



Pós-Graduação em Automação de Testes de Software

AUTOMAÇÃO DE TESTES WEB

Curso de Pós-Graduação *Lato Sensu* de
Automação de Testes de Software em parceria
entre Julio de Lima Consultoria e Treinamentos
de Testes e Qualidade de Software e a
Faculdade VINCIT.

Dados comerciais:

Julio de Lima Consultoria e Treinamentos de Testes e
Qualidade de Software LTDA
Rua Antônio Rosa Felipe, 203
Jardim Universo
Araçatuba, São Paulo
CEP 16056-808
CNPJ 18.726.544/0001-09
IE 177.428.549.110

Contato:

comercial@juliodelima.com.br
(18) 99793-6246

OBJETIVO GERAL

Capacitar os alunos a compreenderem e aplicarem os conceitos fundamentais e avançados relacionados à automação de testes em aplicações web, fornecendo assim uma base sólida de conhecimentos teóricos e práticos para que os alunos possam desenvolver e implementar testes automatizados de alta qualidade, eficiência e confiabilidade em projetos web.

CONTEÚDO

Introdução a aplicações web	6
HTML para construção das páginas	8
Introdução	8
Estrutura do HTML	9
CSS para estilização das páginas	11
Javascript para interação com usuário e servidor	13
Seletores CSS em automação de testes web	15
Introdução	15
Estratégias para o uso de seletores	15
Seletores XPATH em automação de testes web	18
Introdução	18
Estratégias para uso de seletores	18
XPath Axis	19
Funções XPath: text() e contains()	20
Seletores por Contexto	22
Introdução	22
FUNDAMENTOS DE AUTOMAÇÃO DE TESTES WEB	24
Introdução	24
Arquiteturas	28
Introdução ao Cypress	31
Esperas Explícitas e Implícitas	41
Screenshots	43
Execução de Testes Headless	44
Modularização e Separação de Responsabilidades	46
Relatórios de execução de testes	48
Boas Práticas em Automação de Testes Web	50

Execução de Testes na Nuvem	53
Comboboxes, Janelas e Drag and Drop	59
IA e Automação de Testes Web	63
Exercícios	65
Referências	66

INTRODUÇÃO A APLICAÇÕES WEB

Uma aplicação web é um software que pode ser acessado e executado diretamente em um navegador, como o Google Chrome, Firefox, Safari, Internet Explorer. As aplicações web são diferentes dos programas tradicionais, que precisam ser instalados para serem utilizados. Elas funcionam conectando-se a servidores remotos, o que elimina a necessidade de download local.

Podem funcionar através da internet ou aplicativos desenvolvidos utilizando tecnologias web HTML, CSS e Javascript. Pode ser executado a partir de um **servidor HTTP** ou localmente, no dispositivo do usuário. Por exemplo, ao acessar um e-commerce, a página que você visualiza é gerada por um computador conectado à internet, que executa as regras de negócio antes de enviar o conteúdo para o seu navegador.

A função de um **servidor web** é receber uma requisição e enviar uma resposta de volta ao **cliente**. O navegador permite ao usuário **solicitar um recurso**, e o servidor responde com: páginas html, imagens, documentos, dados formatados, que são exibidos para o usuário. Um **cliente** refere-se a qualquer dispositivo ou software que faz solicitações ao servidor web. no contexto de aplicações web, o cliente é o navegador (browser).

Nessa disciplina iremos direcionar nosso foco a testes de ponta a ponta de aplicações web, visto que mais de 50% das vagas em testes de software pedem essa habilidade aos testadores.

HTML PARA CONSTRUÇÃO DAS PÁGINAS

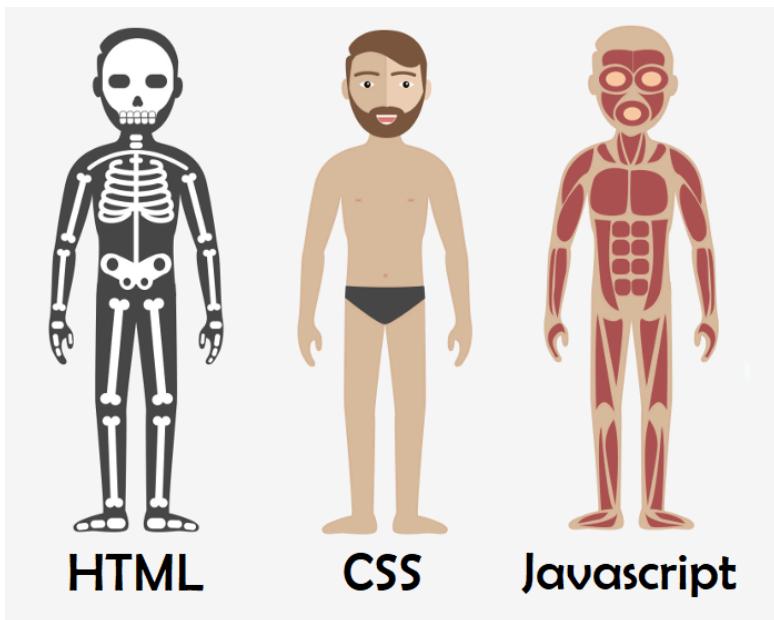


Figura 2: Imagem ilustrativa da relação entre as tecnologias web

Introdução

HTML é a sigla para HyperText Markup Language, que significa Linguagem de Marcação de Hipertexto. É uma linguagem de computador que define a estrutura e o significado do conteúdo de uma página web. Ele foi criado

pelo físico britânico Tim Berners-Lee na década de 1990, com o objetivo de organizar e estruturar arquivos.

É considerado o componente básico da web, e é utilizado para construir páginas e inserir conteúdo como imagens e vídeos. O código HTML é interpretado pelos navegadores, e possui uma estrutura e sintaxe próprias.

Estrutura do HTML

Elementos HTML são definidos entre < e >. Cada elemento tem uma tag de abertura e, geralmente, uma tag de fechamento. Exemplo:

```
<tag> conteúdo </tag>
```

Elementos HTML podem ter atributos que definem informações adicionais sobre o elemento, como *id*, *class*, *name*, *placeholder*. Esses atributos são fundamentais para um tema que vamos ver mais a frente, os seletores.

A semântica HTML também é um conceito fundamental, que trata da utilização correta das tags HTML para representar o conteúdo de uma página web de forma clara e significativa. A semântica é importante por vários motivos, como acessibilidade, indexação por mecanismos de busca e legibilidade do código.

Por exemplo, para escrever um título devemos utilizar a tag <h1>, enquanto um parágrafo deve ser envolvido pela tag <p>. As *tags semânticas* nos ajudam a estruturar o conteúdo da nossa página de forma mais significativa.

Alguns exemplos de tags semânticas:

- header - utilizamos para representar o cabeçalho de um documento
- nav - utilizamos para representar um conjunto de links de navegação, que são criados com elementos ``, `` e `<a>`
- figure - utilizamos para representar a inserção de uma figura
- footer - utilizamos para representar o rodapé de um documento

Para descrever a aparência e apresentação de uma página web é utilizada outra tecnologia, o CSS.

CSS PARA ESTILIZAÇÃO DAS PÁGINAS

CSS é a sigla para Cascading Style Sheets ou Folhas de Estilo em Cascata. É uma linguagem de estilo, responsável por definir como os elementos de um documento HTML são exibidos em tela. A estrutura básica de uma declaração CSS possui 3 elementos principais: seletor, propriedade e valor da propriedade, como na ilustração abaixo:

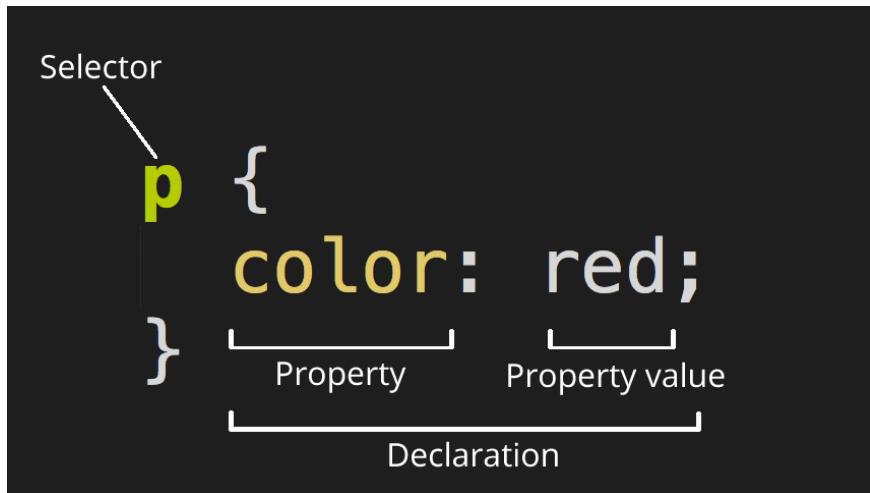


Figura 3: Estrutura básica de uma declaração css

O seletor funciona como um filtro, usado para selecionar um conjunto de elementos. Neste conjunto, são aplicadas regras que podem especificar: comprimento, cor, dentre outras dezenas de possibilidades visuais que podem ser aplicadas à tela.

No contexto de automação web, Seletores CSS funcionam exatamente da mesma forma para o "filtro", mas com outra finalidade: encontrar e interagir com elementos da página. Ou seja, ao invés de aplicar um estilo, normalmente interagimos de alguma forma com aquele elemento usando uma ferramenta de automação para: inserir texto, clicar, deslizar, verificar conteúdo.

JAVASCRIPT PARA INTERAÇÃO COM USUÁRIO E SERVIDOR

Javascript é uma linguagem de programação que permite "dar vida" a páginas web. Sempre que a aplicação web faz mais do que mostrar conteúdos estáticos, provavelmente há Javascript envolvido. Animações, mapas, consulta de informações, tudo isso passa pelo dinamismo que o Javascript nos fornece.

No contexto de automação de testes web, o Javascript tem um papel crucial pois lida tanto com a interação e experiência do usuário, quanto com a comunicação entre o cliente (navegador) e o servidor.

Javascript e interação e experiência do usuário

O Javascript, como visto anteriormente, é o que permite interatividade nas páginas web. Ele é responsável por controlar o comportamento dos elementos da interface, gerenciar eventos como cliques, interações, navegação, etc.

Javascript e a interação com o servidor

O Javascript também é usado para fazer requisições ao servidor, sem precisar recarregar a página. Essa abordagem é fundamental pois permite atualizar o conteúdo da página de forma dinâmica. Existem algumas técnicas comuns de comunicação com o servidor: AJAX, Fetch API e WebSockets. Vejamos um pouco mais sobre cada uma:

AJAX (Asynchronous Javascript and XML) é um dos métodos mais antigos de fazer requisições ao servidor de forma assíncrona, e é usado para buscar ou enviar dados sem recarregar a página.

Fetch API é uma *API* mais moderna, usada para fazer requisições de forma assíncrona, substituindo o AJAX. É mais simples de usar e mais poderosa, e isso faz com que seja amplamente adotada em projetos web modernos.

WebSockets permitem uma comunicação bidirecional contínua entre o navegador e o servidor. São muito utilizados em aplicações em tempo real, como chats, jogos e atualizações de dados ao vivo.

Todas estas características têm um impacto significativo na automação de testes web. As ferramentas e os testes automatizados devem ser capazes de lidar com as particularidades e abordagem de cada aplicação quanto ao uso de Javascript.

Ferramentas como Selenium, Cypress e Playwright, oferecem recursos para simular interações de usuário, interceptar requisições, lidar com o assincronismo e esperas, para garantir que o teste simula o mais próximo possível de um usuário real interagindo com a aplicação.

SELETORES CSS EM AUTOMAÇÃO DE TESTES WEB

Introdução

Os seletores CSS, como vistos anteriormente, são essenciais para automação de testes de aplicações web. Eles permitem selecionar elementos HTML para executar alguma ação com a ferramenta.

Estratégias para o uso de seletores

Existem pelo menos 3 estratégias principais que são usadas para o mapeamento de elementos usando seletores css: id, classe e atributos. Vejamos um exemplo:

```
1 <div id="toast" class="toast" name="toast">Texto</div>
```

Selecionar elementos via ID é uma das estratégias mais robustas, porque um ID deve ser único por página. Se o ID for bem definido e não for alterado com frequência, é a escolha ideal. O id no exemplo é representado pelo atributo "id" seguido do valor "toast". Quando um elemento html possui este atributo, podemos identificá-lo com a sintaxe: hashtag + valor do atributo id. Da seguinte forma: #toast

Selecionar elementos por classe é útil quando vários elementos compartilham a mesma aparência ou comportamento. Não é tão específico quanto o id, já que múltiplos elementos podem compartilhar a mesma classe. A classe no exemplo é representada pelo atributo "class" seguido do valor "toast". Quando um elemento html possui este atributo, podemos identificá-lo com a sintaxe: ponto + valor do atributo class. Da seguinte forma: .toast

Seletores de atributo são úteis para selecionar elementos com valores específicos de atributos e, diferente de id ou class, não possuem um caractere para simplificar sua escrita. Sua sintaxe pode ser aplicada para quaisquer atributos disponíveis no HTML. No exemplo acima, podemos usar o name, que é um atributo diferente de classe e id. A sintaxe fica: [atributo=valor]. Da seguinte forma: [name=toast].

Em automação de testes, é comum utilizar atributos dedicados a testes como data-test-id, data-test, test-id, etc. A ideia é criar um atributo para fins de testes, menos propenso a alterações até mesmo que o id.

Curingas de seletores por atributos

Os seletores por atributo podem ser aprimorados usando caracteres "curinga" como ^, \$ e *. Esses caracteres oferecem uma forma mais flexível de selecionar elementos que compartilham parte de um valor de atributo, sem que o valor completo precise ser especificado. Vamos detalhar como cada um desses operadores funciona

O caractere "^" (starts with - começa com)

Seleciona elementos cujo valor de atributo **começa com** uma sequência de caracteres específica.

O caractere "\$" (ends with - termina com)

Seleciona elementos cujo valor de atributo **termina com** uma sequência de caracteres específica.

O caractere "*" (contains - contém)

Seleciona elementos cujo valor de atributo **contém** uma sequência de caracteres específica.

SELETORES XPATH EM AUTOMAÇÃO DE TESTES WEB

Introdução

No contexto da automação de testes de aplicações web, o XPath é uma alternativa poderosa e flexível aos seletores CSS para mapear e interagir com elementos HTML. O XPath (XML Path Language) permite navegar pela estrutura hierárquica de um documento XML (ou HTML) de maneira mais detalhada e avançada, especialmente quando os seletores CSS não oferecem a precisão ou flexibilidade necessária.

Estratégias para uso de seletores

Selecionar elementos usando XPath

Selecionar elementos no XPath é similar ao uso de atributos no CSS, mas a sintaxe é diferente. O XPath utiliza o operador @ para referenciar atributos, exemplo:

- HTML: `<div id="toast">Texto</div>`
- XPath: `//*[@id='toast']`

Essa expressão XPath localiza qualquer elemento (*) cujo atributo id seja igual a "toast". A sintaxe é a mesma para qualquer atributo, seja uma *class*, *name*, *placeholder*, etc.

XPath Axis

O XPath Axis é um conjunto de operadores que permitem navegar pela árvore de elementos HTML em diferentes direções. Ele permite que você selecione elementos com base em sua relação hierárquica com outros elementos. É similar ao conceito de navegação por contexto que vimos anteriormente, só que com comandos nativos do XPath.

Principais Axes:

1. child: Seleciona os elementos filhos diretos de um nó.

Exemplo: `//div[@id='parent']/child::div` → Seleciona todos os div filhos diretos do elemento com id="parent".

2. parent: Seleciona o elemento pai do nó atual.

Exemplo: `//div[@id='child']/parent::*` → Seleciona o pai do elemento com id="child".

3. descendant: Seleciona todos os descendentes de um nó (filhos, netos, bisnetos, etc.).

Exemplo: `//div[@id='ancestor']/descendant::p` → Seleciona todos os elementos p descendentes do elemento com id="ancestor".

4. ancestor: Seleciona todos os antecessores de um nó (pais, avós, etc.).

Exemplo: //div[@id='child']/ancestor::div → Seleciona todos os elementos div antecessores do elemento com id="child".

5. following-sibling: Seleciona os elementos irmãos que aparecem depois do nó atual.

Exemplo: //div[@id='start']/following-sibling::div → Seleciona os elementos div que são irmãos e estão depois do div com id="start".

Funções XPath: `text()` e `contains()`

O XPath oferece várias funções para ajudar a criar seletores mais avançados. Duas das funções mais comuns são `text()` e `contains()`.

Função `text()`

A função `text()` é usada para selecionar elementos com base no conteúdo textual.

- Exemplo de HTML: <button>Salvar</button>
- XPath: //button[text()='Salvar']

Essa expressão XPath localiza todos os botões cujo texto é exatamente "Salvar".

Função `contains()`

A função `contains()` é amplamente usada para encontrar elementos que contêm parte de um texto em um atributo ou no texto do próprio elemento. É similar ao comando `.contains`, do Cypress. Exemplo:

- HTML: <button class="btn-primary">Salvar</button>
- XPath para atributo: //button[contains(@class, 'btn')]
- XPath para texto: //button[contains(text(), 'Salvar')]

O primeiro XPath localiza botões cuja classe contém "btn", e o segundo localiza botões cujo texto contém "Salvar".

XPath Absoluto, Relativo e Exato

O XPath oferece diferentes maneiras de localizar elementos. As três abordagens mais comuns são o XPath absoluto, XPath relativo e o XPath exato. Cada uma delas tem suas características, vantagens e desvantagens, dependendo da complexidade e da estabilidade da estrutura do documento.

XPath Absoluto

O XPath absoluto é o caminho completo, partindo da raiz do documento (/html) até o elemento desejado, navegando por cada nó do DOM, independentemente de sua profundidade.

XPath Relativo

O XPath relativo não depende da estrutura completa da página. Ele começa a partir de qualquer nó intermediário, geralmente usando a notação // para indicar que não importa onde o elemento está, desde que atenda a certas condições (como ter um atributo específico ou estar em um determinado nível da hierarquia).

XPath Exato

O XPath exato é uma abordagem que combina o uso de atributos e propriedades do elemento (como id, class, name, ou texto) para encontrar o elemento desejado de forma

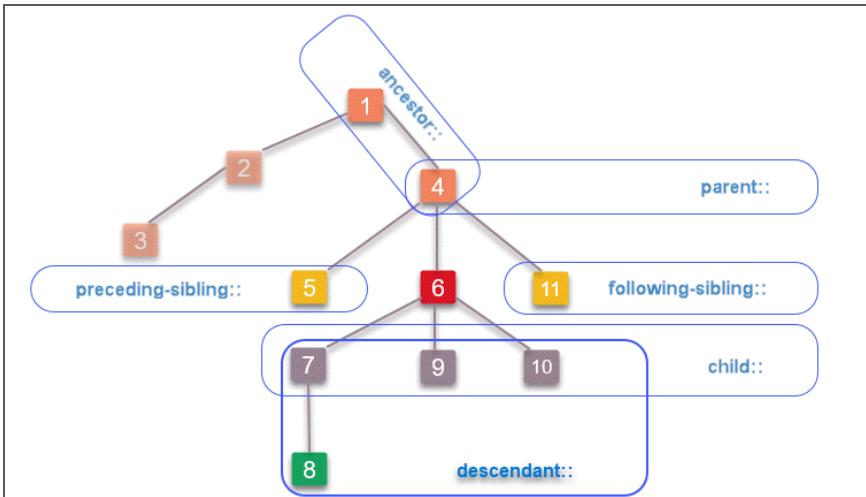
precisa. A ideia é fazer uso de identificadores únicos ou características específicas do elemento para garantir a localização correta.

SELETORES POR CONTEXTO

Introdução

Assim como no XPath, algumas ferramentas possuem recursos para navegação na árvore de elementos. O Cypress e o Playwright são dois exemplos. Eles permitem navegar para elementos em um nível acima, abaixo, no mesmo, dentre outras variações.

De certa forma, é muito similar aos recursos do XPath através dos *Axis*. Com isso, quando a ferramenta disponibiliza tais comandos, o uso de XPath é minimizado.



Vejamos a imagem acima como exemplo. O elemento de número 6 é o elemento "âncora", e a partir dele é que podemos navegar para outras partes da página, usando este como elemento de referência.

Vamos a alguns exemplos de navegação contextual usando o Cypress, ferramenta que usada para a prática de automação de testes web nessa disciplina:

- **parent** - navega para o elemento "pai" do elemento âncora. Na imagem, representado pelo elemento de número 4.
- **siblings** - lista os elementos "irmãos" do elemento âncora. Na imagem, representado pelos elementos de número 5 e 11.
- **children** - lista ou navega para os elementos "filhos" do elemento âncora. Na imagem, representado pelos elementos de número 7, 9 e 10.

Essa estratégia de navegação é especialmente útil quando o elemento que está sendo mapeado não possui algum

seletor, mas possui outro elemento próximo que facilite uma busca por contexto.

FUNDAMENTOS DE AUTOMAÇÃO DE TESTES WEB

Introdução

Navegadores

Os navegadores são o ambiente onde as aplicações web são executadas e testadas. Quando falamos em automação de testes web, o navegador é essencial, pois é através dele que o usuário interage.

Existem dois tipos de navegadores que normalmente são utilizados em automação de testes: navegadores convencionais e navegadores *headless*.

Os navegadores convencionais, são os navegadores que os usuários normalmente utilizam, como Google Chrome, Mozilla Firefox, Microsoft Edge, Safari.

Já o "navegador headless" é uma versão do navegador convencional que não tem interface gráfica, onde as interações são simuladas em um segundo plano sem abrir uma janela do navegador visível. Vamos falar com mais profundidade sobre *testes headless* nos próximos capítulos da apostila.

Durante os testes, esses navegadores podem ser controlados por ferramentas de automação para simular a interação do usuário com a aplicação. A forma como serão controlados depende da *arquitetura* da aplicação que está sendo usada.

Testar uma aplicação em diferentes navegadores é essencial para garantir a compatibilidade em diferentes plataformas. Entretanto, é importante avaliar quais as possíveis diferenças que se espera encontrar para evitar desperdício de testes.

Responsivo e Adaptativo

Um dos aspectos a se considerar durante o desenvolvimento e testes de aplicações web modernas é garantir que o *design da interface* seja adaptável e funcional em diferentes dispositivos. Isso envolve conhecer algumas abordagens como a de layout responsivo e layout adaptativo.

O **layout responsivo** utiliza *media queries* e um design fluido para adaptar automaticamente a interface a

qualquer resolução de tela, independente do dispositivo (desktop, tablet, celular).

O objetivo principal de um layout responsivo é garantir que o conteúdo se ajuste de forma dinâmica conforme o tamanho da tela, mantendo a experiência do usuário e funcionalidades sem impacto.

Em automação de testes web, o "tamanho da tela" normalmente se refere a porção visível do navegador. A esta parte visível, damos o nome de *viewport*. As ferramentas costumam ter configurações ou comandos que permitem *manipular* ou *consultar* essa característica do navegador para adequar os testes.

O **layout adaptativo**, por outro lado, utiliza *breakpoints* predefinidos para carregar um design fixo, mas adaptado a diferentes tamanhos de tela. Na prática, é como se o usuário tivesse que dar um "F5" na página para visualizar ela na versão adequada.

Ao invés de adaptar o layout de forma fluida, ele carrega um layout específico para cada tamanho de tela. Ou seja, o site pode ter diferentes versões para cada dispositivo e em alguns casos isso pode ser percebido nas URLs, com "m." antes do host em aplicações que possuem versões para mobile.

Isso é particularmente um desafio em automação de testes pois o projeto, assim como as aplicações, precisa se adequar para funcionar em diferentes resoluções. De forma resumida, existem duas possibilidades: um projeto para cada resolução; ou condicionais em código para cada componente.

CSR vs SSR

Outro aspecto extremamente importante é compreender como uma aplicação renderiza suas páginas, pois isso impacta diretamente a estratégia de testes e as possibilidades de automação.

As duas principais abordagens para renderização de páginas em aplicações web modernas são: Client-Side Rendering (CSR); e Server-Side Rendering (SSR). Vejamos um pouco mais sobre as características de cada uma delas.

No *CSR*, o conteúdo da página web é carregado e processado no lado do cliente (navegador). Isso significa que, quando um usuário acessa uma aplicação, o servidor envia tudo que a aplicação precisa como HTML, CSS e Javascript.

Esse conjunto, como vimos anteriormente, é responsável por renderizar a página diretamente no navegador, solicitando ao servidor dados ou arquivos adicionais à medida em que o usuário navega.

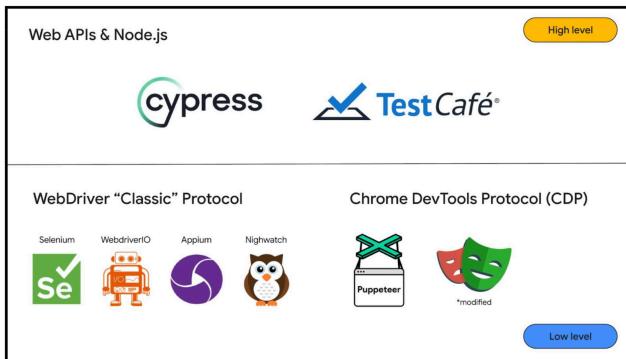
Em resumo, a renderização da página ocorre no navegador. O servidor entrega apenas a estrutura base e os arquivos que geram o conteúdo da página.

No *SSR*, a maior parte do processamento da página ocorre diretamente no servidor. Ou seja, o servidor gera a página HTML completa, incluindo o conteúdo dinâmico e envia "pronto" para o navegador.

Dessa forma, quando o navegador exibe a página, o conteúdo já foi totalmente renderizado no lado do servidor, poupando boa parte do processamento que ocorre no lado do *cliente*.

Existem ainda algumas aplicações que adotam uma abordagem híbrida, combinando CSR e SSR e para obter o melhor de cada uma. Isso é comum em aplicações que usam frameworks como o *Next.js*.

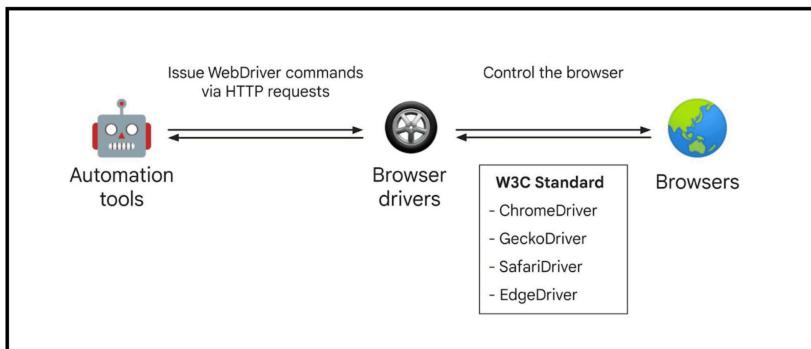
Arquiteturas



As ferramentas de automação de testes para aplicações web possuem, pelo menos, 3 abordagens arquiteturais para interagir com a aplicação que está executando no navegador.

Conhecer sobre a arquitetura das ferramentas nos ajuda a compreender suas virtudes e limitações, por isso, vamos conhecer cada uma com mais profundidade.

Protocolo WebDriver



Usado por ferramentas como Selenium, Appium e Nightwatch, o protocolo WebDriver é o mais popular no contexto de automação de testes web.

O WebDriver é um protocolo HTTP que envia comandos do *script* de testes para o *navegador*. O WebDriver é um padrão W3C.

Para usar este protocolo, você precisa de um servidor proxy que traduz os comandos e os executa no ambiente alvo, ou seja, no navegador.

Para automação web, esse proxy é o driver do próprio navegador. Cada navegador fornece um Driver em que o WebDriver usa para se comunicar.

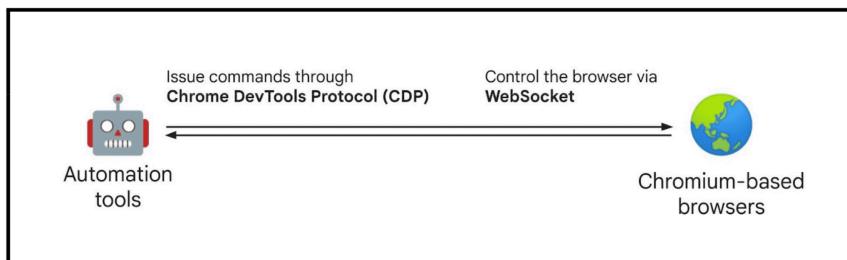
Exemplos de Drivers:

- ChromeDriver, do Google Chrome
- GeckoDriver, do Firefox
- SafariDriver, do Safari
- EdgeDriver, do Microsoft Edge

Sendo um padrão W3C, sua compatibilidade com navegadores é alta. Também é possível implementar a *comunicação* com o driver em diferentes linguagens

de programação, uma vez que ele é o responsável por "traduzir" os comandos para o navegador.

Protocolo ou Padrão CDP



Usado por ferramentas como Puppeteer e Playwright, o protocolo CDP (Chrome Devtools Protocol) é uma abordagem diretamente conectada ao navegador.

O CDP foi criado para atender a todas as necessidades de *depuração* e oferece suporte a mais controles de baixo nível em relação ao WebDriver.

Ele se comunica com a API do navegador, ou seja, os comandos são executados usando diretamente no navegador usado para testes.

De certa forma, é similar ao protocolo WebDriver, só que ao invés de um *Driver* usamos a própria API do navegador para executar comandos.

O *CDP* usa conexão WebSocket para comunicar o script de testes ao navegador. Esse protocolo é mantido pelo Chrome e é parcialmente implementado pelo Firefox.

O Firefox inclusive, está [descontinuando o suporte ao CDP](#) a partir da versão 129 (Agosto de 2024) em prol

do uso de uma proposta mais moderna, o WebDriver BiDi.

> *Cada navegador possui seu próprio CDP?*

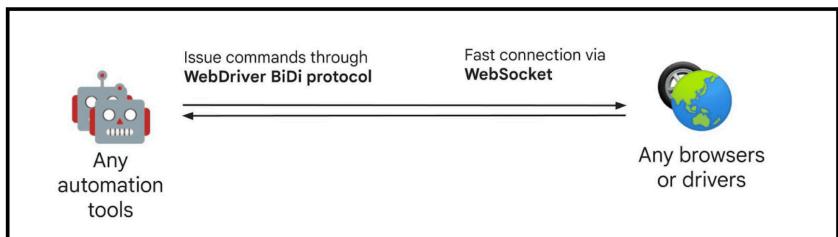
Para navegadores Chromium, CDP; Firefox usa o mesmo protocolo com uma nomenclatura diferente, chamam de "[Remote Protocol](#)".

Javascript no navegador como aplicação

Usado por ferramentas como Cypress e TestCafe, essa é uma abordagem que executa a ferramenta e os testes diretamente no navegador.

Os testes e a aplicação compartilham o mesmo processo e domínio, portanto, o teste pode acessar diretamente o DOM, os objetos da aplicação e as APIs do navegador.

Protocolo WebDriver BiDi



O WebDriver BiDi é um novo protocolo padrão de automação de navegador que está em desenvolvimento e busca combinar o melhor do WebDriver "clássico" e o CDP.

O WebDriver BiDi promete comunicação bidirecional, tornando-o rápido por padrão, e vem com controle de baixo nível. A visão por trás do WebDriver Bidi é

permitir que você escreva testes usando qualquer ferramenta, em qualquer navegador ou driver, oferecendo flexibilidade.

INTRODUÇÃO AO CYPRESS

Nesta disciplina, vamos utilizar o Cypress para ilustrar e exercitar os conceitos de aula.

Pré-requisitos:

- NodeJS nas versões 18, 20, 22 ou superior;
- Visual Studio Code
- Extensões do VSCode (opcional)
 - ES6 Mocha Snippets
 - Cypress Snippets (Andrew Smith)

Iniciando um projeto com Cypress

Passo 01: crie uma pasta, de modo tradicional ou usando linha de comandos.

```
 samuellucas in pgats
◦ > mkdir automacao-web-pgats-i └
```

Passo 02: abra a pasta criada no VSCode

```
samuellucas in      /pgats/automacao-web-pgats-i
◦› code .
```

Passo 03: abra o terminal do VSCode e inicialize um projeto npm, usando o comando *npm init -y*

```
samuellucas in      /pgats/automacao-web-pgats-i
◦› npm init -y
Wrote to
                  /pgats/automacao-w

{
  "name": "automacao-web-pgats-i",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": ""
}
```

Passo 04: Instale o Cypress como dependência de desenvolvimento do projeto, usando o comando *npm install -D cypress@13.7.3*

```
 samuellucas in      /pgats/automacao-web-pgats-i
•› npm install -D cypress@13.7.3

added 173 packages, and audited 174 packages in 8s

39 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

Passo 05: Abra o Cypress e siga os passos iniciais para criar a estrutura padrão do projeto. Para abrir o Cypress, use o comando *npx cypress open*

```
 samuellucas in      /pgats/automacao-web-pgats-i
•› npx cypress open

DevTools listening on ws://127.0.0.1:61488/devtools,
8a6c9678be
□
```

Cypress

automacao-web-pgats-i

v13.7.3 · Upgrade Docs Log in

Welcome to Cypress!

Review the differences between each testing type →



E2E Testing

Build and test the entire experience of your application from end-to-end to ensure each flow matches your expectations.

Not Configured



Component Testing

Build and test your components from your design system in isolation in order to ensure each state matches your expectations.

Not Configured

Cypress

automacao-web-pgats-i › E2E Testing

v13.7.3 · Upgrade Docs Log in

Configuration files

We added the following files to your project:

 cypress.config.js	The Cypress config file for E2E testing.	▼
 cypress/support/e2e.js	The support file that is bundled and loaded before each E2E spec.	▼
 cypress/support/commands.js	A support file that is useful for creating custom Cypress commands and overwriting existing ones.	▼
 cypress/fixtures/example.json	Added an example fixtures file/folder	▼

Continue

Cypress

automacao-web-pgats-i > E2E Testing

v13.7.3 · Upgrade Docs Log in

Choose a browser

Choose your preferred browser for E2E testing.



Chrome
v128



Electron
v118

[Start E2E Testing in Chrome](#)

← Switch testing type

automacao-web-pgats-i Specs v13.7.3 · Upgrade Chrome 128 · Docs Log in

Create your first spec

Since this project looks new, we recommend that you use the specs and tests that we've written for you to get started.



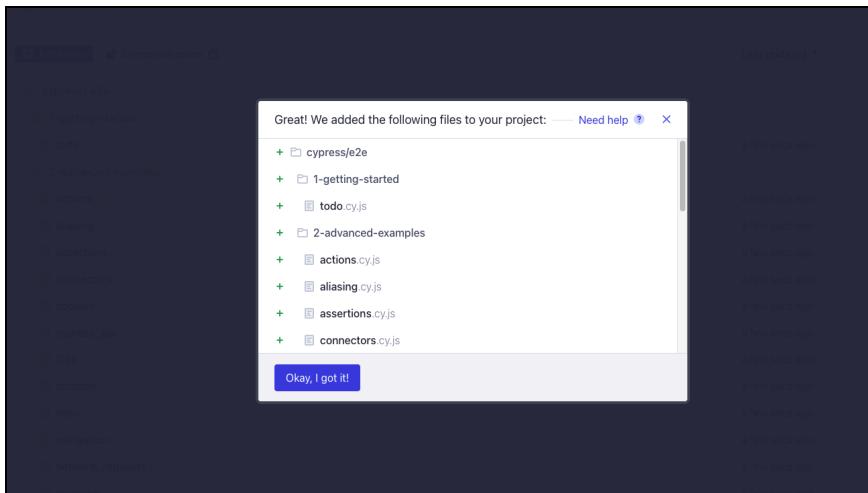
Scaffold example specs
We'll generate several example specs to help guide you on how to write tests in Cypress.



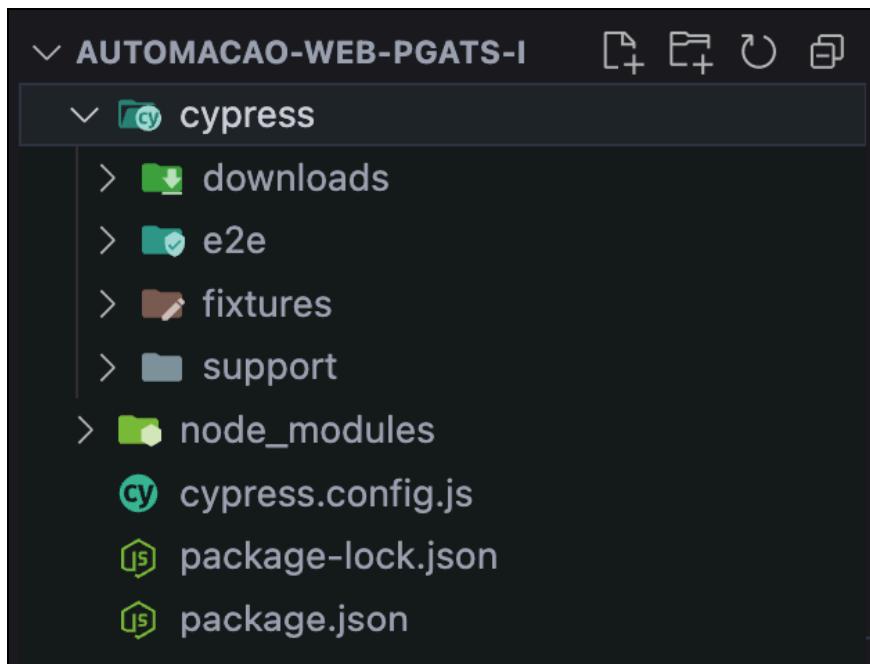
Create new spec
We'll generate a template spec file which can be used to start testing your application.

If you feel that you're seeing this screen in error, and there should be specs listed here, you likely need to update the spec pattern.

[View spec pattern](#)



This screenshot displays two side-by-side windows. The left window is the Specs page, showing a single test file named "todo.cy.js" with a green checkmark icon. The right window is a browser preview of a "todos" application. The URL is https://example.cypress.io/todo. The browser title is "cypress.io". The application interface has a header with "todos", a text input field with "What needs to be done?", and a list containing a single item: "Walk the dog". Below the list are buttons for "All", "Active", and "Completed". At the bottom of the browser window, there is some small text about developer tools and GitHub integration.



Escrevendo o primeiro teste automatizado

Passo 01: Acesse a aplicação que vamos utilizar durante as aulas, que simula um e-commerce e tem apenas a finalidade de estudo de testes automatizados:

<https://automationexercise.com>

Automation Exercise

Home Products Cart Signup / Login Test Cases API Testing
Video Tutorials Contact us

AutomationExercise

Full-Fledged practice website for Automation Engineers

All QA engineers can use this website for automation practice and API testing either they are at beginner or advance level. This is for everybody to help them brush up their automation skills.

Test Cases APIs list for practice



...
CATEGORY FEATURES ITEMS

CATEGORY	FEATURES ITEMS
WOMEN	+
MEN	+
KIDS	+

Passo 02: Exclua os arquivos de exemplo da pasta `cypress/e2e` e crie um novo arquivo de testes com o nome que preferir, desde que tenha a extensão `.cy.js`

Exemplo: `automation-exercise.cy.js`

✓ AUTOMACAO-WEB-PGATS-I

✓  cypress

>  downloads

✓  e2e

|  automation-exercise.cy.js

>  fixtures

>  support

>  node_modules

 cypress.config.js

 package-lock.json

 package.json

Passo 03: Crie a estrutura de testes, que no Cypress, usa basicamente a sintaxe do Mocha

The screenshot shows a code editor window with a dark theme. The title bar says "automation-exercise.cy.js e2e X". Below the title bar, the path "cypress > e2e > automation-exercise.cy.js > ..." is visible. The code itself is a simple Mocha-style test structure:

```
1  describe(' ', () => {
2    it(' ', () => {
3    });
4  });
5});
```

Passo 04: Acompanhe a implementação dos testes, mapeando os elementos e interagindo durante a aula. Os principais comandos que serão usados são:

- visit
- get
- contains
- type
- click
- select
- should
- wait
- eq
- selectFile
- url

ESPERAS EXPLÍCITAS E IMPLÍCITAS

Em automação de testes web, um dos principais desafios é garantir que a execução do comando e o estado da aplicação estejam "sincronizados". Ou seja, o comando deve esperar por uma determinada condição da aplicação para que seja executado.

Por exemplo, digamos que você esteja testando uma página de cadastro e deseja adicionar uma verificação de que, ao clicar no botão "Cadastrar", o usuário seja direcionado para a página inicial.

Ao clicar no botão Cadastrar, uma série de ações pode acontecer para que o cadastro seja finalizado: chamadas para APIs, carregamento de novos componentes e scripts, dentre outros. Essas ações podem ocorrer em milissegundos ou levar alguns segundos.

A ferramenta de testes, de modo geral, não sabe que a aplicação está "trabalhando" para concluir aquele processo. Sendo assim, o comando que faz verificação do redirecionamento para a página inicial pode ser executado antes do carregamento ser concluído. Isso vai fazer com que o teste falhe, mesmo que não exista um problema real na aplicação em teste.

Para lidar com esta necessidade de "sincronismo", as ferramentas de testes possuem mecanismos de esperas, que servem como auxiliares e garantem que o comando seja executado no momento certo. Existem pelo menos 3 tipos de esperas: implícitas, explícitas e automáticas.

Esperas Implícitas

Espera implícita é o tempo máximo pelo qual a ferramenta deve aguardar enquanto estiver executando um comando. Normalmente, é uma configuração global.
Exemplo:

```
// Set the implicit wait time to 2 seconds
Cypress.config('defaultCommandTimeout', 2000)
```

Esperas Explícitas

Espera explícita é uma condição pela qual a ferramenta deve aguardar antes de prosseguir. Pode ser de comportamento ou tempo. Por exemplo, esperar que o elemento esteja visível ou esperar por 5 segundos.

Exemplo:

```
// Wait until an element becomes visible
cy.wait('#some-element').should('be.visible')
```

Esperas Automáticas

Ferramentas modernas para automação de testes web como Cypress e Playwright possuem também um mecanismo chamado Espera Automática, que executa algumas verificações de forma automática a cada comando, como checagens de visibilidade e estabilidade do elemento antes de interagir.

Here is the complete list of actionability checks performed for each action:

Action	Visible	Stable	Receives Events	Enabled	Editable
locator.check()	Yes	Yes	Yes	Yes	-
locator.click()	Yes	Yes	Yes	Yes	-
locator dblclick()	Yes	Yes	Yes	Yes	-
locator.setChecked()	Yes	Yes	Yes	Yes	-
locator.tap()	Yes	Yes	Yes	Yes	-

Exemplo de checagens automáticas do Playwright

SCREENSHOTS

Screenshots são capturas de tela feitas durante a execução dos testes e servem como suporte tanto para análise dos resultados quanto para gerar evidências da execução do teste. Eles podem ser utilizados de várias maneiras para aumentar a confiabilidade e o diagnóstico de testes automatizados.

Existem várias abordagens para capturar essas imagens:

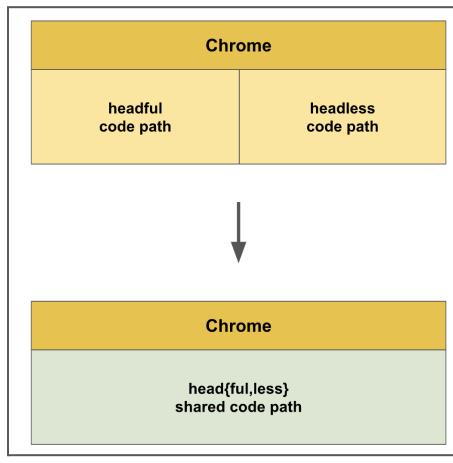
- **Durante o teste:** Como no caso do cy.screenshot no Cypress, onde é possível capturar a tela em momentos específicos, seja em etapas intermediárias ou após verificações importantes.

- **Quando há falha:** Configurações como screenshotOnRunFailure no Cypress, permitem capturar

automaticamente a tela quando o teste falha, permitindo uma análise visual do momento exato em que o erro ocorreu.

- **A cada comando de forma automática:** Plugins como o stepByStepReport no CodeceptJS capturam a tela de forma automática a cada passo do teste, criando um relatório com evidências visuais da execução mostrando o passo a passo.

EXECUÇÃO DE TESTES HEADLESS



antigo vs novo headless no Chrome

O modo headless é, basicamente, executar o navegador sem nenhuma interface visível. No passado, usar o modo headless exigia alguns cuidados, dada a sua

diferença em relação ao modo "headed" ou "headful" - com a interface visível.

Por exemplo, o antigo modo headless que o Google lançou em 2017 para o Chrome 59 era [uma implementação de navegador alternativa e separada](#) do Chrome.

A atualização de [2023](#) para o navegador headless do Google Chrome mesclou os caminhos de código dos dois modos, onde a única diferença é que no modo headless o Chrome cria, mas não exibe nenhuma janela do navegador.

Execução de testes no modo headless

Com a ferramenta que estamos usando (Cypress), para executar testes no modo headless usamos o seguinte comando:

```
npx cypress run
```

Após executar este comando em seu terminal, você deve visualizar algo similar a imagem abaixo:

```
> npx cypress run
DevTools listening on ws://127.0.0.1:64206/devtools/browser/3e4d8753-b4ff-4086-884c-2db06cfb2a10
=====
(Run Starting)

Cypress:          13.7.3
Browser:         Electron 118 (headless)
Node Version:    v22.5.1 (/opt/homebrew/Cellar/node/22.5.1/bin/node)
Specs:            5 found (automation-exercise-com-xpath.cy.js, automation-exercise-with-pom.cy.js, automation-exercise.cy.js, failing-tests.cy.js, windows-e-drag-and-drop.cy.js)
Searched:        cypress/e2e/**/*.cy.{js,jsx,ts,tsx}
```

Ao finalizar a execução, você deve visualizar um resumo dos resultados de forma similar a imagem abaixo:

(Run Finished)

Spec		Tests	Passing	Failing	Pending	Skipped
✓ automation-exercise-com-xpath.cy.js	00:16	1	1	-	-	-
✓ automation-exercise-with-pom.cy.js	00:07	2	2	-	-	-
✓ automation-exercise.cy.js	00:24	1	1	-	-	-
✓ failing-tests.cy.js	00:10	6	6	-	-	-
✓ windows-e-drag-and-drop.cy.js	00:05	2	2	-	-	-
✓ All specs passed!	01:04	12	12	-	-	-

MODULARIZAÇÃO E SEPARAÇÃO DE RESPONSABILIDADES

Conforme os projetos de automação crescem, torna-se cada vez mais importante estruturar o código de forma organizada, previsível e fácil de manter.

A organização e separação de responsabilidades é o princípio que garante que cada parte do código tenha um papel claro dentro do processo de teste, seja localizar elementos, executar ações ou validar resultados.

Modularizar é dividir um sistema complexo em partes menores e independentes, onde cada parte (ou módulo):

- Cumpre uma função específica (ex: preencher um formulário, autenticar um usuário, consultar uma API).
- Pode ser reutilizada por outros testes

- Pode ser alterada isoladamente, sem impacto em outras áreas

Essa prática segue o mesmo princípio do desenvolvimento de software tradicional: “Cada módulo deve fazer uma coisa, e fazê-la bem.”. Na automação de testes, isso significa evitar que o script de teste contenha toda a lógica, seletores e verificações no mesmo arquivo.

A modularização pode ser feita de várias formas, dependendo da ferramenta e linguagem adotadas. No caso do JavaScript (com ferramentas como Cypress ou Playwright), a estrutura costuma ser dividida entre:

Módulo	Responsabilidade Principal
Módulos, Pages, Features	Executar tarefas como clicar, preencher ou navegar
Seletores, Locators	Guardar os caminhos ou identificadores dos elementos
Assertions	Verificar comportamentos esperados
Utilitários, Helpers	Funções auxiliares e genéricas

Separar essas responsabilidades permite construir um fluxo de testes limpo e intuitivo, em que cada parte do código sabe exatamente o que deve fazer. Porém, essa divisão não é uma regra, apenas um ponto de partida que pode ser adaptado a cada contexto.

Durante as aulas, vamos aprender como implementar este padrão na prática usando recursos do Javascript.

RELATÓRIOS DE EXECUÇÃO DE TESTES

Relatórios nos ajudam a passar a visão resumida da execução dos testes, permitindo uma análise dos resultados assim que a execução é concluída. Eles podem ser gerados em diversos formatos como logs em texto, arquivos JSON, páginas HTML interativas ou xml.

A escolha do relatório e formato de saída depende da necessidade de visualização e como ele será compartilhado. Por exemplo, para visualizar de forma integrada a sistema de CI é comum usar o formato de saída em XML.

Vejamos um exemplo prático de relatório html:

The screenshot shows a browser window displaying a Cypress test report. At the top, there's a header with a green checkmark icon, the text "default todos exists", a red X icon, and the text "fail test". Below this is a red error message: "AssertionError: Timed out retrying: Not enough elements found. Found '4', expected '10'." A large block of stack trace follows, detailing the error from the Cypress runner to the application code. Below the stack trace, a section titled "+ expected - actual" shows the difference between the expected value (4) and the actual value (10). The test context below shows a "Test 1" entry with a green checkmark, indicating it passed. The "TODO list" section shows four items: todo1, todo2, todo3, and todo4. The third item, todo3, is highlighted in red, corresponding to the failing assertion in the test log.

O Mochawesome Reporter é uma ferramenta que gera relatórios HTML a partir de testes executados com o Mocha, um framework de testes. Ele apresenta uma visão interativa e com os detalhes do teste, além de gráficos para facilitar a leitura.

Algumas ferramentas possuem integração com o Mocha e isso permite a compatibilidade com esse relatório. O Cypress Mochawesome Reporter é uma biblioteca que integra o Mochawesome ao Cypress, com uma configuração simplificada.

Durante as aulas, vamos aprender a configurar este relatório passo a passo, assim como visualizar o relatório.

BOAS PRÁTICAS EM AUTOMAÇÃO DE TESTES WEB

1. Usar seletores que provavelmente não serão alterados

Em testes, é importante utilizar seletores que sejam estáveis e menos propensos a mudanças. A intenção é evitar que os testes quebrem quando a interface for alterada. Seletores como *data-cy*, *data-test* ou *test-id* são mais recomendados do que IDs, classes ou outros atributos que podem mudar com frequência.

2. Mover ações que se repetem antes ou depois de cada teste para Hooks

Se você possui ações que se repetem em vários testes como abrir uma página ou gerar dados, você pode movê-las para uma estrutura dentro do teste chamada *Hooks*.

Hooks são trechos de código que serão executados antes de todos os testes, antes de cada teste, depois de cada testes ou depois de todos (before, beforeEach, afterEach e after, respectivamente).

Isso ajuda a reduzir a duplicação de código. Exemplo:

```
beforeEach(() => {
  cy.visit('/login');
  cy.get('[data-cy=username]').type('user');
  cy.get('[data-cy=password]').type('password');
```

```
    cy.get('[data-cy=login-button]').click();
});

it('teste', () => {
  // passos do teste
})
```

3. Definir uma URL base global para execução dos testes

Em vez de repetir a URL do site em cada teste, uma boa prática é definir uma URL global em um arquivo de configuração ou variáveis de ambiente. Isso facilita a manutenção pois, se a URL mudar, você só precisará atualizá-la em um lugar.

No Cypress, essa definição é feita através da variável de configuração `e2e.baseUrl`, da seguinte forma:

```
module.exports = {
  e2e: {
    baseUrl: 'url do site',
  },
};
```

4. Usar múltiplas asserções sempre que possível

Ao testar funcionalidades, é recomendado usar verificar múltiplos aspectos com várias asserções em um mesmo teste. Isso garante que vários comportamentos sejam validados, aumenta a confiabilidade do teste e aproveita o estado da aplicação.

Exemplo:

```
cy.get('[data-testid="first-name"]')
```

```
.type('johnny')
.should('have.attr', 'data-validation', 'required')
.and('have.class', 'active')
.and('have.value', 'Johnny')
```

5. Mover códigos que se repetem para funções ou arquivos

Quando blocos de código se repetem em vários testes, é uma boa prática movê-los para funções ou arquivos de utilitários. Isso mantém os testes mais limpos, facilita a legibilidade e centraliza lógicas que podem ser reutilizadas.

Exemplos:

```
Cypress.Commands.add('login', (username,
password) => {
  cy.visit('/login');
  cy.get('[data-cy=username]').type(username);
  cy.get('[data-cy=password]').type(password);
  cy.get('[data-cy=login-button]').click();
});
```

ou

```
function login (username, password) {
  cy.visit('/login');
  cy.get('[data-cy=username]').type(username);
  cy.get('[data-cy=password]').type(password);
  cy.get('[data-cy=login-button]').click();
};
```

EXECUÇÃO DE TESTES NA NUVEM

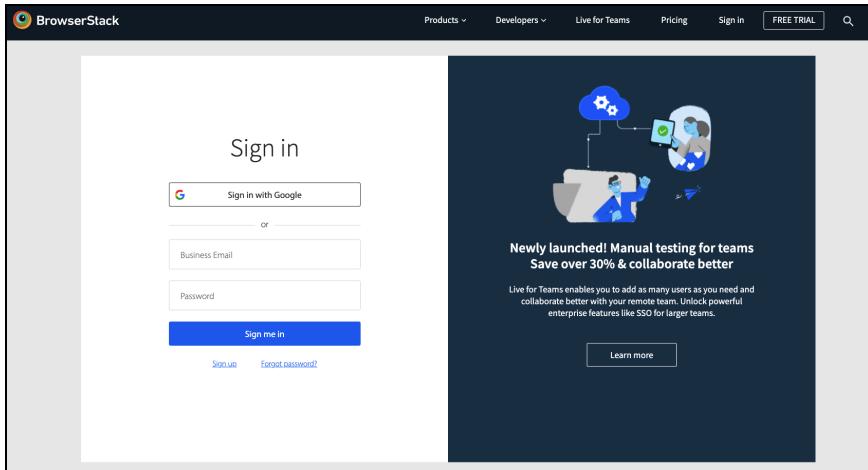
Plataformas de testes na nuvem oferecem uma infraestrutura para execução dos testes. Em automação de testes web, auxiliam em aspectos como **escalabilidade**, **fragmentação** de dispositivos e **infraestrutura** de testes.

O Browserstack é a principal plataforma de testes na nuvem atualmente (2024). Podemos tanto explorar aplicações de forma manual, quanto executar testes automatizados web e mobile. É uma ferramenta **paga**, porém, cada novo usuário possui 100 min de uso gratuito dos recursos da plataforma para testes web (Out/2025).

Integrando com o BrowserStack

Passo 01 - Acesse o site oficial do BrowserStack e faça login. Se você não possui cadastro, pode entrar com uma conta da Google ou criar uma nova conta.

https://www.browserstack.com/users/sign_in



Passo 02 - Após o login, navegue para Products -> Automate. Deve ser direcionado para o Dashboard, onde serão exibidas as execuções de testes após a configuração.

A screenshot of the BrowserStack Products dashboard. The top navigation bar includes "Test Management", "Products", "Invite team", "Plans and pricing", and a search bar. On the left, there's a sidebar titled "OS Real Devices" listing "el 6 Pro", "el 7", "el 7 Pro", "el 8", "el 8 Pro", and "el 9". The main content area is divided into several sections: "Web Testing" (Test websites or web apps on real browsers), "App Testing" (Test iOS & Android mobile apps on real devices), "Manual Testing" (Live Cross-browser testing), "Test Automation" (Automate Browser automation grid, Automate TurboScale, Accessibility Automation, Percy, Low Code Automation), and "Management & Optimization" (Test Management, Test Observability). A callout box highlights the "Automate" section. At the bottom, there's a message: "Empower teams with BrowserStack for Enterprise → Tools: SpeedLab, Screenshots, Responsive".

Passo 03 - Clique em Access Key e guarde as credenciais de *User Name* e *Access Key*, serão usadas nos próximos passos.

The screenshot shows the BrowserStack Automate interface. At the top, there's a navigation bar with 'Live', 'App Automate', 'Test Observability', 'Products', and a 'Plans and pricing' button. Below the navigation is a search bar with placeholder text 'Search for Projects, Builds, Sessions or Cypress Specs'. The main area is titled 'DASHBOARD' and shows a build named 'automacao-web-pgats'. The build summary indicates '4 executions + 2 Parallels', '0 Flaky' tests, and a 'PASSED' status. It was started at '20 Jul 2024 20:12 UTC' and is running 'Local Testing'. Below the summary are filters for 'Status', 'OS', 'Browser', and 'Flakiness'. A modal window titled 'ACCESS KEY' is open, prompting for 'User Name' (with 'sa' entered) and 'Access Key' (with 'Ac' entered). The modal also shows 'Parallel THREADS: Running' and features like 'Faster integration' and 'Minimal effort'.

Passo 04 - Em seu terminal, instale o BrowserStack - Cypress CLI, uma ferramenta de linha de comando que é usada para permitir a comunicação entre o Cypress e o BrowserStack. Use o comando abaixo:

```
npm install -g browserstack-cypress-cli
```

Passo 05 - Em seu terminal, inicialize a configuração do BrowserStack no projeto, usando a ferramenta que instalamos no passo anterior. Use o comando abaixo:

```
browserstack-cypress init
```

Em seu projeto, após a configuração, um arquivo deve ter sido criado automaticamente: *browserstack.json*.

Este arquivo possui as configurações necessárias para que os testes sejam executados no BrowserStack. Basicamente, são as credenciais, informação de arquivo de configuração do Cypress, navegadores, configuração de paralelismo e dependências adicionais.

Passo 06 - Insira o *User Name* e *Access Key* obtidos anteriormente no arquivo gerado:



```
1 "auth": {  
2     "username": "",  
3     "access_key": ""  
4 },
```

Passo 07 - (opcional) Para otimizar o uso dos minutos *gratuitos*, reduza ao máximo a combinação de browsers no arquivo de configuração do browserstack:

```
1 "browsers": [
2   {
3     "browser": "chrome",
4     "os": "Windows 10",
5     "versions": ["latest"]
6   }
7 ],
```

Passo 08 - Atualize o run_settings.cypress_config_file para o caminho do arquivo de config do cypress

Passo 09 - Adicione no run_settings.cypress_version a versão do Cypress que quiser usar (máx v14)

Passo 10 - Atualize o run_settings.project_name com o nome do seu projeto

Passo 11 - Atualize o run_settings.parallels para 1

Passo 12 - Adicione as dependências que foram usadas no seu projeto em run_settings.npm_dependencies

```
1 "run_settings": {  
2     "cypress_config_file": "./cypress.config.js",  
3     "cypress_version": "13.latest",  
4     "project_name": "pgats-automacao-web",  
5     "parallels": 1,  
6     "npm_dependencies": {  
7         "@faker-js/faker": "^9.9.0",  
8         "cypress-mochawesome-reporter": "^4.0.2"  
9     },
```

Após concluir as configurações acima, para executar os testes usando a plataforma do BrowserStack, execute:

browserstack-cypress run --sync --spec “caminho da spec”

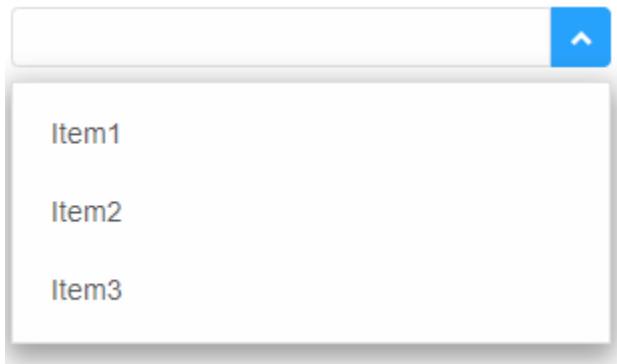
Isso deve iniciar uma nova execução diretamente na plataforma. Observação: Eventualmente, a plataforma pode estar desenvolvendo e testando novos *dashboards*, e isso pode atrapalhar a visualização dos resultados.

Para ter certeza que sua execução ocorreu com sucesso, você pode alterar para o dashboard antigo; ou conferir a saída da execução no terminal, que deve exibir o resultado.

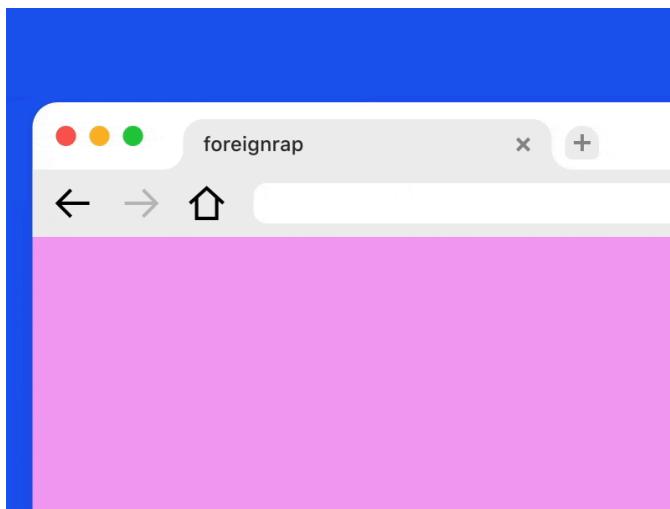
COMBOBOXES, JANELAS E DRAG AND DROP

Comboboxes

Comboboxes, também conhecidos como dropdown, são elementos que permitem aos usuários selecionar uma opção de uma lista suspensa, por exemplo: lista de estados ou cidades.



Janelas



Janelas, também conhecidas como Abas, são uma forma de navegar em múltiplas instâncias dentro de um mesmo navegador. Aplicações web costumam usar esse recurso para abrir sites externos ou recursos de pagamento e autenticação.

Em automação de testes web isso representa um desafio devido ao fato de que muitas ferramentas de automação não oferecem suporte direto para a lidar com múltiplas janelas ou abas.

A dificuldade surge porque, durante a execução de testes automatizados web, o foco do navegador normalmente permanece em uma única janela ou aba. Testar fluxos que envolvem múltiplas janelas, exige que o framework consiga alternar o controle entre essas janelas, algo que nem sempre é simples ou possível.

Como o Cypress normalmente lida com isso

O Cypress, por design, não suporta múltiplas janelas ou abas. Ele opera dentro de uma única janela do navegador para garantir a confiabilidade e a consistência dos testes. Segundo a documentação, essa limitação é intencional e visa simplificar o ambiente de teste, evitando problemas comuns associados à manipulação de múltiplos contextos de navegador.

Para lidar com situações onde a aplicação tenta abrir uma nova janela ou aba, o Cypress sugere **remover o atributo que abre nova janela**. Essa estratégia envolve modificar o comportamento do elemento que abre a nova janela, removendo o atributo `target="_blank"` para que o link abra na mesma janela.

Janelas e ferramentas baseadas no WebDriver

Ferramentas baseadas no WebDriver, como o Selenium, oferecem suporte direto para manipulação de múltiplas janelas e abas.

Elas permitem que o script de teste identifique e troque entre diferentes janelas, usando métodos como `getWindowHandles()`. Após capturar as janelas abertas, é possível alternar o foco entre elas com métodos como o `switchTo().window()`.

Drag and Drop



Drag and Drop (arrastar e soltar), são interações que permitem ao usuário mover um elemento de uma área para a outra, utilizando o mouse ou o toque, *arrastando* o elemento de uma área e *soltando* em outra. Essa funcionalidade é muito usada em aplicações web para tarefas como organizar listas. Um exemplo muito popular são os cartões nas colunas do Trello.

Em automação de testes isso representa um desafio porque envolve eventos complexos, como "mousedown", "mousemove", "mouseup", e a simulação da movimentação do cursor do usuário. Além disso, algumas bibliotecas Javascript utilizadas para implementar Drag and Drop podem ter comportamentos específicos, o que torna mais complicado ainda.

Como o Cypress lida com isso

O Cypress, por padrão, não oferece suporte nativo para a simulação de Drag and Drop. No entanto, existem algumas abordagens para lidar com isso. Uma delas, é usar plugins e bibliotecas de terceiros.

Uma das bibliotecas mais populares é a *cypress-drag-drop*, que adiciona comandos personalizados para realizar a ação de arrastar e soltar. Exemplo:

Após instalar e configurar o plugin *cypress-drag-drop*, seguindo os passos da documentação oficial, você pode utilizá-lo da seguinte forma:

```
cy.get('[data-cy=draggable]').drag('[data-cy=droppable]');
```

IA E AUTOMAÇÃO DE TESTES WEB

O uso de IA em automação de testes está cada vez mais popular. Ferramentas e modelos como ChatGPT, Claude e Gemini já extrapolaram o papel de meros *chatbots*, sendo capazes de compreender contextos de testes, sugerir cenários e até mesmo gerar scripts automatizados assim como fizemos durante a disciplina.

Alguns dos principais conjuntos de uso de IA são:

- Chatbots: Claude, ChatGPT, Gemini, ideais para discutir cenários, analisar erros, pedir sugestões;
- Plugins / Integrações: Copilot, Cursor, ajudam diretamente dentro do editor de código, sugerindo comandos, funções e correções em tempo real.
- SDKs e Protocolos: ChatGPT SDK, Agentes, MCP, permitem integração direta com dados e ferramentas, padronizando o diálogo entre IA e sistemas.

O Protocolo MCP (Model Context Protocol) representa um avanço importante para conectar Modelos de IA a fontes de dados externas (como bancos, arquivos, APIs, etc.).

Isso conecta diretamente com automação de testes, dado que os *frameworks* poderão utilizar este protocolo e permitir com que os scripts de testes sejam mais “descritivos” e não necessariamente comandos “fixos” como clicar e digitar.

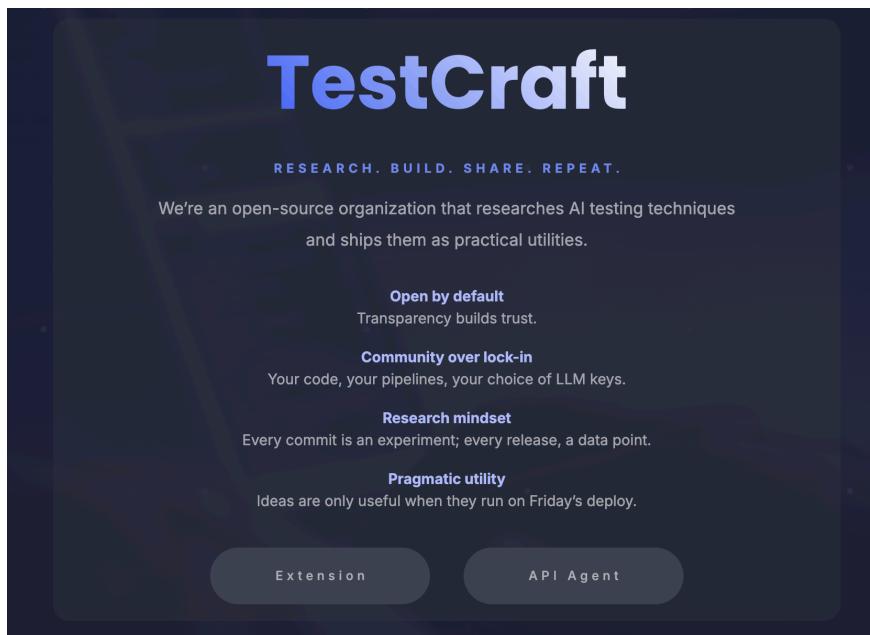
Ferramentas como Playwright, já tem integração com MCP.

Laboratório Prático: Geração de Testes com IA

Existem inúmeras possibilidades de uso de IA em automação de testes como um todo, conforme vimos anteriormente nos *principais conjuntos*.

Vamos explorar uma ferramenta que usa IA para entender o contexto de uma aplicação Web, sugerir ideias de cenários e gerar scripts automatizados a partir destas ideias.

Essa ferramenta é um *plugin*, chamado TestCraft.



EXERCÍCIOS

Exercício 01:

Hands-on: Usando o projeto e conceitos que aprendemos em aula, implemente os 5 primeiros cenários da lista na página *testcases* do site usado em aula.

Exercício 02:

Hands-on: Analise os testes criados para uma aplicação e que estão com falhas diversas, de sintaxe, funcionalidade, seletores. Execute, identifique a origem do erro e corrija.

Exercício 03:

Hands-on: Instale a biblioteca *cypress-xpath* no projeto e tente adaptar os seletores para usar xpath.

Observe aspectos como legibilidade, velocidade de execução e compartilhe suas percepções na aula.

Exercício 04:

Hands-on: Usando o projeto e conceitos que aprendemos em aula, conclua o processo de modularização das ações feitas para os demais testes do projeto.

Exercício 05:

Hands-on: Usando o projeto e conceitos que aprendemos em aula, implemente os cenários **8, 9, 10, 15 e 16** da lista na página *testcases* do site usado em aula.

REFERÊNCIAS

1. **Definition of HTML**
<https://developer.mozilla.org/pt-BR/docs/Web/HTML>
2. **Definition of CSS**
<https://developer.mozilla.org/pt-BR/docs/Web/CSS>
3. **Definition of JavaScript**
<https://developer.mozilla.org/pt-BR/docs/Web/JavaScript>
4. **CSS Selectors**
https://developer.mozilla.org/pt-BR/docs/Web/CSS/CSS_selectors
5. **Definition of XPath**
<https://developer.mozilla.org/en-US/docs/Web/XPath>
6. **XPath Expression Components**
<https://www.iro.umontreal.ca/~lapalme/ForestInsteadOfTheTrees/HTML/ch04s01.html>
7. **Cypress vs Other Test Runners**
<https://glebbahmutov.com/blog/cypress-vs-other-test-runners>
8. **Automation Protocols in WebdriverIO**
<https://webdriver.io/docs/automationProtocols>
9. **Test Automation Evolution**
<https://developer.chrome.com/blog/test-automation-evolution>

10. **Deprecating CDP Support in Firefox**
<https://fxdx.dev/deprecating-cdp-support-in-firefox-embracing-the-future-with-webdriver-bidi/>
11. **Firefox Remote Protocol Documentation**
<https://firefox-source-docs.mozilla.org/remote/cdp>
12. **Cypress Architecture**
<https://docs.cypress.io/guides/overview/key-differences#Architecture>
13. **TestCafe Architecture**
<https://testcafe.io/documentation/402631/guides/overview/why-testcafe#an-architecture-like-no-other>
14. **WebDriver BiDi in Chrome**
<https://developer.chrome.com/blog/webdriver-bidi?hl=pt-br>
15. **JavaScript Testing Best Practices**
<https://github.com/goldbergyoni/javascript-testing-best-practices>
16. **Webdriver Waits in Selenium**
<https://www.selenium.dev/pt-br/documentation/webdriver/waits/>
17. **Actionability in Cypress**
<https://docs.cypress.io/guides/core-concepts/interacting-with-elements#Actionability>
18. **Actionability in Playwright**
<https://playwright.dev/docs/actionability>
19. **Step by Step Report in CodeceptJS**
<https://codecept.io/plugins/#stepbystepreport>
20. **Screenshot Configuration in Cypress**
<https://docs.cypress.io/guides/references/configuration#Screenshots>

21. **Screenshot Command in Cypress**
<https://docs.cypress.io/api/commands/screenshot>
22. **Screenshots in Playwright**
<https://playwright.dev/docs/screenshots>
23. **New Chromium Headless Mode**
<https://developer.chrome.com/docs/chromium/new-headless?hl=pt-br>
24. **Design Patterns for Test Automation: Page Factory**
<https://www.nagarro.com/en/blog/design-patterns-test-automation-page-factory>
25. **Page Object Model**
<https://martinfowler.com/bliki/PageObject.html>
26. **Page Object Pattern: An Industrial Case Study**
<https://sepl.dibris.unige.it/publications/2013-leotta-ICSTW.pdf>
27. **Selenium Page Object Model**
https://www.selenium.dev/documentation/test_practices/encouraged/page_object_models
28. **Single Page Scripts in Test Automation**
https://testautomationpatterns.org/wiki/index.php/SINGLE_PAGE_SCRIPTS
29. **Fluent Page Object Pattern in Test Automation**
<https://www.codeproject.com/Articles/1019472/Fluent-Page-Object-Pattern-in-Automation-Testing>
30. **Fluent Interface in Object-Oriented Programming**
https://www.wikiwand.com/en/Fluent_interface
31. **Factory Method Design Pattern**
<https://refactoring.guru/pt-br/design-patterns/factory-method>

32. Run Your First Test in BrowserStack with Cypress

<https://www.browserstack.com/docs/automate/cypress#run-your-first-test>

33. Cypress Mochawesome Reporter

<https://www.npmjs.com/package/cypress-mochawesome-reporter>

34. Automation Exercise (aplicação usada nas aulas)

<https://automationexercise.com>