionalidades nativas projetadas para otimizar a experiência de desenvolvimento e aumentar a confiabilidade dos testes. Estes recursos abordam desafios comuns na automação de testes, como a instabilidade, a dificuldade de depuração e a complexidade na criação de novos scripts. A seguir, detalharemos as ferramentas que compõem esses diferenciais.

3.3.2.1. Trace Viewer: Depuração de Viagem no Tempo

Um dos maiores desafios dos testes E2E é a depuração. Testes que falham em um ambiente de integração contínua podem ser difíceis de diagnosticar, pois o desenvolvedor não tem acesso ao estado do navegador no momento da falha. O Playwright soluciona este problema com o Trace Viewer, uma de suas ferramentas mais poderosas.

Ao final de uma execução de testes, o Playwright gera um relatório em HTML, como o apresentado na Figura 3.5, que exibe o resultado de cada teste executado em diferentes navegadores. Este relatório centraliza os resultados e serve como ponto de partida para a análise.

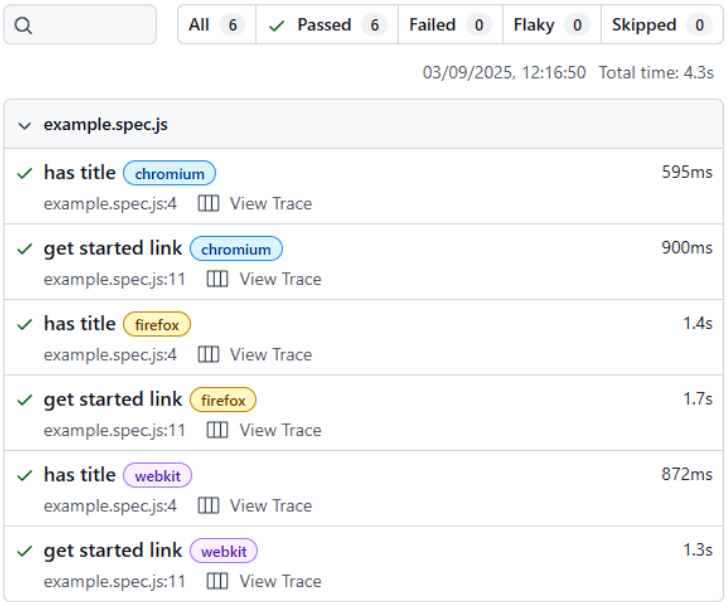

Figure 3.5. Relatório de testes do Playwright com a lista de execuções.

A partir do relatório, é possível inspecionar cada teste individualmente para visualizar os passos executados, como ganchos (**hooks**), ações e asserções, conforme ilustrado na Figura 3.6. Para uma análise mais profunda, o relatório oferece acesso ao histórico completo da execução.

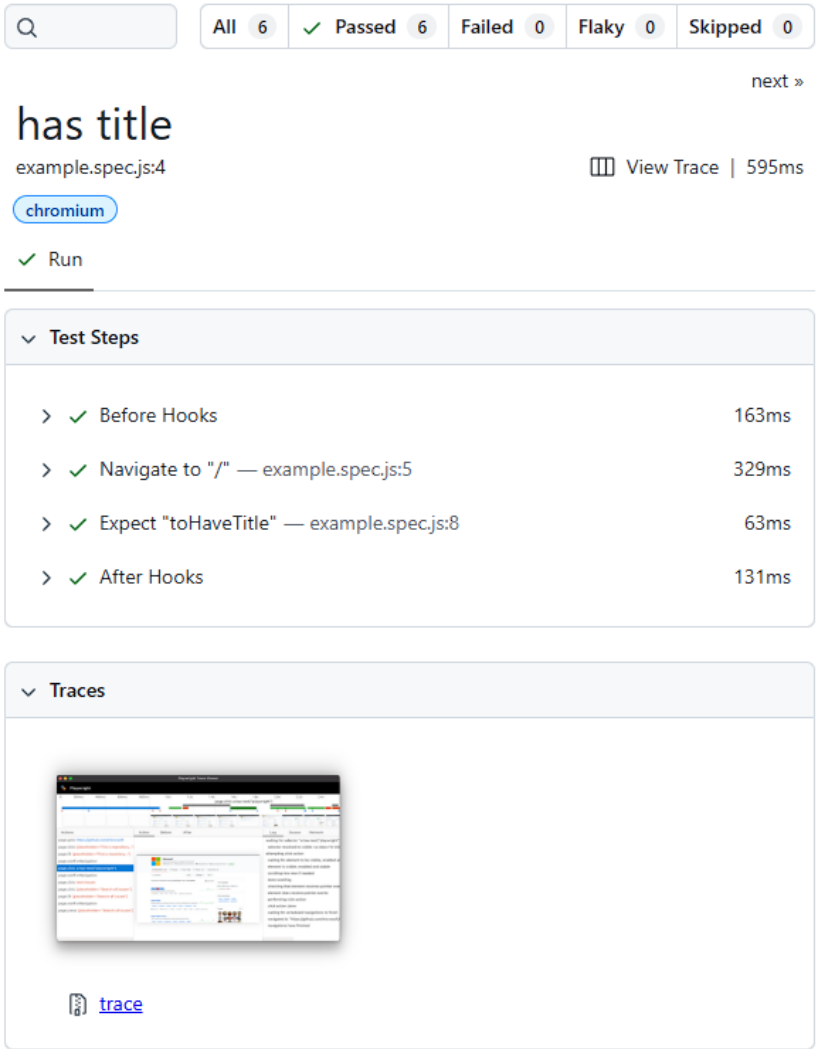

Figure 3.6. Visualização detalhada de um teste específico no relatório.

A interface do Trace Viewer, exibida na Figura 3.7, proporciona uma experiência de depuração de "viagem no tempo". Ela captura um traço completo da execução do teste, permitindo que o desenvolvedor navegue pela linha do tempo e inspecione o estado da aplicação em cada momento. A ferramenta exibe a lista de ações, o snapshot do DOM antes e depois de cada ação, logs do console e requisições de rede. Esse nível de detalhe contextualizado reduz drasticamente o tempo necessário para identificar a causa raiz de uma falha [Microsoft 2025].

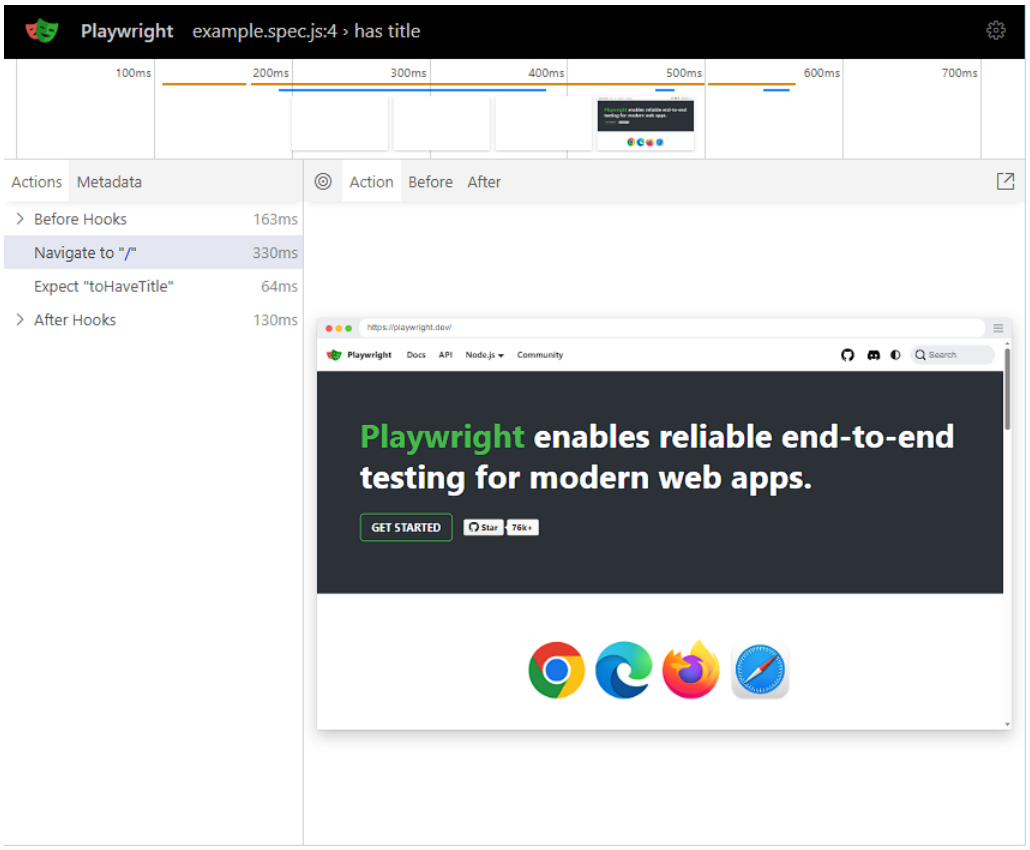

Figure 3.7. Interface principal do Trace Viewer com a linha do tempo, ações e o snapshot do DOM.

3.3.2.2. Auto-waits: O Fim da Instabilidade

Um dos problemas mais comuns na automação de testes é a instabilidade (flakiness), onde testes falham de forma intermitente sem uma causa aparente. Frequentemente, a raiz do problema está em condições de corrida (race conditions), nas quais o script de teste tenta interagir com um elemento da interface antes que ele esteja totalmente carregado ou pronto para receber uma ação. A solução tradicional para isso envolvia a inserção de pausas fixas ou esperas explícitas no código, o que o torna mais lento e complexo.

O Playwright resolve este problema de forma nativa com seu mecanismo de esperas automáticas, conhecido como auto-wait. Antes de executar qualquer ação, como um clique ou o preenchimento de um campo, a ferramenta realiza uma série de verificações para garantir que o elemento alvo esteja pronto para a interação. Essas checagens incluem verificar se o elemento está visível, estável (sem animações em andamento) e habilitado para receber eventos.

Essa inteligência nativa simplifica enormemente o código do teste. A Figura 3.8 demonstra a clareza de um script de teste em Playwright, onde os comandos são diretos e focados na intenção do usuário. Por trás dessa simplicidade, a ferramenta gerencia a

sincronização.

Locator Call Log Errors Console Network 31 Source Attachments

example.spec.js

```
1 // @ts-check
2 import { test, expect } from '@playwright/test';
3
4 test('has title', async ({ page }) => {
5   await page.goto('https://playwright.dev/');
6
7   // Expect a title "to contain" a substring.
8   await expect(page).toHaveTitle(/Playwright/);
9 });
10
11 test('get started link', async ({ page }) => {
12   await page.goto('https://playwright.dev/');
13
14   // Click the get started link.
15   await page.getByRole('link', { name: 'Get started' }).click();
16
17   // Expects page to have a heading with the name of Installation.
18   await expect(page.getByRole('heading', { name: 'Installation' })).toBeVisible();
19 });
20
```

Figure 3.8. Exemplo de código de teste em Playwright, com comandos diretos.

Na Figura 3.9, extraída do log de uma execução, mostra que, mesmo em um comando como ‘page.goto‘, o Playwright aguarda ativamente por eventos específicos da página, como o evento "load". Esse mecanismo de espera automática é um dos principais motivos da alta confiabilidade dos testes escritos com Playwright [Microsoft 2025].

Locator Call Log Errors Console Network 31 Source Attachments

navigating to "https://playwright.dev/", waiting until "load" 317ms

Figure 3.9. Log de execução de um comando, que evidencia a espera por um evento específico da página.

3.3.2.3. Codegen: Geração de Testes Acelerada

Para acelerar a criação de novos testes e diminuir a barreira de entrada para novos usuários, o Playwright inclui a ferramenta Codegen. Ao ser iniciada, ela abre uma janela de navegador junto com uma janela "Playwright Inspector". À medida que o usuário interage com a aplicação no navegador (clica em botões, preenche formulários, navega entre páginas), o Codegen grava essas ações e as traduz em tempo real para código Playwright.

A Figura 3.10 demonstra a ferramenta em ação. Na parte superior, o navegador exibe a aplicação web sendo testada, enquanto na parte inferior, o "Playwright Inspector" exibe o código gerado a partir das interações. Esta funcionalidade é extremamente útil tanto para aprender a sintaxe da API do Playwright quanto para criar rapidamente o

esqueleto de um novo teste, que pode então ser refinado e adaptado pelo desenvolvedor [Microsoft 2025].

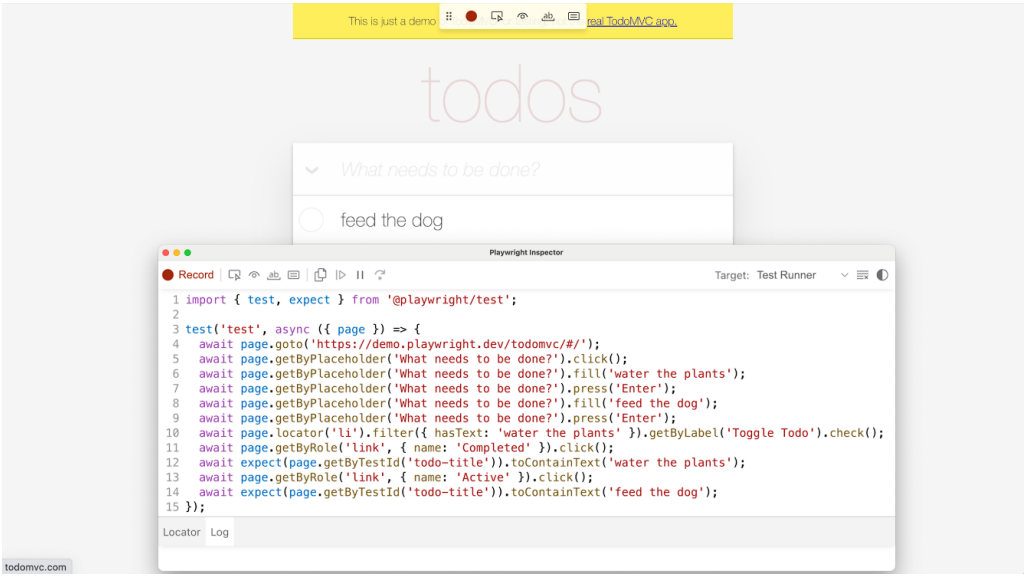

Figure 3.10. Ferramenta Codegen em execução, com o navegador e a janela do inspetor que gera o código em tempo real.

3.3.2.4. Suporte Multi-Navegador e Execução Paralela

Finalmente, um dos maiores diferenciais do Playwright é seu suporte nativo e de primeira classe aos três principais motores de renderização de navegadores: Chromium, Firefox e WebKit. Isso permite que as equipes de desenvolvimento validem o comportamento de suas aplicações em um ambiente verdadeiramente multiplataforma, o que garante uma experiência de usuário consistente.

A capacidade de executar o mesmo conjunto de testes em diferentes navegadores é evidenciada no relatório de testes, como já demonstrado na Figura 3.5, onde cada teste foi validado nos três motores. Adicionalmente, o Playwright foi projetado para executar testes em paralelo por padrão, distribuindo-os entre múltiplos processos de trabalho. Essa abordagem reduz drasticamente o tempo total de execução da suíte de testes, um fator crucial em ambientes de integração contínua.

3.3.3. Análise Comparativa

Para compreender o valor e os diferenciais do Playwright, é útil posicioná-lo no ecossistema de ferramentas de automação de testes E2E. Embora existam diversas soluções, o mercado foi historicamente dominado pelo Selenium, com o Cypress emergindo como uma alternativa moderna popular. Cada uma dessas ferramentas possui uma arquitetura e uma filosofia distintas, que resultam em diferentes vantagens e desvantagens. As informações apresentadas na Tabela 3.1 foram compiladas a partir da documentação oficial de cada ferramenta [Microsoft 2025, Selenium 2025, Cypress 2025].

Table 3.1. Análise Comparativa de Frameworks de Testes E2E.

Critério	Playwright	Selenium	Cypress
Suporte a Linguagens	JavaScript, Java, Python, .NET	JavaScript, Java, C#, Python, etc.	JavaScript
Driver do Navegador	Não requer driver	Requer um driver para cada navegador	Não requer driver
Relatórios Nativos	Sim	Não	Sim
Recursos de Depuração	Ferramentas nativas e depuração "time-traveling"	Não possui ferramentas de depuração nativas	Ferramentas nativas e depuração "time-traveling"
Esperas Automáticas	Sim	Não	Sim

A tabela destaca as diferenças na experiência do desenvolvedor. O Selenium, com seu vasto suporte a linguagens, oferece grande flexibilidade, mas exige uma configuração mais complexa, como a gestão de drivers específicos para cada navegador. Por outro lado, Cypress e Playwright proporcionam uma experiência mais integrada, com relatórios nativos, esperas automáticas e ferramentas de depuração avançadas que simplificam a escrita e a manutenção dos testes.

Já o Playwright, diferente das duas outras alternativas, se posiciona de forma vantajosa ao combinar o amplo suporte a linguagens, similar ao do Selenium, com as conveniências modernas encontradas no Cypress, estabelecendo-se como uma solução que equilibra poder e facilidade de uso.

3.4. Instalando o Playwright no Projeto

Após a apresentação conceitual do ecossistema Playwright, esta seção inicia a jornada prática. O objetivo é guiar o leitor através dos passos fundamentais para a criação e execução de um projeto de testes do início ao fim.

Iniciaremos com as instruções para a instalação e configuração completa do ambiente de desenvolvimento.

Em seguida, faremos uma análise detalhada da anatomia de um arquivo de teste, explicando seus componentes essenciais, como comandos e asserções.

Por fim, demonstraremos como executar a suíte de testes e analisar os relatórios de resultados gerados pela ferramenta.

3.4.1. Configuração do Ambiente

O primeiro passo para utilizar o Playwright é a preparação do ambiente de desenvolvimento. O principal pré-requisito é ter o Node.js instalado, que inclui o gerenciador de pacotes npm. Com o ambiente Node.js pronto, a instalação do Playwright é realizada por meio de um único comando que inicia um assistente de configuração interativo.

Para iniciar um novo projeto ou adicionar o Playwright a um projeto existente, o comando a ser executado no terminal é o apresentado no Código 3.1.

```
npm init playwright@latest
```

Listing 3.1. Comando de inicialização do Playwright via npm.

Ao executar este comando, o assistente de configuração fará algumas perguntas para personalizar o projeto, como:

- A escolha entre TypeScript ou JavaScript para a escrita dos testes.
- O nome do diretório onde os testes serão armazenados (o padrão é *tests*).
- A adição de um fluxo de trabalho para o GitHub Actions, útil para a integração contínua.
- A confirmação para instalar os navegadores necessários (Chromium, Firefox e WebKit).

Após a conclusão do assistente, o Playwright criará uma estrutura de arquivos e diretórios no projeto. Essa estrutura inicial, conforme detalhado na documentação e demonstrado no Código 3.2, organiza os arquivos de forma lógica e inclui exemplos para facilitar os primeiros passos do desenvolvedor [Microsoft 2025].

```
.
|-- playwright.config.ts
|-- package.json
|-- tests/
|   |-- example.spec.ts
|-- tests-examples/
    |-- demo-todo-app.spec.ts
```

Listing 3.2. Estrutura de arquivos gerada pela instalação do Playwright.

O arquivo *playwright.config.ts*, presente na estrutura do Código 3.2, é o centro de controle do projeto, onde são definidas configurações como os navegadores alvo, tempos de espera e formatos de relatório. O diretório *tests/* é o local padrão para os testes do projeto, enquanto o *tests-examples/* contém exemplos mais elaborados que demonstram diferentes funcionalidades da ferramenta.

3.4.2. Anatomia de um Teste: Comandos e Asserções

Um teste em Playwright, em sua essência, é uma sequência de duas operações fundamentais: a execução de ações para simular a interação do usuário com a página e a verificação do estado da aplicação por meio de asserções. A estrutura de um teste é declarada dentro de uma função *test()*, que recebe como argumento a fixture *page*, um objeto que representa uma única aba no navegador e serve como a principal interface para a automação.

O Código 3.3 apresenta um arquivo de teste completo, *example.spec.ts*, que contém dois cenários de teste distintos. O primeiro verifica se a página possui o título esperado, e o segundo valida a navegação ao clicar em um link e checar o conteúdo da nova página.

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

test('has title', async ({ page }) => {
  await page.goto('https://playwright.dev/');
```

```
5   // Espera que o título da página contenha o texto "Playwright
        ".
6   await expect(page).toHaveTitle(/Playwright/);
7 });
8
9 test('get started link', async ({ page }) => {
10  await page.goto('https://playwright.dev/');
11  // Clica no link com o nome "Get started".
12  await page.getByRole('link', { name: 'Get started' }).click();
13  // Espera que a página possua um cabeçalho com o nome "
        Installation".
14  await expect(page.getByRole('heading', { name: 'Installation'
        })).toBeVisible();
15 });
```

Listing 3.3. Exemplo de um arquivo de teste completo em Playwright.

As ações em um teste começam, geralmente, com a navegação para uma URL, como visto na linha *await page.goto('https://playwright.dev/')*. Após o carregamento da página, o teste interage com seus elementos. Para encontrar esses elementos, o Playwright utiliza a API de **Locators**, que são objetos que representam uma forma de encontrar um ou mais elementos na página a qualquer momento. No exemplo do Código 3.3, *page.getByRole('link', name: 'Get started')* é um locator que encontra um link com o texto específico. Uma vez que o elemento é localizado, ações como *.click()* podem ser executadas.

As asserções são utilizadas para verificar se a aplicação se comporta como o esperado após uma série de ações. O Playwright utiliza a função *expect()*, que, combinada com seus validadores assíncronos, aguarda até que uma condição seja atendida ou um tempo limite seja atingido. Essa característica torna os testes mais resilientes e menos suscetíveis a falhas por tempo. Nos exemplos, *expect(page).toHaveTitle(/Playwright/)* e *expect(locator).toBeVisible()* são asserções que pausam a execução do teste até que a condição de validação seja verdadeira.

É importante notar que o Playwright garante o isolamento total entre os testes. Cada função *test* recebe uma instância de *page* que pertence a um *BrowserContext* exclusivo, o que é equivalente a um perfil de navegador completamente novo. Isso significa que cookies, armazenamento local e sessões não são compartilhados entre os testes, o que garante que a execução de um não possa interferir no resultado do outro.

3.4.3. Executando Testes e Analisando Resultados

Uma vez que os testes são escritos, o próximo passo é executá-los para verificar o comportamento da aplicação. O Playwright oferece uma interface de linha de comando (CLI) para gerenciar a execução dos testes de forma flexível e poderosa.

O comando fundamental para executar toda a suíte de testes é apresentado no Código 3.4. Por padrão, este comando executa todos os arquivos de teste encontrados no projeto, em paralelo e em modo *headless* (sem abrir uma interface gráfica do navegador) para todos os navegadores configurados no arquivo *playwright.config.ts*.

```
npx playwright test
```

Listing 3.4. Comando para executar a suíte de testes.

Após a execução, os resultados são exibidos diretamente no terminal, como mostra o Código 3.5, com um resumo de quantos testes passaram ou falharam em cada navegador.

```
Running 6 tests using 3 workers
  6 passed (4.3s)
```

Listing 3.5. Exemplo de saída do terminal após a execução dos testes.

A execução pode ser personalizada com diversos parâmetros. Por exemplo, a flag *–headed* executa os testes em um navegador com interface gráfica visível, enquanto a flag *–project* permite especificar um único navegador, como em *npx playwright test –project chromium*.

O principal artefato para a análise dos resultados é o Relatório HTML. Ele fornece um painel interativo para explorar os resultados de cada teste. O relatório é aberto automaticamente quando há falhas, mas pode ser acessado a qualquer momento com o comando do Código 3.6.

```
npx playwright show-report
```

Listing 3.6. Comando para visualizar o relatório HTML.

A interface principal do relatório, já apresentada na Figura 3.5, permite uma análise visual completa dos resultados. A partir dela, é possível filtrar testes por status (aprovado, falhou, etc.) e por navegador, além de inspecionar os passos de cada teste e acessar o traço completo da execução para uma depuração aprofundada, como será detalhado na próxima seção.

Após a exploração dos conceitos teóricos e das ferramentas práticas, esta seção consolida o aprendizado através da aplicação completa dos conhecimentos em um projeto real. O objetivo é guiar o leitor na construção de uma suíte de testes E2E para uma aplicação web do tipo "ToDo List", um exemplo clássico utilizado para a demonstração de tecnologias de frontend. Seguiremos um fluxo de trabalho estruturado: primeiramente, definiremos o escopo e os cenários de teste; em seguida, implementaremos os testes utilizando o padrão de projeto Page Object Model; e, por fim, demonstraremos como depurar e analisar os resultados.

3.5. Estudo de Caso: Todo List com Definição do Escopo e Cenários de Teste

A aplicação "ToDo List" a ser testada possui uma interface simples para o gerenciamento de tarefas. O usuário pode adicionar, editar, marcar como concluída e excluir tarefas. Nosso objetivo é criar um conjunto de testes automatizados que valide estas funcionalidades críticas, garantindo a integridade da experiência do usuário.

Para este capítulo, implementaremos um conjunto representativo dos cenários de teste mais importantes. A suíte de testes completa, com validações adicionais, estará disponível no repositório do projeto para consulta. Os cenários que abordaremos são:

- **Carregamento da Página:** Verificar se a aplicação carrega corretamente e exibe seus elementos principais.
- **Adição de Tarefa:** Validar a criação de uma ou mais tarefas.
- **Edição de Tarefa:** Assegurar que uma tarefa existente pode ser editada.
- **Exclusão de Tarefa:** Verificar se uma tarefa pode ser removida da lista.
- **Marcar Tarefa como Concluída:** Validar a funcionalidade de marcar e desmarcar uma tarefa.
- **Validação de Tarefa Duplicada:** Garantir que a aplicação lida corretamente com a tentativa de adicionar uma tarefa com o mesmo texto de uma já existente.
- **Validação de Tarefa Vazia:** Verificar se o sistema impede a adição de uma tarefa sem texto.

A Figura 3.11 exibe a interface principal da aplicação que será o objeto de nosso estudo de caso.

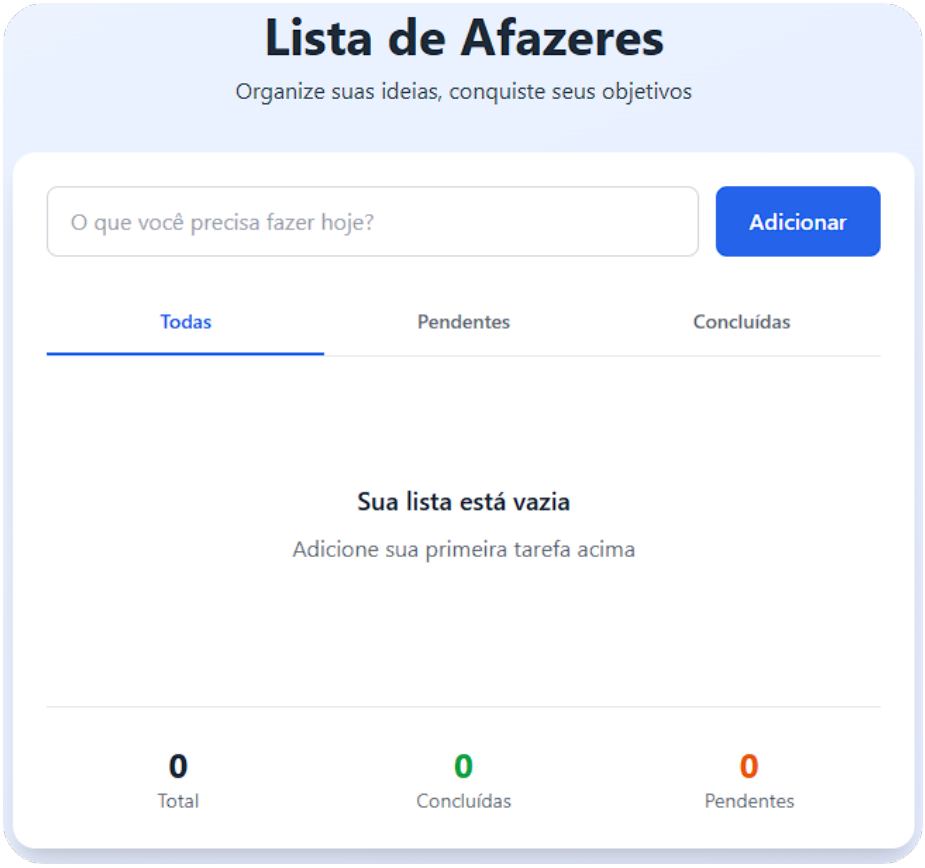

Figure 3.11. Interface da aplicação "ToDo List" utilizada no estudo de caso.

3.5.1. Implementação Prática com Page Object Model

Para implementar os cenários de teste de forma organizada e de fácil manutenção, utilizaremos o padrão de projeto Page Object Model (POM). O POM é uma técnica de design que consiste em criar uma classe para cada página da aplicação, encapsulando os detalhes da interface do usuário. O principal benefício desta abordagem é a separação de responsabilidades: a classe Page Object lida com a complexidade de encontrar e interagir com os elementos da página, enquanto o arquivo de teste se concentra apenas na lógica e nas asserções do cenário [Microsoft 2025].

A Figura 3.12 ilustra este padrão. As páginas da aplicação web são mapeadas para classes Page Object, que por sua vez são utilizadas pelos scripts de teste. Essa camada de abstração desacopla os testes da estrutura interna da interface, tornando-os mais resilientes a mudanças no HTML.

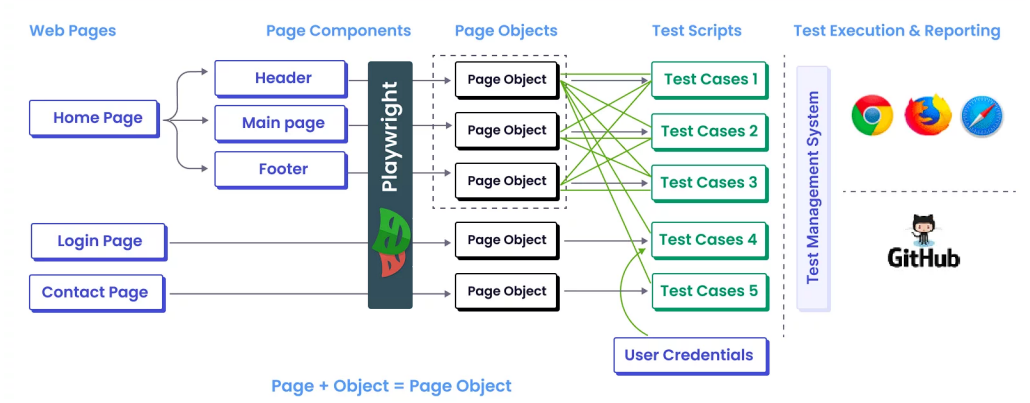

Figure 3.12. Diagrama do fluxo de trabalho com o padrão Page Object Model.

Para a nossa aplicação, criamos a classe *TodoPage*. O código completo desta classe está disponível no repositório do projeto para consulta [Alves 2025]; aqui, destacaremos seus componentes essenciais. Primeiramente, a classe centraliza todos os localizadores de elementos em seu construtor, como demonstrado no Código 3.7. Se um seletor na interface do usuário mudar no futuro, só precisaremos atualizá-lo neste único local.

```
// Trecho de TodoPage.js
constructor(page) {
  this.page = page;

  // Localizadores dos elementos principais
  this.taskInput = page.locator('#taskInput');
  this.addButton = page.locator('#addButton');
  this.taskList = page.locator('#taskList');
  // ... outros localizadores disponiveis no repositorio
}
```

Listing 3.7. Trecho do construtor da classe TodoPage com os localizadores.

Em seguida, a classe expõe métodos de alto nível que representam as ações do usuário. O Código 3.8 mostra o método *addTask*, que esconde os detalhes de implementação (preencher o campo e clicar no botão) por trás de uma única ação com um nome claro e intuitivo.

```
// Trecho de TodoPage.js
async addTask(taskText) {
  await this.taskInput.fill(taskText);
  await this.addButton.click();
}
```

Listing 3.8. Exemplo de um método de ação na classe TodoPage.

Com a classe *TodoPage* pronta, o arquivo de teste (*todo.spec.js*), apresentado no Código 3.9, se torna extremamente limpo e legível. Ele se concentra no fluxo do cenário e nas validações, utilizando a instância do Page Object para orquestrar as interações com o navegador. Note como o teste lê quase como um roteiro de caso de uso, sem a desordem de seletores de CSS.

```
import { test, expect } from '@playwright/test';
import { TodoPage } from './TodoPage'; // Importa a classe

test.describe('Gerenciamento de Tarefas', () => {
  let todoPage;

  test.beforeEach(async ({ page }) => {
    todoPage = new TodoPage(page);
    await todoPage.goto();
  });

  test('deve adicionar uma nova tarefa', async () => {
    const taskText = 'Comprar pao';
    await todoPage.addTask(taskText);
    await expect(todoPage.taskList).toContainText(taskText);
  });

  // ... outros cenarios de teste disponiveis no repositorio
});
```

Listing 3.9. Arquivo de teste que utiliza a classe TodoPage para implementar os cenarios.

3.5.2. Depuração e Análise Avançada com o Trace Viewer

Um dos aspectos mais desafiadores na manutenção de uma suíte de testes é a investigação de falhas. Um teste pode falhar por diversos motivos, desde um defeito real na aplicação até um problema no próprio script de teste. Para demonstrar como diagnosticar um problema de forma eficiente, simularemos um cenário comum: um teste que falha devido a um localizador incorreto.

O Código 3.10 apresenta uma variação do nosso teste de adicionar tarefa. Nele,

introduzimos um erro proposital na linha 22: o seletor para o botão de adicionar (‘[data-testid="adicionar-button"]‘) está incorreto; o correto seria ‘[data-testid="add-button"]‘.

```
1  test('Teste de Falha: Seletor incorreto do botao adicionar',
       async () => {
2      const taskText = 'Minha primeira tarefa';
3      await todoPage.locators.taskInput.fill(taskText);
4
5      // ERRO PROPOSITAL: Usa seletor incorreto para o botao
6      // Original: [data-testid="add-button"]
7      // Incorreto: [data-testid="adicionar-button"]
8      await todoPage.page.locator('[data-testid="adicionar-button
           "]').click();
9
10     // Estas verificacoes vao falhar porque a tarefa nao foi
           adicionada
11     await expect(todoPage.getTaskCount()).toBe(1);
12 });
```

Listing 3.10. Exemplo de um teste com uma falha proposital (seletor incorreto).

Ao executar este teste, o Playwright irá falhar. O primeiro passo da nossa análise é inspecionar o Relatório HTML. A Figura 3.13 exibe a tela de resultado para o teste que falhou. O relatório imediatamente nos informa o tipo de erro (*TimeoutError*, indicando que a ferramenta esperou por um elemento que nunca apareceu) e aponta para a linha exata do código que causou a falha.

Para uma análise mais profunda, clicamos em "View Trace". Dentro do Trace Viewer, a aba "Actions", mostrada na Figura 3.14, destaca em vermelho a ação exata que falhou. Neste caso, o comando *.click()*. Isso permite que o desenvolvedor isole imediatamente o ponto problemático da execução.

Para entender o motivo da falha, a aba "Errors", na Figura 3.15, fornece o log detalhado. A mensagem "*TimeoutError: waiting for locator('[data-testid="adicionar-button"]')*" confirma que o Playwright esgotou o tempo de espera porque não conseguiu encontrar o elemento com o seletor especificado. Através deste processo de análise, do geral para o específico, o desenvolvedor pode diagnosticar rapidamente que o problema é um erro no seletor, em vez de um defeito na funcionalidade da aplicação.

3.6. Conclusão

Ao longo deste capítulo, realizamos uma jornada completa pelo universo dos testes E2E, desde seus fundamentos teóricos até a sua aplicação prática com o framework Playwright. Iniciamos por estabelecer a disciplina de testes de software como um pilar essencial da engenharia de software moderna, explorando sua evolução histórica e o modelo estratégico da Pirâmide de Testes. Com essa base, aprofundamos no ecossistema Playwright, analisando sua arquitetura, suas funcionalidades-chave e seu posicionamento em relação a outras ferramentas do mercado.

A parte prática do capítulo guiou o leitor desde a configuração inicial de um pro-

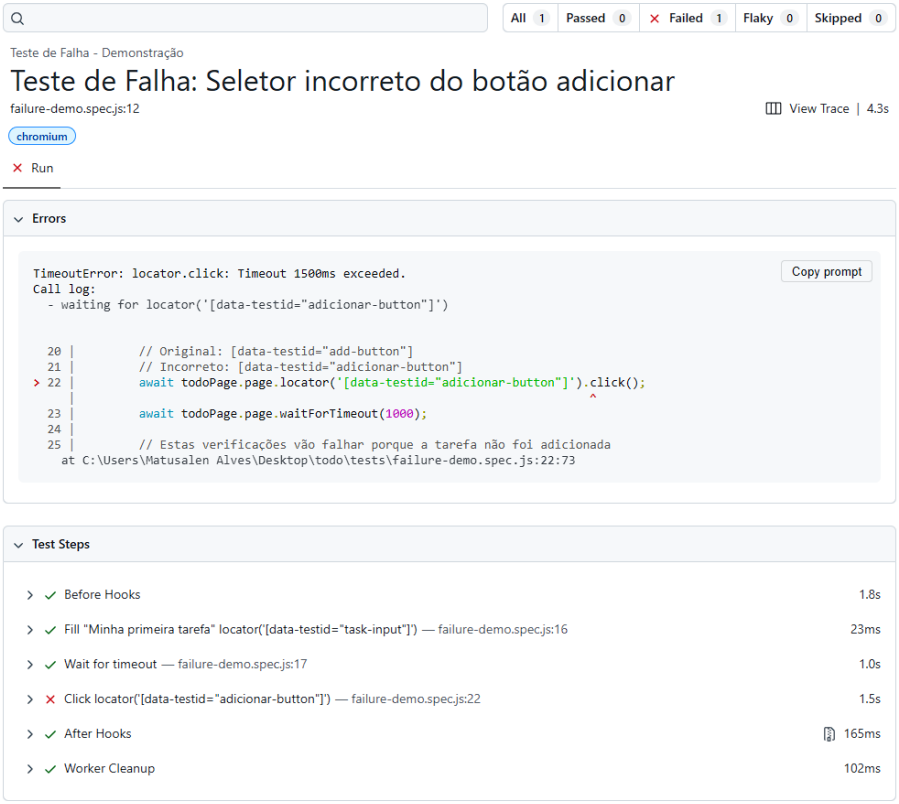

Figure 3.13. Relatório HTML exibindo o teste que falhou e o erro correspondente.

jeto até a escrita e execução de testes, culminando em um estudo de caso detalhado. Na implementação para a aplicação "ToDo List", demonstramos como escrever os testes e como estruturá-los de forma fácil, aplicando o padrão de projeto POM. Além disso, abordamos um aspecto crucial do dia a dia do desenvolvimento: a depuração de testes, mostrando como o Trace Viewer acelera a identificação e a correção de falhas.

A principal mensagem deste capítulo é que a automação de testes E2E, com o auxílio de ferramentas modernas como o Playwright, é uma prática acessível e de alto impacto para qualquer equipe de desenvolvimento. Recursos como as esperas automáticas e as ferramentas de depuração visual não são apenas conveniências, mas soluções diretas para os desafios de instabilidade e complexidade que historicamente tornaram os testes de UI um processo custoso.

Para os leitores que desejam aprofundar seus conhecimentos, o próximo passo natural é explorar as outras capacidades que o ecossistema Playwright oferece, como o suporte nativo para testes de API, a implementação de testes de regressão visual e a funcionalidade de testes de componentes. Além disso, a integração da suíte de testes a um pipeline de Integração e Entrega Contínua (CI/CD), como o GitHub Actions, é uma etapa fundamental para automatizar completamente o processo de garantia de qualidade. A documentação oficial do Playwright permanece como o recurso mais completo para o

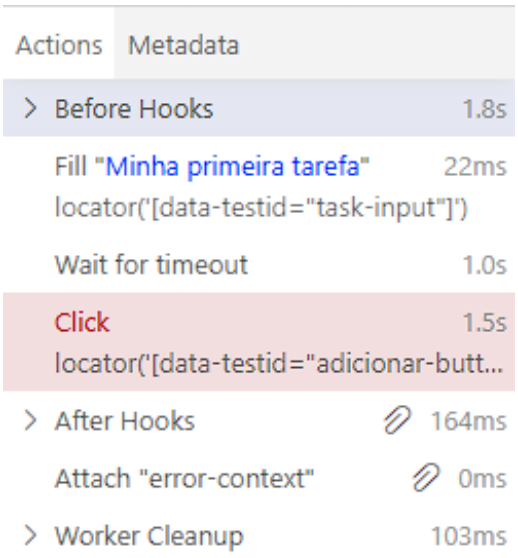

Figure 3.14. Aba "Actions" do Trace Viewer, destacando a etapa que falhou.

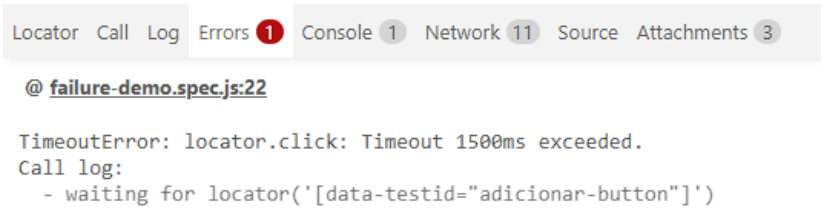

Figure 3.15. Aba "Errors" do Trace Viewer com o log detalhado da falha.

estudo contínuo, e o repositório do nosso estudo de caso serve como um exemplo prático e funcional de referência.

References

[Alves 2025] Alves, M. C. (2025). todo-codec: Aplicação para o minicurso de testes e2e com playwright. `https://github.com/watusalen/todo-codec`. GitHub. Acesso em: 2025-10-20.

[Cohn 2009] Cohn, M. (2009). *Succeeding with Agile: Software Development Using Scrum*. Addison-Wesley Professional.

[Cypress 2025] Cypress (2025). Cypress documentation. `https://docs.cypress.io/`. Acesso em: 2025-10-20.

[Fowler 2012] Fowler, M. (2012). The test pyramid. `https://martinfowler.com/bliki/TestPyramid.html`. Acesso em: 2025-10-20.

[Fowler 2020] Fowler, M. (2020). *Refatoração: Aperfeiçoando o design de códigos existentes*. Novatec Editora.

[Martin 2012] Martin, R. C. (2012). *O codificador limpo: um código de conduta para programadores profissionais*. Alta Books.

[Meszaros 2007] Meszaros, G. (2007). *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley Professional.

[Microsoft 2025] Microsoft (2025). Playwright documentation. `https://playwright.dev/docs/intro`. Acesso em: 2025-10-20.

[Selenium 2025] Selenium (2025). Selenium documentation. `https://www.selenium.dev/documentation/`. Acesso em: 2025-10-20.

[Sommerville 2019] Sommerville, I. (2019). *Engenharia de Software*. Pearson Brasil.

[Vaithilingam et al. 2022] Vaithilingam, P., Zhang, T., and Glassman, E. L. (2022). Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*. ACM.