Cypress – Guia rápido

1 – Requisitos e configuração

Instalar:

- **NodeJS:** https://nodejs.org/pt/download
- **VS Code:** https://code.visualstudio.com/download
- **Biblioteca Faker:** https://www.npmjs.com/package/@faker-js/faker

Projeto prático: https://www.saucedemo.com/v1/

Configurando projeto:

1. Crie a pasta do projeto e abra com a IDE VS Code
2. Abra um novo terminal e execute os comandos abaixo:
    - npm init –y
    - npm install cypress --save-dev
    - npx cypress open

Estrutura do projeto cypress:

cypress/
- **e2e/**: diretório onde criamos nossos testes end-to-end (E2E).
- **fixtures/**: diretório para arquivos estáticos, como dados simulados (JSON) que podem ser usados nos testes.
- **support/**: diretório de arquivos de configuração e comandos personalizados.

cypress.config.js: arquivo de configuração principal do Cypress.
node_modules/: diretório com todas as dependências do projeto.
package.json: arquivo que lista as dependências e scripts do projeto.

2 – Criando o primeiro teste

Crie um arquivo no diretório **cypress/e2e** com a extensão ***.cy.js***, como login.cy.js:

```
describe('Minha Suite de testes', function(){
     it("Meu primeiro teste", function(){
          cy.visit('https://meu-site.com')
          cy.get('#text-nome').type('Conteúdo')
          cy.get('.select-sexo').select('Feminino')
          cy.get('input[type="checkbox"]').check('sunday')
          cy.get('button[value="cadastrar"]').click()
          cy.get('#span').contains('Usuário cadastrado').should('be.visible')
     })
})
```

Estrutura organizacional dos testes:

- **Describe():** usado para definir uma suíte de testes
- **It():** define um teste especifico, dentro do contexto describe().

Observação: ambos trabalham com a estrutura de function ou arrow-function;

Comandos utilizados:

- **Cy.visit():** usado para visitar/abrir um página no navegador
- **Cy.get():** usado para selecionar um elemento na página html
- **Cy.type():** usado para inserir conteúdo em campos de texto.
- **Cy.select():** usado para selecionar o valor de um elemento do tipo combobox;
- **Cy.check():** usado para selecionar um elemento do tipo checkbox
- **.contains():** usado para verificar se um elemento contém um valor textual;
- **.should():** usado para fazer validações, como se um elemento está visível, possuí um texto ou um valor.

Usando seletores

- **Id:** elementos são selecionados por seu id usando o operador #nome-id
- **Classe:** elementos são selecionados por sua classe usando o operador.nome-classe
- **Elemento e atributos:** selecionando elemento pela tag e seus atributos input[type=’text’]

3 - Padrão Page-Object

Padrão utilizado como camada de abstração entre os testes e a interface da aplicação, ajudando a manter o código limpo, organizado e fácil de atualizar.

Basicamente, separamos do teste toda a lógica de mapeamento dos elementos e criamos funções para realizar as ações com nomes com maior significado. Além disso, o mapeamento dos elementos ficam agrupados pelo contexto da página que deseja testar, como: página de login, página de cadastro, página de configurações etc.

Passo-a-passo:

1 Passo: crie uma nova pasta na pasta **Support**, por exemplo com o nome Pages.

2 Passo: cria a pasta que representa a página que deseja mapear, por exemplo Login;

3 Passo: crie os arquivos elements.js e index.js.

elements.js – usado para mapear os elementos da página

```
# Elementos da pagina
export const ELEMENTS ={
 fieldUserName: '#user-name',
 buttonLogin: '#login-button',
}
```

index.js – usa a estrutura de Classe no JavaScript para criar os métodos responsáveis pelas ações na página, usando os elementos mapeados anteriormente.

```
# Classe com métodos de acesso aos elementos
const el = require('./elements').ELEMENTS
class Login {
    preenherInputUsername(username){
        cy.get(el.campoUsername).type(username)
    }
    clicarBtnLogin(){
        cy.get(el.botaoLogin).click()
    }
}
export default new Login()
```

4 Passo – criando o teste automatizado com a estrutura criada para o Page Object

```
/// <reference types='cypress' />
import Login from '../support/Pages/Login'
describe('Realizar Login', ()=>{
    it("Realizar Login com sucesso", ()=>{
         cy.visit("https://www.meu-site.com")
         Login.preenherInputUsername("José")
         Login.clicarBtnLogin()
    })
})
```

Observações:

- Na primeira linha usamos um código para referenciar o typo do cypress para ativar o auto-complete para comando dos cypress.
- Na segunda linha estamos importando o conteúdo do arquivo com a classe javascript, que contém os métodos de acesso aos elementos da página.
- No teste, usamos o objeto **Login** para chamar as funções definidas na página importada;

4 - Hooks

Os **hooks do Cypress** permitem executar trechos de código **antes ou depois** dos testes, sendo essenciais para preparar o ambiente de testes ou fazer limpezas após a execução.

Hook	Quando é executado	Para que serve
Before()	Uma vez **antes** de todos os testes	Inicializar dados e configurar estado.
BeforeEach()	**Antes** de cada bloco de teste	Resetar dados e consistência dos testes
After	Uma vez **depois** de todos os testes	Limpar dados e encerrar conexões
AfterEach()	**Depois** de cada bloco de teste	Limpar cookies, localStorage, ou resetar mocks.

Observação: O cypress recomenda realizar as ações descritas no after() e afterEach() nos hooks before() e beforeEach(), para garantir a consistência antes da execução dos testes.

5 - Interagindo com lista de elementos de um mesmo tipo e estado anterior

O Cypress trabalha com o encadeamento de comando, onde o comando atual usa os dados processados pelos comandos anteriores.

É comum precisarmos processar os dados retornados pela execução anterior, seja fazer uma validação, alterar o estado ou definir uma regra para processar valores quando a busca retorna vários elementos do mesmo tipo.

.each() => usado quando uma busca retornar vários elementos do mesmo tipo e precisamos interagir nesses vários elementos:

```
# Interagindo sobre um elemento do tipo checkbox para marcar os elementos
que iniciam com um texto definido
cy.get('input[type="checkbox').each(($check)=>{
        const valueText = $check.attr('value')
        if(valueText.startsWith('t')){
          cy.wrap($check).check()
        }
    })
```

.then() => usado para processar um único resultado retornado pelo comando anterior

```
# Interagindo sobre um elemento do tipo texto
cy.get('input[type="text').then(($then)=>{ ... })
```

.wrap() => usado para incluir o objeto processado novamente no escopo do Cypress, para assim conseguirmos realizar ações como o click() ou should(). Sem ele, a acção de click no exemplo irá falhar. Esse comando deve ser usado no each(), then() e qualquer outra estrutura que tenham o comportamento acima.

.and() => permite encadear várias validações, evitando a necessidade de usar um novo .get() no elemento.

```
cy.get('button[name="stop"]').should('be.visible').and('have.text',
'stop')
```

6 - Massa de Dados

O uso de massa de dados permite seu reaproveitamento em vários cenários de teste e facilita a manutenabilidade, uma vez que qualquer alteração em um valor, o ajuste será aplicado para todos os testes.

Para trabalhar com massa de dados podemos adotar três estratégias:

1 – Usando objetos JavaScript com as informações necessárias

```
# Elementos da pagina
const objJS ={
      email: "meuemail@email.com",
      senha: "m@il_123"
}
# acessando propriedades do objeto
cy.get('#email').type(objJS.email)
```

2 – Usando fixtures

As **fixtures no Cypress** são arquivos que armazenam dados estáticos usados nos testes automatizados. Elas ajudam a separar o **código de teste** dos **dados de entrada**, tornando os testes mais organizados, reutilizáveis e fáceis de manter.

Para que servem?

- Simular dados de usuários, produtos, respostas de API, etc.
- Facilitar testes com múltiplos cenários usando diferentes arquivos de dados.

Crie esses arquivos no diretório Cypress/fixtures/credenciais.js

```
# Arquivo JSON com os dados para usar nos testes
{
  "email":"standard_user",
  "senha": "standard_user"
}
```

Importe o arquivo usando o comando cy.fixture()

```
beforeEach(function(){
     cy.fixture(credenciais).then((dados)=>{
          this.credenciaisExt = dados
     })
})
# acessando propriedades do objeto
cy.get('#email').type(this.credenciaisExt.email)
```

Observações

1. O comando fixture() só precisa informar o nome do arquivo, não é obrigado a extensão.
2. Ao trabalhar com fixtures, o blocos de testes, como describ() e it(), devem ser definidos com functions, definido na linha 1 acima, pois fixtures não funcionam como arrow-functions.

3 - Trabalhando com Fabrica de dados

Use um objeto javaScript com a biblioteca Faker para criar dados dinâmico a cada nova execução.

Passos

1. Instalar a bliblioteca Faker: https://www.npmjs.com/package/@faker-js/faker
2. Criar a pasta Factories no diretório SUPPORT
3. Colar a estrutura disponibilizada na página web em nosso arquivo que criará registros **Faker**
4. Importar o arquivo: **import** {nomeFunction} **from** '../support/factories/destinatario.js'
5. Instanciar objeto: **const** destinatarioObj = nomeFunction();
6. Atribuir valores aos testes: destinatarioObj.propriedade

Exemplo:

1 – Criando nossa fábrica de registros

```
# Arquivo userFake.js que será nossa fábrica de registros
import { faker } from '@faker-js/faker';
export function criarUser() {
  return {
    firstName: faker.person.firstName(),
    lastName: faker.person.lastName()
  };
}
```

Observação: precisamos importar a biblioteca faker, linha 1, pois seu objeto de retorno que permitirá acessar os métodos de criação de geristros.

2 – Usando nossa fábrica para gerar registros automáticos

```
# Arquivo de teste
import {criarUser} from '../support/Factories/userFake.js'
# Instanciando nosso objeto com a fábrica
const usuarioFaker = criarUser()

it("Meu primeiro teste", ()=>{
     cy.visit('https://meu-site.com')
     cy.get('#first-name).type(usuarioFaker.firstName)
     cy.get('#last-name').select(usuarioFaker.lastName)
     cy.get('button[value="entrar"]').click()
})
```

7 - Trabalhando com ViewPorts

Por padrão, o Cypress usa as configurações de tamanho da tela para expandir as dimensões da tela que o teste está sendo realizado. Porém, alguns testes podem ser necessários que as dimensões sejam alteradas, simulando a execução em telas maiores ou menores, validando o uso em dispositivos moveis ou televisões.

1 - Altere as dimensões de forma especifica para cada teste manualmente

```
cy.viewport(960, 680)
```

2 - Use definições correspondentes para um dispositivo especifico. A lista completa pode ser encontrada na documentação oficial do cypress.

```
cy.viewport('ipad-mini', 'landscape')
```

Observação: é possível definir a rotação do dispositivo passando o segundo argumento como **landscape** para o padrão horizontal. O valor padrão é **portrait**, vertical.

3 - Podemos definir globalmente as configurações de altura e largura da tela no arquivo cypress.config.js

```
module.exports = defineConfig({
  e2e: {
    viewportWidth: 700,
    viewportHeight: 450,
    setupNodeEvents(on, config) {
      // implement node event listeners here
    },
  },
});
```

8 - Testes Visuais – validando CSS

Validar estilos CSS com Cypress é útil para garantir que elementos visuais importantes estejam exibidos corretamente, validando propriedades CSS como cores, tamanhos e posicionamentos:

Usando o método .should('have.css')

```
// Validando que o texto é vermelho
cy.get('.meu-elemento').should('have.css', 'color', 'rgb(255, 0, 0)');
//Verificar cor de fundo
cy.get('.card') .should('have.css', 'background-color', 'rgb(240, 240,
240)');
// Verificar fonte e estilo
cy.get('.titulo').should('have.css', 'font-size', '24px').and('have.css',
'font-weight', '700');
```

Observação: As cores geralmente retornam no formato **rgb(r, g, b)** — então se você está usando **hexadecimais** no CSS, como **#FF0000**, lembre-se de converter para **rgb(255, 0, 0)** no teste.

Link converter: https://www.w3schools.com/colors/colors_hexadecimal.asp

9 - Criando Comandos personalizados – Usando commands

Os **commands no Cypress** são comandos personalizados que você pode criar para **reutilizar ações comuns** nos testes automatizados. Eles funcionam como funções encapsuladas que deixam seu código mais limpo, organizado e fácil de manter.

Para que servem os commands?

- **Reutilização de código**: Evita repetir os mesmos trechos em vários testes.
- **Facilidade de manutenção**: Se algo muda na interface, você atualiza em um único lugar.
- **Melhoria na legibilidade**: Os testes ficam mais claros e focados na lógica, não nos detalhes técnicos.

É possível criar novos comandos usando **.add()** ou reescrever comandos existentes usando **.overwrite().** Para reescrever um comando existente, informando como nome, o nome do comando desejado.

Criando um novo commands

Comandos personalizados são criados no arquivo cypress/support/commands.js.

```
// cypress/support/commands.js
Cypress.Commands.add('login', (email, senha) => {
  cy.visit('/login');
  cy.get('input[name=email]').type(email);
  cy.get('input[name=password]').type(senha);
  cy.get('button[type=submit]').click();
});
```

Depois, pode usá-los nos testes com cy.nomeDoComando().

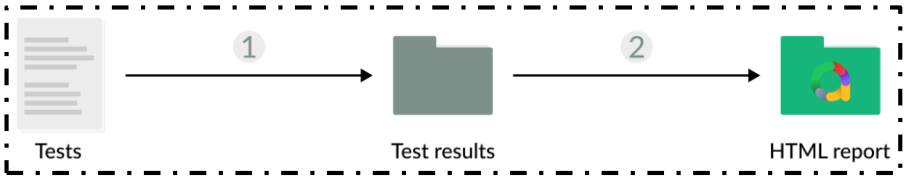
```
// login.spec.js
describe('Login', () => {
  it('Deve logar com sucesso', () => {
    cy.login('usuario@teste.com', '123456');
    cy.contains('Bem-vindo').should('be.visible');
  });
});
```

Comando para validar o estilo css aplicado em uma página

```
Cypress.Commands.add('validarCSS', (seletor, atributo, valor)=>{
  cy.get(seletor).should('have.css', atributo, valor)
})
```

10 – Criando relatórios com Allure

O Allure Report é uma biblioteca de código aberto para gerar relatórios de execução de testes e que podem ser abertos em qualquer lugar, pois construído como um pequeno site HTML estático. O relatório pode ser aberto em qualquer computador usando o próprio utilitário Allure Report ou até mesmo apenas um navegador.



Um fluxo de trabalho de teste com o Allure Report consiste em duas etapas, ambas as quais podem ser realizadas localmente, sem enviar nada pela rede.

1. **Fase de coleta**: enquanto os testes estão em execução, a estrutura de teste grava seus resultados em um arquivo ou diretório.
2. **Fase de visualização**: o utilitário de linha de comando do Allure Report lê os resultados do teste e cria um relatório HTML.

Instalação com NodeJS

Link: https://allurereport.org/docs/install-for-nodejs/

1 - Instalar a dependência do Allure na pasta raiz do projeto:

- npm install --save-dev mocha-allure-reporter allure-commandline

2 - Criando scripts de execução:

```
# Package.json
"scripts": {
    "cy:run": "npx cypress run --browser=chrome --headed --spec
cypress/e2e/advanced-examples/actions.cy.js --reporter mocha-allure-reporter",
    "report:allure": "allure generate allure-results --clean -o allure-report &&
allure open  allure-report"
  }
```

Execute os comandos acima usando o npm:

- Npm run cy:run
- Npm run report:allure