**Curso de React: escreva testes end to end com Cypress**

[*https://cursos.alura.com.br/course/react-escreva-testes-end-to-end-cypress*](https://cursos.alura.com.br/course/react-escreva-testes-end-to-end-cypress)

**INSTALANDO O CYPRESS**

Documentação:

<https://www.cypress.io/>

O Cypress é uma ferramenta de testes front-end de ponta a ponta (end-to-end). Ele visita uma guia no navegador e executa ações, por meio da interface do usuário, simulando o uso de uma pessoa usuária testando a aplicação.

Essa ferramenta permite a escrita de:

- testes de ponta a ponta

- testes de integração

- testes de unidade

- testes de componente, entre outros.

No terminal do front:

***npm install cypress --save-dev***

***npm i***

Finalizada a instalação, precisamos subir o servidor do Cypress:

***npm start***

***npx cypress open***

Na janela que abrir, selecione a opção **E2E Testing**.

Selecione o navegador.

No vscode, no arquivo de front, irá aparecer a pasta **Cypress** com os arquivos referentes.

Na API:

***npm run start-api***

-------

**ESCREVENDO OS PRIMEIROS TESTES**

Na página que foi aberta depois de fazer ***npx cypress open***, clique em **Create new spec**.

**Spec (especificação):** Arquivo de cenários de teste.

-------

**ACESSANDO UMA URL**

Forma correta de visitar uma url em um teste no Cypress:

*cy.visit('url')*

**CONSULTANDO UM ELEMENTO**

*get('h1').contains('texto')*

Para consultar um elemento usamos o get() e para verificar se ele possui um determinado texto podemos utilizar o .contains(), de forma encadeada.

-------

**DATA ATRIBUTOS**

**SELEÇÃO DE ELEMENTOS**

É uma prática recomendada utilizar data-atributos para fornecer contexto aos seletores e isolá-los das alterações de CSS ou JavaScript. Para evitar problemas com seus testes, você deve escrever seletores que sejam resilientes e resistam a mudanças no código.

Documentação sobre:

<https://docs.cypress.io/guides/references/best-practices#Selecting-Elements>



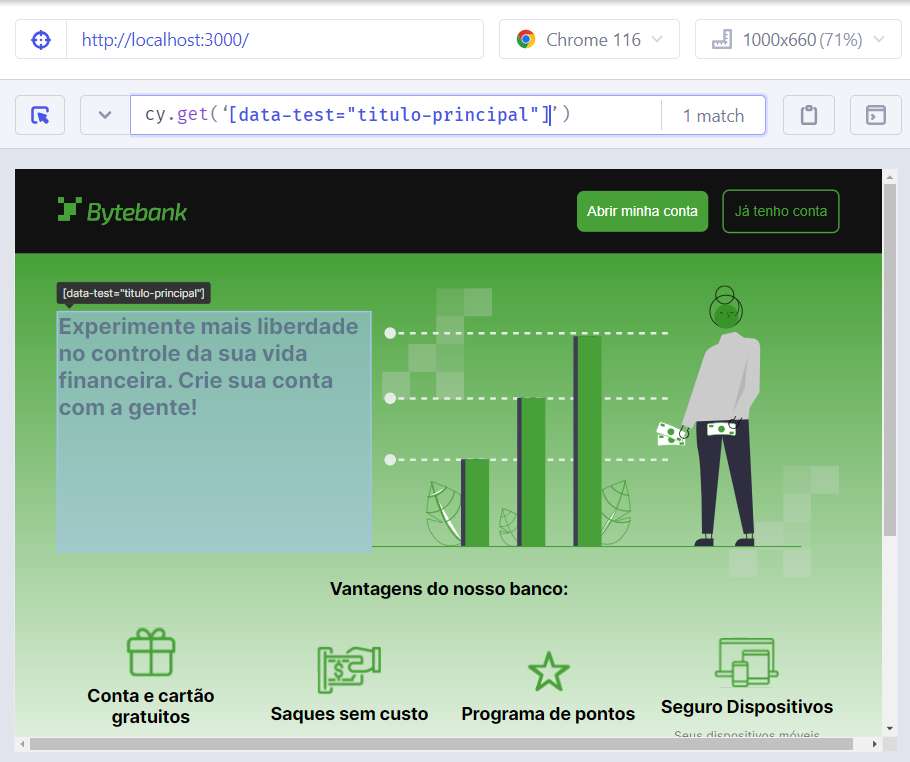
-------

Criar um teste que verifique se o texto que fala sobre as vantagens do Bytebank está correto ou não.

|  |
| --- |
| describe(**'template spec'**, () => {  it(**'passes'**, () => {  cy.visit(*'http://localhost:3000/'*);  });  it(**'Deve renderizar corretamente o texto da seção de vantagens '**, () => {  cy.visit(*'http://localhost:3000/'*);  cy.get('**h2**').contains(**'Vantagens do nosso banco:'**);  });  }); |

-------

**BUSCAR DATA ATTRIBUTE**



Do lado da url tem um símbolo de alvo. Clicando sobre ele e depois no elemento desejado, vai aparecer o data attribute.

-------

**CRIANDO COMANDOS PERSONALIZADOS**

**Evitar repetições**

Na pasta *cypress/support/****command.js*** ficam os comandos personalizados.

command.js:

|  |
| --- |
| Cypress.Commands.add(**'getByData'**, (**seletor**) => {  return cy.get(`[data-test=${**seletor**}]`);  }); |

**'getByData' 🡪 nome do comando**

**seletor 🡪 parâmetro, atributo**

ANTES:

|  |
| --- |
| describe(**'Página inicial'**, () => {  it(**'Deve renderizar o h1 com o texto correto'**, () => {  cy.visit('http://localhost:3000/');  *cy.****get****('[data-test="titulo-principal"]').contains(*  *'Experimente mais liberdade no controle da sua vida financeira. Crie sua conta com a gente!'*  *);*  });  it('Deve renderizar corretamente o texto da seção de vantagens ', () => {  cy.visit('http://localhost:3000/');  cy.get('h2').contains('Vantagens do nosso banco:');  });  }); |

DEPOIS:

|  |
| --- |
| describe(**'Página inicial'**, () => {  it(**'Deve renderizar o h1 com o texto correto'**, () => {  cy.visit('http://localhost:3000/');  *cy.****getByData****('titulo-principal').contains(*  *'Experimente mais liberdade no controle da sua vida financeira. Crie sua conta com a gente!'*  *);*  });  it('Deve renderizar corretamente o texto da seção de vantagens ', () => {  cy.visit('http://localhost:3000/');  cy.get('h2').contains('Vantagens do nosso banco:');  });  }); |

**SAIBA MAIS SOBRE COMANDOS PERSONALIZADOS:**

<https://docs.cypress.io/api/cypress-api/custom-commands#Syntax>

-------

**BEFORE EACH**

Ao vez de usar o *cy.visit* antes de cada teste podemos coloca-lo dentro de ***beforeEach***. Ou seja, antes de qualquer um dos testes, o que está dentro do call-back de beforeEach será chamado.

|  |
| --- |
| describe('Página inicial', () => {  ***beforeEach****(() => {*  *cy.visit('http://localhost:3000/');*  *});*  it('Deve renderizar o h1 com o texto correto', () => {  cy.getByData('titulo-principal').contains(  'Experimente mais liberdade no controle da sua vida financeira. Crie sua conta com a gente!'  );  });  it('Deve renderizar corretamente o texto da seção de vantagens ', () => {  cy.get('h2').contains('Vantagens do nosso banco:');  });  }); |

-------

**quando usar o cy.contains()?**

*Se o conteúdo do elemento fosse alterado, eu gostaria que o teste falhasse?*

Se a resposta for sim, então use o comando contains(). Agora se a resposta for não, então use um data-atribute.

-------

**Encadeamento de comandos**

|  |
| --- |
| cy.get(".todo-list li").find("label").should("contain", "To-do List") |

Nesta linha temos três comandos encadeados. Vamos entender passo a passo o comando:

- Estamos buscando o elemento com o **get()** que fornecerá um elemento <li /> para o próximo comando da sequência, o **find()**;

- Então o **find()** procurará um elemento **<label/>** dentro do <li />;

- Ao encontrar esta label, finalmente o comando **should()** pode fazer uma asserção, ou seja, verificar se a label possui ou não o texto “To-do List”.

**SAIBA MAIS:**

<https://docs.cypress.io/guides/core-concepts/introduction-to-cypress#Chains-of-Commands>

-------

**Comando personalizado com dois parâmetros**

Criar um comando personalizado que recebe dois parâmetros, um seletor e um texto e verifica se esse seletor possui o texto informado.

|  |
| --- |
| Cypress.Commands.add(**'verificaTexto'**, (**seletor**, **texto**) => {  cy.get(`${**seletor**}`).contain(`${**texto**}`);  }); |

-------

**TESTANDO FORMULÁRIOS**

**Realizando login**

|  |
| --- |
| describe(**'Formulário de Login'**, () => {  beforeEach(() => {  cy.visit('http://localhost:3000/');  });  it(*'Não deve permitir um email inválido'*, () => {  cy.getByData('botao-login').click();  cy.getByData('email-input').type('neilton@alura');  cy.getByData('senha-input').type('123456');  cy.getByData('botao-enviar').click();  cy.getByData('mensagem-erro')  .should('exist')  .and('have.text', 'O email digitado é inválido');  });  }); |

Explicando:

*cy.getByData('botao-login').click(); 🡪* clica no botão de login

*cy.getByData('email-input').type('neilton@alura');* 🡪 digita o email

*cy.getByData('senha-input').type('123456');* 🡪 digita a senha

*cy.getByData('botao-enviar').click();* 🡪 clica no botão de enviar

*cy.getByData('mensagem-erro').should('exist').and('have.text', 'O email digitado é inválido');* 🡪 vê se a mensagem-erro existe e se tem o texto descrito.

-------

**ENVIANDO FORMULÁRIO VAZIO**

|  |
| --- |
| it(***'Não deve permitir um campo em branco'***, () => {  cy.getByData('botao-login').click();  cy.getByData('senha-input').type('123456');  cy.getByData('botao-enviar').click();  cy.getByData('mensagem-erro')  .should('exist')  .and('have.text', ***'O campo email é obrigatório'***);  }); |

Quase a mesma coisa do exemplo anterior. Só não passei o type para o campo do email, dessa forma ficou vazio.

-------

**FAZENDO UM CADASTRO**

|  |
| --- |
| describe(**'FormularioCadastro'**, () => {  beforeEach(() => {  cy.visit('http://localhost:3000');  });  it(***'Usuário deve conseguir se cadastrar com sucesso'***, () => {  cy.getByData('botao-cadastro').click();  cy.getByData('nome-input').type('Jose');  cy.getByData('email-input').type('jose3@email.com');  cy.getByData('senha-input').type('123456');  cy.getByData('botao-enviar').click();  cy.getByData('mensagem-sucesso')  .should('exist')  .and('have.text', 'Usuário cadastrado com sucesso!');  });  it(***'Não deve permitir o cadastro de usuários sem campo nome preenchido'***, () => {  cy.getByData('botao-cadastro').click();  cy.getByData('email-input').type('maria@email.com');  cy.getByData('senha-input').type('123456');  cy.getByData('botao-enviar').click();  cy.getByData('mensagem-erro')  .should('exist')  .and('have.text', 'O campo de nome é obrigatório');  });  }); |

-------

**UM TESTE DE CADA VEZ**

Assim como outras ferramentas de testes, o Cypress possui um comando que permite executar um único caso de teste entre vários.

|  |
| --- |
| it.**only**('Descrição do teste', () => {//teste...}) |

-------

**Cypress com Testing Library**

Dá para usar o Cypress com o Testing Library.

Primeiro você precisa fazer a instalação da biblioteca no seu projeto. Esta biblioteca permite o uso de consultas de teste dom nos testes de ponta a ponta do Cypress. Para instalar é só digitar o seguinte comando no terminal:

|  |
| --- |
| npm install --save-dev cypress @testing-library/cypress |

Para usar é bem simples, você precisa importar o testing library no arquivo commands.js, dentro da pasta cypress/support.

|  |
| --- |
| import '@testing-library/cypress/add-commands' |

Agora você pode usar todos os comandos do Testing Library DOM como o findBy, findAllBy, queryBy. Um detalhe importante é que as consultas get do Testing Library não são suportadas no Cypress.

Exemplo:

|  |
| --- |
| cy.**findByRole**('button', {name: /Jackie Chan/i}).click()  // ou então  cy.get('form')  .**findByRole**('button', {name: /Button Text/i})  .should('exist')  cy.**findByRole**('dialog').within(() => {  cy.**findByRole**('button', {name: /confirm/i})  }) |

-------

**baseUrl**

Nós vamos usar a URL *http://localhost:3000* em todos os testes. Para evitar repetição, seria interessante configurar uma URL base para tornar mais simples o acesso a ela.

Vá em **Cypress.config.js**:

|  |
| --- |
| const { defineConfig } = require('cypress');  module.exports = defineConfig({  e2e: {  ***baseUrl****: 'http://localhost:3000',*  },  }); |

Agora digitar **‘/’** para que entenda que a url é essa que foi especificada em baseUrl.

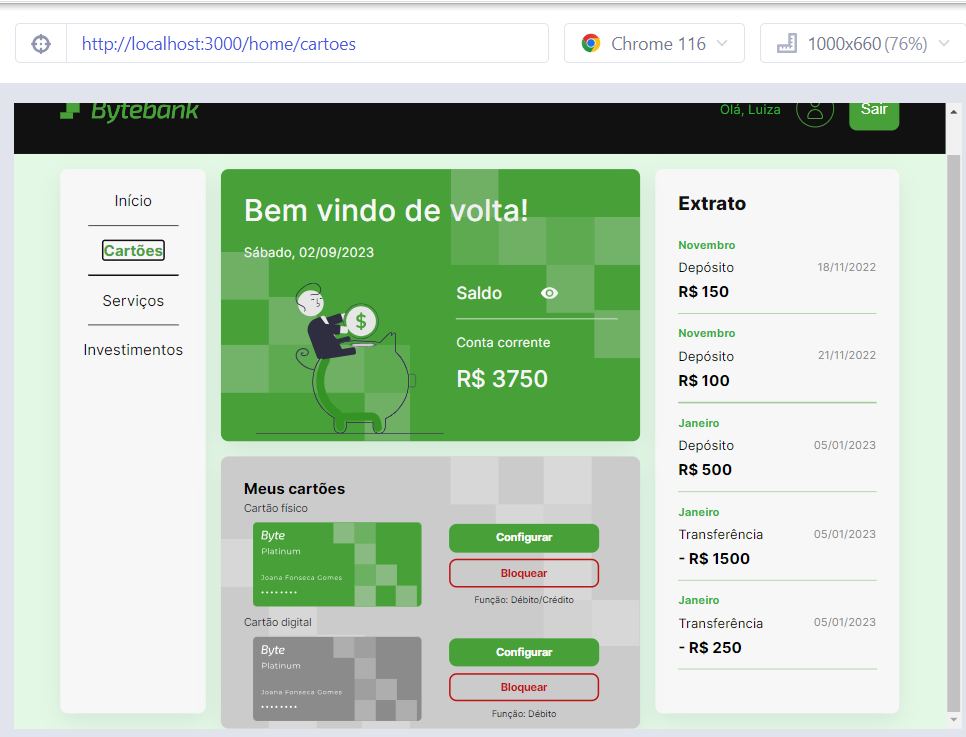
|  |
| --- |
| describe(**'Testando múltiplas páginas'**, () => {  it('Deve conseguir acessar a página de cartões', () => {  cy.visit('/');  });  }); |

-------

**ACESSANDO OUTRAS PÁGINAS**

|  |
| --- |
| describe(**'Testando múltiplas páginas'**, () => {  it(**'Deve conseguir acessar a página de cartões'**, () => {  cy.visit('/');  cy.getByData('botao-login').click();  cy.getByData('email-input').type('luiza@email.com');  cy.getByData('senha-input').type('123456');  cy.getByData('botao-enviar').click();  cy.getByData('app-home').**find**('a').**eq**(1).**click**();  cy.getByData('titulo-cartoes')  .should('exist')  .and('have.text', 'Meus cartões');  });  }); |

*cy.getByData('app-home').find('a').eq(1).click(); 🡪* só o elemento pai tinha um data test atribute, então pegamos ele e procuramos algum filho com a tag **‘a’**. Na página tem 4 elementos ‘a’, queremos o segundo, por isso fizemos **.eq(1)**.



-------

**Verificando url**

|  |
| --- |
| describe('Testando múltiplas páginas', () => {  it('Deve conseguir acessar a página de cartões', () => {  cy.visit('/');  cy.getByData('botao-login').click();  cy.getByData('email-input').type('luiza@email.com');  cy.getByData('senha-input').type('123456');  cy.getByData('botao-enviar').click();  cy.**location**('**pathname**').should(**'eq'**, **'/home'**);  cy.getByData('app-home').find('a').eq(1).click();  cy.getByData('titulo-cartoes')  .should('exist')  .and('have.text', 'Meus cartões');  cy.**location**(**'pathname'**).should(**'eq'**, **'/home/cartoes'**);  });  }); |

*cy.****location****('****pathname****').should(****'eq'****,* ***'/home'****);* 🡪 Para saber se o ‘pathname’ é igual a ‘/home’.

-------

**JORNADA DO USUÁRIO**

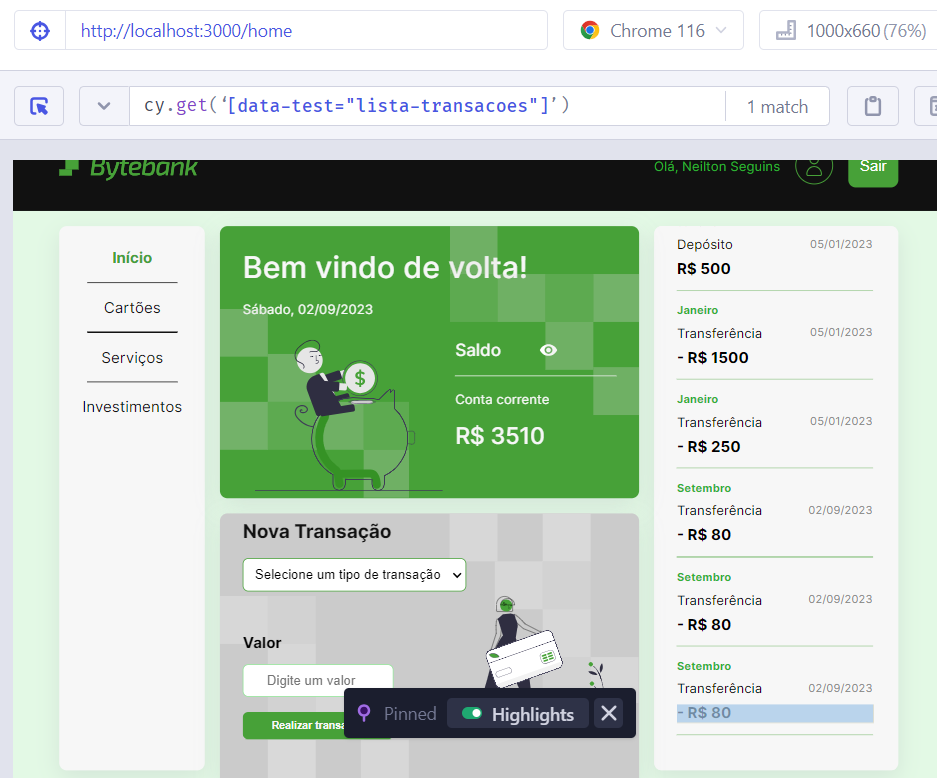
Podemos imaginar a jornada de usuário como se fosse os caminhos essenciais que a pessoa usuária pode percorrer ao utilizar uma aplicação.

cypress/e2e/**jornadaUsuario.cy.js**:

|  |
| --- |
| describe('Jornadas de usuário', () => {  it(**'Deve permitir que a pessoa usuária acesse a aplicação, realize uma transação e faça um logout'**, () => {  cy.visit('/');  cy.getByData('botao-login').click();  cy.getByData('email-input').type('neilton@alura.com');  cy.getByData('senha-input').type('123456');  cy.getByData('botao-enviar').click();  cy.location('pathname').should('eq', '/home');  cy.getByData('select-opcoes').select('Transferência');  cy.getByData('form-input').type('80');  cy.getByData('realiza-transacao').click();  cy.**getByData**('lista-transacoes').**find**('li').**last**().**contains**('- R$ 80');  cy.getByData('botao-sair').click();  cy.location('pathname').should('eq', '/');  });  }); |

*cy.getByData('select-opcoes').select('Transferência'); 🡪* Seleciona o opção em **Nova Transação.**

*cy.getByData('lista-transacoes').find('li').last().contains('- R$ 80');* 🡪 encontra a ‘li’ dentro de ‘lista-transacoes’ e pegar o item do array(do ul).



-------

**Encontrando um elemento**

|  |
| --- |
| <nav id="menu">  <ul>  <li><a href='/início'>Início</a></li>  <li><a href='/sobre'>Sobre</a></li>  <li><a href='/contato'>Contato</a></li>  </ul>  </nav> |

Você não pode modificar o código HTML, mas precisa escrever uma linha de código para clicar no link de Contato.

|  |
| --- |
| cy.**get**('#menu').**find**('a').**eq**(2).click() |

É exatamente assim que mesmo na ausência das boas práticas de uso de data-atributes conseguimos testar cenários de nossa aplicação. Neste exemplo, o comando get() busca o elemento que possui o id menu, e dentro desse elemento o find() procura por aquele que corresponde a uma tag <a /> e seleciona o terceiro, fazendo um click nele após isso.

-------

**Identificando o que testar**

**Jornadas do usuário**

Se você estiver testando uma aplicação já existente, é recomendado que comece escrevendo os testes para as **partes mais críticas** do seu aplicativo. Considere as partes mais críticas do seu aplicativo aquelas que não podem ser interrompidas ou quebradas, como o login/autenticação, compra de um produto, processamento de cartões de crédito, formulários de cadastro, etc. Escreva os primeiros testes de suas aplicações focados nestas partes e que eles sejam testes de ponta a ponta.

Depois de testar as áreas críticas e portanto mais importantes, escreva testes para jornada de usuário. As jornadas de usuário são os caminhos essenciais percorridos por um usuário do seu aplicativo. Toda a jornada do usuário deve ser testada em um único teste, e a razão disso é para ter certeza de que todas as partes do seu aplicativo estão funcionando corretamente.

Veja esse link:

<https://learn.cypress.io/testing-foundations/testing-is-a-mindset>

-------

**CYPRESS SEM CABEÇA (HEADLESS)**

**DIRETO NO TERMINAL**

Até agora rodamos nossos testes com o comando ***npx cypress open***. Isso faz abrir uma guia de interface do usuário do Cypress, onde podemos selecionar o tipo de teste que estamos fazendo, o navegador e as especificações onde queremos rodar nossos testes.

Porém, quando trabalhamos com integração contínua, por exemplo, não queremos subir essa guia de interface do usuário do próprio Cypress, mas precisamos trabalhar mais no **terminal integrado**.

No terminal do vscode:

***npx cypress run***

Esse comando executará algumas configurações e as nossas especificações, que deixamos na pasta **"cypress > e2e"**. Ao final, ele mostra um relatório das especificações que rodaram.

Era pra ter aparecido algo nas pastas **video** e **screenshots**. Ver na documentação:  
<https://docs.cypress.io/guides/guides/screenshots-and-videos>

E SE EU QUISER RODAR APENAS UMA ESPECIFICAÇÃO?

***npx cypress run --spec "caminho-do-arquivo-do-teste"***

***npx cypress run –spec "./cypress/e2e/paginas.cy.js"***

-------

**DIFERENTES NAVEGADORES**

Escolhendo o navegador no terminal:

***npx express run –browser nomedonavegador***

***npx express run –browser edge***

Ou dentro da própria especificação:

|  |
| --- |
| describe('Testando múltiplas páginas', () => {  it('Deve conseguir acessar a página de cartões', **{browser: 'edge'}**, () => {  cy.visit('/');  cy.getByData('botao-login').click();  cy.getByData('email-input').type('luiza@email.com');  cy.getByData('senha-input').type('123456');  cy.getByData('botao-enviar').click();  cy.location('pathname').should('eq', '/home');  cy.getByData('app-home').find('a').eq(1).click();  cy.getByData('titulo-cartoes')  .should('exist')  .and('have.text', 'Meus cartões');  cy.location('pathname').should('eq', '/home/cartoes');  });  }); |

-------

**DIFERENTES TAMANHOS DE TELAS**

*npx cypress open* ***--config viewportWidth=375, viewportHeight=667***

É possível controlar o tamanho da janela de exibição em nossos testes, seja adicionando uma configuração no ***cypress.config.js*** ou **via linha de comando** ao executar os testes. Mas também é possível definir o viewport dentro de nossos testes mesmo.

Existe um comando do Cypress chamado de viewport(), onde você consegue definir a altura e largura da janela de exibição da aplicação.

Com isso, você pode organizar seus testes para desktop e mobile dentro das próprias especificações.

|  |
| --- |
| describe(**'Menu de navegação'**, () => {  context(**'Resolução de 720p'**, () => {  beforeEach(() => {  // Roda os testes como se fossem em um monitor de 720p de resolução  *cy.****viewport****(1280, 720)*  })  // seu teste aqui  })  context(**'Resolução do iphone-5 '**, () => {  beforeEach(() => {  // roda os testes como se fossem em um dispositivo com a resolução de um iphone-5  *cy.****viewport****('iphone-5')*  })  // seu teste aqui  })  }) |

**SAIBA MAIS:**

<https://docs.cypress.io/api/commands/viewport#docusaurus_skipToContent_fallback>

OUTRO EXEMPLO:

|  |
| --- |
| describe(**'Menu de navegação burguer icon'**, () => {  context(**'Resolução de iphone xr'**, () => {  beforeEach(() => {  cy.viewport('iphone-xr');  });  it('Deve existir um botão menu burguer', () => {  cy.visit('/');  cy.getByData('botao-login').click();  cy.getByData('email-input').type('neilton@alura.com');  cy.getByData('senha-input').type('123456');  cy.getByData('botao-enviar').click();  cy.location('pathname').should('eq', '/home');  cy.getByData('menu-burguer').should(**'be.visible'**);  });  });  context(**'Resolução do mackbook 13'**, () => {  beforeEach(() => {  cy.viewport('macbook-13');  });  it('Deve existir um botão menu burguer', () => {  cy.visit('/');  cy.getByData('botao-login').click();  cy.getByData('email-input').type('neilton@alura.com');  cy.getByData('senha-input').type('123456');  cy.getByData('botao-enviar').click();  cy.location('pathname').should('eq', '/home');  cy.getByData('menu-burguer').should(**'not.be.visible'**);  });  });  }); |

Um detalhe importante é que realizamos uma **asserção** para ambos os casos. Primeiro, para o caso do iphone xr onde nosso menu burguer DEVE aparecer, escrevemos.

E para o caso do macbook 13 onde o menu burguer NÃO DEVE aparecer, fizemos.

SAIBA MAIS SOBRE **ASSERÇÕES**:

<https://docs.cypress.io/guides/references/assertions>

-------------------------------------------------------------------------------------------------------------------

**React: avançando em testes e2e com Cypress**

*https://cursos.alura.com.br/course/react-avancando-testes-e2e-cypress*

Api:

***npm i***

***npm run dev***

Arquivos React:

***npm i***

***code .***

***npm start***

***npx cypress open***

-------

**Comando personalizado de Login**

Seria interessante termos um comando personalizado, pois precisaremos realizar o login sempre que acessarmos a aplicação. Desta forma, informaremos somente o e-mail e a senha e poderemos realizar o login.

Na pasta **support** e arquivo **commands.js**:

|  |
| --- |
| Cypress.Commands.add(**'login'**, (**email**, **senha**) => {  cy.**session**([email, senha], () => {  cy.visit('/');  cy.getByData('botao-login').click();  cy.getByData('email-input').type(**email**);  cy.getByData('senha-input').type(**senha**);  cy.getByData('botao-enviar').click();  cy.url().should('contain', '/home');  });  }); |

Em **e2e**> **formularioLogin.cy.js**:

|  |
| --- |
| describe(**'Formulario de Login'**, () => {  it.only('Deve acessar a página home', () => {  cy.**login**(**'neilton@alura.com'**, **'123456'**);  cy.visit('/home');  cy.getByData('titulo-boas-vindas').should('contain', 'Bem vindo de volta!');  });  it('Não deve permitir um email inválido', () => {  cy.getByData('botao-login').click();  cy.getByData('email-input').type('neilton@alura');  cy.getByData('senha-input').type('123456');  cy.getByData('botao-enviar').click();  cy.getByData('mensagem-erro')  .should('exist')  .and('have.text', 'O email digitado é inválido');  });  it('Não deve permitir um campo em branco', () => {  cy.getByData('botao-login').click();  cy.getByData('senha-input').type('123456');  cy.getByData('botao-enviar').click();  cy.getByData('mensagem-erro')  .should('exist')  .and('have.text', 'O campo email é obrigatório');  });  }); |

-------

**cy.session()**

Para salvar as informações de login em cache. Isso ajuda a não depender de ganchos externos e nem de estados de testes anteriores.

Com o cy.session() podemos salvar o estado da sessão atual e reutilizá-lo em sessões futuras pois eles serão armazenados em cache. Podemos inclusive recuperar dados do **localStorage** e **SessionStorage**

Mais:  
O Cypress oferece as funções **Cypress.session**, **clearAllLocalStorage** e **clearAllSessionStorage** para gerenciar o armazenamento do navegador. A função **Cypress.session** permite armazenar dados em um objeto de sessão global para serem compartilhados entre diferentes testes. Já as funções **clearAllLocalStorage** e **clearAllSessionStorage** limpam o armazenamento local e de sessão, respectivamente.

Um exemplo de como usar esses métodos seria um teste que verifica se um usuário pode fazer login e permanecer logado após uma atualização da página. Usando o Cypress.session, podemos armazenar as informações do usuário após o login e usá-las em diferentes partes do teste. Depois disso, podemos limpar todos os dados de armazenamento usando os métodos clearAllLocalStorage e clearAllSessionStorage antes de atualizar a página e verificar se o usuário ainda está logado.

|  |
| --- |
| describe('Teste de login e sessão', () => {  it('Login e permanência na sessão', () => {  cy.visit('/login')  cy.get('#username').type('usuario')  cy.get('#password').type('senha')  cy.get('#login-btn').click()  cy.session({name: 'user'}).then((user) => {  expect(user.username).to.eq('usuario')  expect(user.token).to.not.be.empty  })  cy.**clearAllLocalStorage()**  cy.**clearAllSessionStorage()**  cy.reload()  cy.session({name: 'user'}).then((user) => {  expect(user).to.be.null  })  })  }) |

Neste exemplo, estamos armazenando as informações do usuário na sessão com a chave 'user'. Depois de limpar todos os dados de armazenamento, verificamos se o usuário ainda está armazenado na sessão, o que não deve ser o caso.

Você também pode usar o **clearAllSessionStorage** e o **clearAllLocalStorage** para deletar dados que possam estar salvos em cache como tokens de usuário, id, senhas, etc.

-------

**O que testar primeiro?**

Ao testar uma aplicação, pode ser desafiador decidir por onde começar. É importante pensar na prioridade dos testes e quais são as áreas mais críticas da aplicação. A seguir, algumas dicas para ajudar a decidir o que testar primeiro:

**- Priorize as áreas críticas:** Considere as áreas da aplicação que são mais importantes para o sucesso do negócio ou que têm um impacto maior nos usuários. Por exemplo, se você estiver testando um aplicativo de e-commerce, é importante garantir que os usuários possam fazer compras com facilidade.

**- Foque nos fluxos principais:** Em vez de tentar testar todos os recursos de uma só vez, concentre-se nos fluxos principais da aplicação. Por exemplo, em um aplicativo de viagens, o fluxo principal pode ser pesquisar por um destino, selecionar um voo e fazer uma reserva.

**- Priorize os testes com falhas frequentes:** Se houver testes que falham com frequência, considere priorizá-los para resolvê-los rapidamente e evitar desperdício de tempo.

**- Comece com os testes mais simples:** Às vezes, é melhor começar pelos testes mais simples para ganhar impulso e garantir que a configuração de testes esteja funcionando corretamente.

Ao decidir o que testar primeiro, é importante pensar em termos de prioridades e impacto nos usuários e no negócio. Priorizar testes críticos pode ajudar a evitar problemas graves na produção e garantir uma melhor qualidade de software.

No Cypress, podemos configurar e utilizar testes de ponta a ponta seguindo essas prioridades, organizando nossos testes de forma a testar as áreas críticas e os fluxos principais em primeiro lugar. Podemos utilizar os comandos do Cypress para definir fluxos de teste, priorizar testes com falhas frequentes e garantir que as configurações de testes estejam funcionando corretamente.

-------

**Testando a API**

**Testando requisições**

Dentro da pasta e2e, criar a pasta api e o arquivo **api-users.cy.js**:

|  |
| --- |
| describe(**'Realizando requisições para a API'**, () => {  context(**'GET /users'**, () => {  it(**'Deve retornar uma lista de usuários'**, () => {  cy.**request**('GET', 'http://localhost:8000/users').then((***response***) => {  expect(***response***.status).to.eq(200);  expect(***response***.body).length.to.be.greaterThan(1);  });  });  });  }); |

-------

**Testando falhas na api**

|  |
| --- |
| describe(**'Realizando requisições para a API'**, () => {  context('GET /users', () => {  it('Deve retornar uma lista de usuários', () => {  cy.request('GET', 'http://localhost:8000/users').then((response) => {  expect(response.status).to.eq(200);  expect(response.body).length.to.be.greaterThan(1);  });  });  });  context(**'GET /users/:userId'**, () => { *🡪 teste feliz*  it(**'Deve retornar um único usuário'**, () => {  cy.**request**({  method: 'GET',  url: 'http://localhost:8000/users/40a41438-84a6-4b4d-ae1d-7f1713d0a9fe',  }).then((**response**) => {  expect(**response.status**).to.eq(**200**);  expect(**response.body**).to.have.property(**'nome'**);  });  });  it(**'Deve retornar um erro quando o usuário for inválido'**, () => { *🡪 teste triste*  cy.**request**({  method: 'GET',  url: 'http://localhost:8000/users/40a41438-84a6-4b4d',  failOnStatusCode: false, *🡪 isso faz com que não falhe o teste antes que eu verifique no then*  }).then((**response**) => {  expect(**response.status**).to.eq(**404**);  expect(**response.body**).to.eq(**'Not Found'**);  });  });    });  }); |

-------

**Interceptações de rede**

Em alguns cenários, quando estamos testando nossa API, pode acontecer dos nossos testes falharem. O motivo é que a API pode estar desligada, por exemplo, porque o servidor deu defeito.

Esse problema acontece quando usamos a URL da API diretamente nos testes e dependemos dela estar ligada para que eles passem. Há uma maneira de burlar essa situação e ter dados que não necessariamente são da API, mas que são bastante semelhantes.

|  |
| --- |
| describe(**'Realizando requisições para a API'**, () => {  context(**'Interceptando solicitações de rede'**, () => {  it(**'Deve fazer a inteceptação do POST users/login'**, () => {  cy.**intercept**('POST', 'users/login').as(**'loginRequest'**);  cy.**login**('neilton@alura.com', '123456');  cy.**wait**('**@loginRequest'**).then((**interception**) => {  **interception**.response = {  statusCode: 200,  body: {  sucess: true,  message: 'Login bem sucedido',  },  };  });  cy.**visit**('/home');  cy.getByData('titulo-boas-vindas').should(  'contain.text',  'Bem vindo de volta!'  );  });  });  }); |

Nós interceptamos a rota - no momento em que estamos fazendo um **POST**, quando realizamos o login na aplicação - e devolvemos informações que queríamos: o **statusCode** de 200 e um novo usuário.

**OUTRO EXEMPLO DE INTERCEPTAÇÃO:**

|  |
| --- |
| describe(**'Testes de Interceptação de Solicitações de Rede'**, () => {  beforeEach(() => {  cy.**intercept**('GET', '/api/clientes/1', (req) => {  **req.reply**((res) => {  **res.body.nome** = **'Cliente A'**;  });  }).as(**'clientes'**);  });  it(**'Deve interceptar a solicitação GET /api/clientes/1 e modificar o nome do cliente'**, () => {  cy.visit('/');  cy.**wait**(**'@clientes'**); *🡪 esse é um alias*  cy.**get**('.cliente-nome').should('have.text', 'Cliente A');  });  it(**'Deve interceptar a solicitação POST /api/usuarios e retornar um objeto com a propriedade "email"'**, () => {  cy.**intercept**(**'POST'**, '/api/usuarios', (req) => {  req.reply((res) => {  res.body = {  **email: 'usuario@exemplo.com'**,  };  });  });  cy.visit('/cadastro');  cy.get('#nome').type('Usuário A');  cy.get('#email').type('usuario@exemplo.com');  cy.get('#senha').type('123456');  cy.get('#botao-cadastrar').click();  cy.get('.mensagem-sucesso').should('have.text', 'Usuário cadastrado com sucesso');  });  }); |

A **req.reply()** função pode ser usada para enviar uma resposta **stub** para uma solicitação interceptada.

- A primeira solicitação interceptada é uma solicitação GET para a URL '/api/clientes/1' que modifica o nome do cliente.

- A segunda solicitação interceptada envia no corpo da solicitação um email de um usuário fictício.

-------

**ALIASES**

Os **aliases** funcionam como variáveis em seus testes. Quando você cria um alias, você dá um nome a um objeto ou resultado específico. Esse nome pode ser usado em outras etapas do teste para se referir a esse objeto ou resultado. Em outras palavras, você pode usar aliases para compartilhar dados entre as diferentes etapas de um teste.

Existem várias maneiras de usar **aliases** no Cypress. Por exemplo, você pode criar **aliases** para elementos HTML, para objetos de solicitação de rede, para resultados de chamadas assíncronas, entre outras coisas. Para criar um alias, você usa o método **.as()** do Cypress, seguido pelo nome que deseja dar ao alias.

Um exemplo comum de uso de aliases é quando você precisa fazer uma série de ações em um elemento específico da página. Em vez de pesquisar o elemento toda vez que precisar acessá-lo, você pode criar um alias para ele e, em seguida, usar esse alias em outras etapas do teste. Por exemplo:

|  |
| --- |
| cy.get(**'#meu-elemento'**).as(***'meuElemento'***)  cy.get(***'@meuElemento'***).click()  cy.get(***'@meuElemento'***).type('Texto de exemplo') |

Nesse exemplo, usamos o alias "@meuElemento" para se referir a um elemento específico na página. Primeiro, criamos o alias usando o método .as(). Em seguida, usamos o alias para clicar no elemento e digitar algum texto nele. Usando o alias, não precisamos pesquisar o elemento novamente em cada etapa do teste.

Outra maneira de usar aliases é para se referir a **objetos de solicitação de rede**. Com o Cypress, você pode interceptar e inspecionar solicitações de rede feitas pela sua aplicação. Quando você intercepta uma solicitação, pode criar um alias para ela e, em seguida, usar esse alias para verificar seus detalhes em diferentes etapas do teste. Por exemplo:

|  |
| --- |
| cy.**intercept**('/api/usuarios').as(**'solicitacaoDeUsuarios'**)  cy.**visit**('/minha-pagina')  cy.**wait**(**'@solicitacaoDeUsuarios'**).then((interception) => {  expect(interception.response.statusCode).to.equal(200)  expect(interception.response.body.length).to.equal(10)  }) |

Nesse exemplo, usamos o alias ***"@solicitacaoDeUsuarios"*** para se referir a uma solicitação de rede específica feita pela aplicação.

Usamos o comando ***cy.intercept()*** para interceptar a solicitação e criar o alias.

Em seguida, visitamos uma página e esperamos pela interceptação usando o comando ***cy.wait().*** Quando a interceptação ocorre, usamos o alias para verificar o código de status da resposta e o número de usuários retornados.

-------

**mocks e stubs**

Tanto mocks quanto stubs são técnicas de simulação de comportamentos em testes automatizados.

**Mocks:** são usados para verificar a interação do sistema com um objeto simulado. Ou seja, testam o comportamento do sistema. Podemos usar um mock para verificar se um determinado método foi chamado durante a execução do teste.

**Stubs:** são usados para fornecer uma resposta pré-definida a uma determinada chamada de função ou método. Ou seja, simulam o comportamento do sistema. Podemos usar um stub para simular uma resposta de um serviço externo que ainda não está disponível para testes.

-------

**Realizando login via API**

Realizar login em uma aplicação via API é uma tarefa comum em testes automatizados. No Cypress, isso pode ser feito usando o comando cy.request() para enviar uma requisição HTTP para a API de login.

**1º** - é necessário obter as informações de autenticação, como o nome de usuário e a senha.

**2º** - podemos enviar uma requisição POST para a API de login, com os dados de autenticação no corpo da requisição.

|  |
| --- |
| describe(**'Teste de login via API'**, () => {  it(**'Deve permitir o login de um usuário válido'**, () => {  cy.**request**({  method: '**POST**',  url: '/api/login',  body: {  username: 'usuario@login.com',  password: 'Shxksi49385!%'  }  }).then((**response**) => {  expect(**response**.status).to.eq(200)  expect(**response**.body).to.have.property('token')  })  })  }) |

- Neste exemplo, enviamos uma requisição POST para a API de login, com os dados de autenticação no corpo da requisição. Usamos o método then() para verificar a resposta da API, verificando se o status da resposta é 200 e se a resposta contém um token de autenticação retornado pela API.

É importante lembrar que este exemplo é apenas uma simplificação do processo de login via API. Em aplicações reais, pode haver outros passos de autenticação, como a geração de tokens de acesso ou o uso de autenticação de dois fatores, por exemplo. Por isso, é importante entender bem o fluxo de autenticação da aplicação e implementar o teste de acordo com os requisitos específicos da mesma.

***Importante!***  
Realizar o login via API em seus testes pode te ajudar a otimizar bastante eles, tornando-os mais rápidos e eficientes. Mas dependerá muito do que você está testando, qual o cenário de testes está sendo considerado, por exemplo, se você precisar apenas realizar o login para acessar a aplicação é uma boa usar o login via API. Caso você queira testar também a interface do usuário no processo de login aí é recomendado utilizar o login via interface do usuário.

-------

**REQUISIÇÕES PUT**

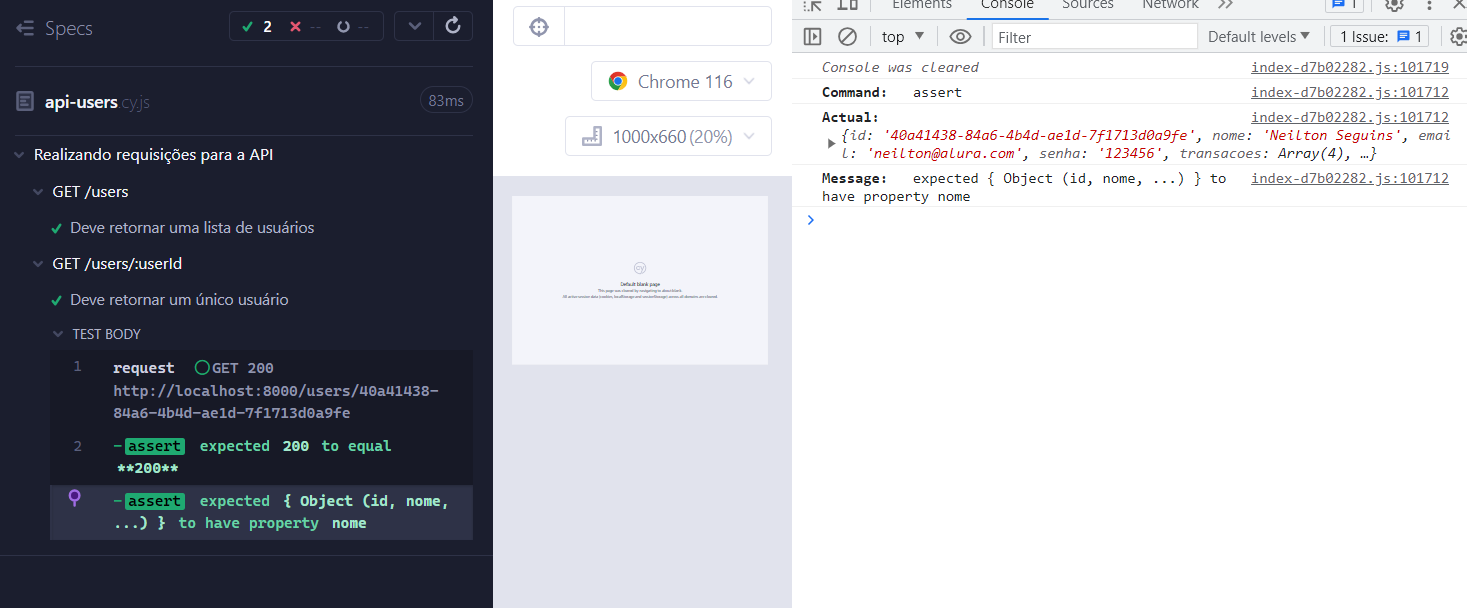
|  |
| --- |
| context(**'Teste método PUT da API Usuários'**, () => {  it(**'Deve atualizar informações do usuário com sucesso'**, () => {  const **usuario** = {  nome: 'Marcos Vinicius Neves',  senha: '123456',  };  cy.**request**({  method: **'PUT'**,  url: 'http://localhost:8000/users/c691fd15-dcd5-4f24-89da-cdfa3cef9d67',  body: usuario,  failOnStatusCode: false,  }).then((response) => {  expect(response.status).to.eq(200);  expect(response.body.nome).to.eq(usuario.nome);  expect(response.body.senha).to.eq(usuario.senha);  });  });  it(**'Retorna erro 404 para usuário inexistente'**, () => {  const **usuario** = {  nome: 'Marcos Vinicius Neves',  senha: '123456',  };  cy.**request**({  method**: 'PUT'**,  url: 'http://localhost:8000/users/c691fd15-dcd5-4f24-89d',  body: usuario,  failOnStatusCode: false,  }).then((response) => {  expect(response.status).to.eq(404);  expect(response.body).to.eq('Not Found');  });  });  }); |

- Então eu faço a requisição por meio do **cy.request()** informando o **método**, **url** e as **novas informações de usuário,** e

- Verifico por meio de asserções se o retorno dessa requisição, ou seja, a resposta condiz com as novas informações atualizadas.

-------

Clicando em algum item do **test body** e clicando com o botão direto> inspecionar na viewport, vai aparecer as infos no console.



-------

**UTILIZANDO DADOS DINÂMICOS**

**Testando interface do usuário**

**Testando API**

e2e>cadastro.cy.js:

|  |
| --- |
| describe(**'Teste de cadastro de usuário'**, () => {  const **usuario** = {  nome: 'Luiza',  email: 'luiza@email.com',  senha: '123456',  };  it(***'Deve permitir cadastrar um usuário com sucesso'***, () => {  cy.visit('/');  cy.getByData('botao-cadastro').click();  cy.getByData('nome-input').type(usuario.nome);  cy.getByData('email-input').type(usuario.email);  cy.getByData('senha-input').type(usuario.senha);  cy.getByData('checkbox-input').check();  cy.getByData('botao-enviar').click({ force: true });  cy.getByData('mensagem-sucesso')  .should('exist')  .contains('Usuário cadastrado com sucesso!');  cy.request(**'GET'**, **'http://localhost:8000/users'**).then((***resposta***) => {  expect(**resposta**.body).to.have.lengthOf.at.least(1);  expect(**resposta**.body[resposta.body.length - 1]).to.deep.include(usuario);  });  });  }); |

***resposta****.body[resposta.body.length - 1]* 🡪 último usuário

***PROBLEMA!***

*Se pedirmos para rodar o teste novamente*, clicando no botão de retorno no canto superior direito da aba lateral, receberemos um erro: aquele e-mail já foi cadastrado. Com isso, *nosso teste falhará*.

Estamos alimentando a nossa aplicação com dados mockados, e isso não é interessante para um teste como esse, em que precisamos cadastrar usuários com e-mails diferentes para não haver falhas.

-------

**Gerando usuários fictícios**

Faker.js:

***npm install @faker-js/faker --save -dev***

Ao invés de fornecer esses dados inventados, vamos gerar informações de **forma aleatória**, usando a biblioteca **Faker.js**.

|  |
| --- |
| ***import { faker } from '@faker-js/faker/locale/pt\_BR';***  describe(**'Teste de cadastro de usuário'**, () => {  const usuario = {  nome: ***faker.person.fullName(),***  email: ***faker.internet.email(),***  senha: ***faker.internet.password(),***  };  it('Deve permitir cadastrar um usuário com sucesso', () => {  cy.visit('/');  cy.getByData('botao-cadastro').click();  cy.getByData('nome-input').type(usuario.nome);  cy.getByData('email-input').type(usuario.email);  cy.getByData('senha-input').type(usuario.senha);  cy.getByData('checkbox-input').check();  cy.getByData('botao-enviar').click({ force: true });  cy.getByData('mensagem-sucesso')  .should('exist')  .contains('Usuário cadastrado com sucesso!');  cy.request('GET', 'http://localhost:8000/users').then((resposta) => {  expect(resposta.body).to.have.lengthOf.at.least(1);  expect(resposta.body[resposta.body.length - 1]).to.deep.include(usuario);  });  });  }); |

-------

**Usando dados dinâmicos**

Criaremos um novo arquivo de **fixture** para um dos nossos testes, clicando na pasta **"fixtures"** com o botão direito e, depois, em "New file...". O nomeamos como, por exemplo, **usuarios.json** e o salvamos.

fixtures> **usuarios.json**:

|  |
| --- |
| [  {  "nome": "Vinicius Neves",  "email": "vinny@alura.com",  "senha": "789456"  },  {  "nome": "Gui Lima",  "email": "gui@alura.com",  "senha": "guilimadev"  },  {  "nome": "Luiza",  "email": "luiza@email.com",  "senha": "123456"  }  ] |

Em e2e>FormularioLogin.cy.js:

|  |
| --- |
| it.only(***'Deve acessar a página home'***, () => {  cy.fixture(**'usuarios'**).then((***usuario***) => {  cy.**login**(***usuario[3].email***, ***usuario[3].senha***);  cy.visit('/home');  cy.url().should('include', '/home');  cy.getByData('titulo-boas-vindas').should(  'contain',  'Bem vindo de volta!'  );  cy.**contains**(***usuario[3].nome***).should(***'be.visible'***);  });  }); |

cy.fixture(**'usuarios'**).then((***usuario***) 🡪 em fixture(‘usuarios’) seleciona o **usuarios.json** da pasta fixtures.

-------

**Fixtures**

São arquivos externos que contêm dados de teste que podem ser usados nos seus testes Cypress. Usar fixtures pode ajudar a manter o seu código de teste organizado e limpo, além de tornar mais fácil a alteração dos dados de teste sem ter que modificar o código do teste.

Para utilizar fixtures no Cypress, basta criar um arquivo de fixture na pasta **cypress/fixtures** e acessá-lo usando o comando **cy.fixture()**. Por exemplo, se você tiver um arquivo chamado ***example.json*** contendo um objeto com informações de usuário, pode-se acessar esse objeto em um teste usando o seguinte código:

|  |
| --- |
| cy.**fixture**('example.json').then((user) => {  *// Usa o objeto de usuário aqui*  } |

Os arquivos de fixture podem ser no formato **JSON, YAML ou CSV**. Quando o Cypress carrega um arquivo de fixture, ele é armazenado em cache e reutilizado em todos os testes que o utilizam. Isso pode tornar seus testes mais rápidos e eficientes.

Além disso, é possível passar dados de fixture para seu aplicativo de destino usando **cy.intercept().** Por exemplo, se seu aplicativo carrega uma lista de usuários de uma API, você pode usar uma fixture para fornecer os dados da resposta da API para o teste. Para isso, basta usar o comando **cy.intercept()** em conjunto com **cy.fixture(),** como neste exemplo:

|  |
| --- |
| cy.intercept('GET', '/api/users', (req) => {  req.**reply**(cy.fixture('users.json'))  }) |

Dessa forma, ao invés de realizar uma requisição real para a API, a resposta é obtida a partir da fixture, garantindo a previsibilidade do teste e evitando a necessidade de ter um ambiente de teste com a API disponível.

Isso pode ser bastante útil quando você não possui o serviço de API disponível e mesmo assim precisa garantir que seus testes rodem e passem. Além de tornar seus testes mais rápidos e menos dependentes de serviços externos.

-------

**faker.js**

O faker.js é uma biblioteca JavaScript que permite a geração de dados fictícios para testes. Com ela, é possível gerar nomes, endereços, números de telefone e outros tipos de dados aleatórios que podem ser úteis para testar formulários e outras funcionalidades que requerem entradas de dados.

No Cypress, podemos utilizar o faker.js para gerar dados aleatórios e utilizá-los em nossos testes automatizados. Para isso, é necessário importar o faker.js no nosso código de teste e, em seguida, utilizar as suas funções para gerar os dados necessários.

Por exemplo, podemos gerar um e-mail aleatório utilizando a função email() do faker.js:

|  |
| --- |
| import faker from 'faker';  describe(**'Testes com faker.js'**, () => {  it(***'Deve preencher um formulário com dados fictícios'***, () => {  cy.visit('https://meusite.com.br');  cy.get('#nome').type(faker.name.findName());  cy.get('#email').type(faker.internet.email());  cy.get('#telefone').type(faker.phone.phoneNumber());  cy.get('#endereco').type(faker.address.streetAddress());  cy.get('#cidade').type(faker.address.city());  cy.get('#estado').type(faker.address.state());  cy.get('#cep').type(faker.address.zipCode());  cy.get('#enviar').click();  *// assert de sucesso*  });  }); |

-------

Desafio:

|  |
| --- |
| describe('Desafio aula 03 - Testa informações de um usuário específico', () => {  it.only('Verifica informações de usuário, como transações, saldo, nome, etc', () => {  *// 'dadosUsuarios' está na pasta fixtures*  cy.**fixture**('dadosUsuarios').then((usuario) => {  cy.**login**(usuario.email, usuario.senha);  cy.**visit**('/home');  cy.**url**().should('include', '/home');  *// Verifica se o nome de usuário aparece na tela*  cy.**contains**(usuario.nome).should(‘be.visible’);  *// Verifica se o valor da última transação corresponde ao valor esperado*  cy.**getByData**('lista-transacoes')  .**find**('li')  .**last**()  .**contains**(usuario.transacoes[usuario.transacoes.length - 1].valor);  *// Verificar se o saldo corresponde ao saldo esperado*  cy.**get**('[data-testid="saldo"]').**contains**(usuario.saldo);  });  }); |

-------

**EXPLORANDO JORNADAS DE USUÁRIOS**

**Escutando eventos do dom**

|  |
| --- |
| import { faker } from '@faker-js/faker/locale/pt\_BR';  describe(***'Atualização de dados do usuário'***, () => {  const **novosDadosDeUsuario** = {  nome: faker.person.fullName(),  senha: faker.internet.password(),  };  it(***'Deve permitir o usuário atualizar seus dados'***, () => {  cy.**fixture**(**'usuarios'**).as('usuarios');  ***// pegou a fixture e fez login com o usuario do arquivo usuarios.json***  cy.get(**'@usuarios'**).then((usuario) => {  cy.**login**(usuario[3].email, usuario[3].senha);  cy.visit('/home');  cy.url().should('include', '/home');  ***//na página home deve ter o nome do usuário***  cy.**contains**(usuario[3].nome).should('be.visible');  ***// procurar o link de 'minha conta'***  cy.getByData('app-home').find('a').eq(1).click();  cy.url().should('include', '/minha-conta');  ***// ver se o botão de 'salvar alterações' está desabilitado***  cy.getByData('botao-salvar-alteracoes').should('be.disabled');  ***// digitar o nome e senha***  cy.get('[name = "nome"]').type(novosDadosDeUsuario.nome);  cy.get('[name = "senha"]').type(novosDadosDeUsuario.senha);  // ver se o botão de 'salvar alterações' está habilitado  cy.getByData('botao-salvar-alteracoes').should('not.be.disabled');  cy.getByData('botao-salvar-alteracoes').click();  ***// escutar evento de alert***  cy.on('window: alert', (textoDoAlert) => {  expect(textoDoAlert).to.equal('Alterações salvas com sucesso!');  });  ***// ver se redirecionou para a página home***  cy.url().should('include', '/home');  });  ***// window é um comando para manipular o que está na janela do nosso navegador.***  cy.window().then((win) => {  expect(win.localStorage.getItem('nomeUsuario')).to.equal(  novosDadosDeUsuario.nome  );  ***// obter o userId para poder fazer o request e ver se as infos da resposta estão iguais ao novosDadosDeUsuario***  const **userId** = win.localStorage.getItem('userId');  cy.request('GET', `http://localhost:8000/users/${**userId**}`).then(  (resposta) => {  expect(resposta.status).to.eq(200);  expect(resposta.body.nome).to.eq(novosDadosDeUsuario.nome);  expect(resposta.body.senha).to.eq(novosDadosDeUsuario.senha);  }  );  });  });  }); |

-------

**cy.on()**

Em alguns cenários de testes temos que interagir com elementos do DOM, como saber se um botão foi clicado ou se uma caixa de diálogo de um **alert** foi chamada.

O comando **cy.on** é usado para adicionar um ouvinte a um elemento, permitindo que um código seja executado quando um determinado evento ocorrer.

-------

**cy.window()**

Durante a aula precisamos obter uma informação do usuário logado na aplicação que estava salva no navegador, no **localStorage**. Para isso usamos o comando cy.window() para recuperar o nome do usuário o seu id.

O comando cy.window é usado para acessar a janela global do navegador no qual a aplicação está sendo executada. Com este comando, podemos executar operações na janela, como obter ou definir valores de propriedades, navegar entre as páginas ou até mesmo abrir novas guias.

-------

**eventos no Cypress**

O Cypress emite uma série de eventos ao ser executado em um navegador, permitindo que o usuário monitore o comportamento do aplicativo e depure os testes. Esses eventos incluem ações do usuário, como cliques e digitações, bem como eventos de ciclo de vida do aplicativo, como carregamento de página e inicialização de componentes.

Para configurar e utilizar esses eventos nos testes de ponta a ponta, o Cypress fornece o comando **cy.on()**, que permite que o usuário se inscreva em eventos específicos e execute ações personalizadas quando esses eventos ocorrem. Por exemplo, é possível se inscrever no evento de carregamento da página **(cy.on('window:load'))** e verificar se todos os recursos da página foram carregados corretamente.

Além disso, o Cypress também oferece eventos personalizados que podem ser disparados no código do aplicativo, permitindo que o usuário teste o comportamento de acordo com as condições específicas do teste. Por exemplo, é possível disparar um evento personalizado quando um formulário é enviado e se inscrever nesse evento **(cy.on('form:submit'))** para verificar se os dados foram enviados corretamente.

Em resumo, a utilização dos eventos do Cypress pode fornecer uma poderosa ferramenta para depuração e controle do comportamento do aplicativo em testes de ponta a ponta, permitindo que o usuário personalize ações de acordo com as condições específicas do teste.

**Saiba mais:**

<https://docs.cypress.io/api/cypress-api/catalog-of-events>

-------

|  |
| --- |
| describe(**'Realizando transações'**, () => {  const novaTransacao = {  tipoTransacao: 'Depósito',  valor: '100'  }  it(**'Deve permitir que usuário acesse a aplicação, realize transações e faça um logout'**, () => {  cy.fixture('dadosUsuarios').as('usuario');  cy.get('@usuario').then((usuario) => {  **// Realiza login na aplicação**  cy.login(usuario.email, usuario.senha);    **// Visita a página home e verifica a url correspondente**  cy.visit('/home')  cy.url().should('include', '/home');    **// Verifica se o nome do usuário logado é visível**  cy.contains(usuario.nome).should('be.visible');    **// Verifica se a mensagem de Boas vindas está na tela**  cy.getByData('titulo-boas-vindas').should('contain', 'Bem vindo de volta!');  **// Seleciona o campo de select com as opções de transação**  cy.getByData('select-opcoes').select(novaTransacao.tipoTransacao);  **// Verifica se a opção escolhida é a que está selecionada no campo de select**  cy.getByData('select-opcoes').should(  'have.value',  novaTransacao.tipoTransacao  );  **// Preenche o campo de texto com o valor da nova transação**  cy.getByData('form-input').type(novaTransacao.valor);  **// Verifica se o valor no campo de input é o mesmo valor da nova transação**  cy.getByData('form-input').should('have.value', novaTransacao.valor)  **// Clica no botão de realizar transação**  cy.getByData('realiza-transacao').click();  **// Verifica se o valor da nova transação está aparecendo na tela**  cy.getByData('lista-transacoes').find('li').last().contains(novaTransacao.valor)  **// \*\*\*\*\*\*TESTANDO A API\*\*\*\*\*\***  **// Recupera informações do localStorage**  cy.window().then((win) => {  const userId = win.localStorage.getItem('userId');  **// Faz uma requisição para a API no endpoint de transações**  cy.request({  method: 'GET',  url: `http://localhost:8000/users/${userId}/transations`,  failOnStatusCode: false,  }).then((resposta) => {  expect(resposta.status).to.eq(200);  expect(resposta.body).is.not.empty;  expect(resposta.body).to.have.lengthOf.at.least(1);  expect(resposta.body[resposta.body.length - 1]).to.deep.include(  novaTransacao  );  });  });  **// Clica no botão de sair da aplicação**  cy.getByData('botao-sair').click();  **// Verifica a url, se ela corresponde a rota de home**  cy.url().should('include', '/')  **// Faz uma asserção acerta do título da página de início**  cy.getByData('titulo-principal')  .should('contain', 'Experimente mais liberdade no controle da sua vida financeira. Crie sua conta com a gente!')  })  })  }) |

-------

**Dados sensíveis**

São informações da web que não queremos compartilhar com terceiros, como senhas, número do cartão e dados pessoais no geral.

Uma forma de fazermos isso utilizando o Cypress é utilizando as **variáveis de ambiente**.

Criar arquivo **cypress.env.json** na raiz do projeto:

|  |
| --- |
| {  "email": "neilton@alura.com",  "senha": "123456"  } |

|  |
| --- |
| context(**'Realizando login via API'**, ()=> {  it(**'Deve permitir o login do usuário Nelson Seguins'**, () => {  cy.request({  method: 'POST',  url: 'http://localhost:8000/users/login',  body: Cypress.env()  }).then((resposta)=> {  expect(resposta.status).to.eq(200)  expect(resposta.body).is.not.empty  expect(resposta.body.user).to.have.property('nome')  expect(resposta.body.user.nome).to.be.equal('Neilton Seguins')  })  })  }) |

-------

**Aplicação responsiva**

**Criando função utilitária**

Dentro da pasta **cypress/ support**, criar o arquivo ***utils.js****.*

|  |
| --- |
| export const **isMobile** = () => {  return (  Cypress.config(**'viewportWidth'**) < **Cypress.env**("**mobileViewportWidthBreakpoint**")  )  } |

Se o **viewportWidth** for menor que **mobileViewportWidthBreakpoint** então **isMobile**.

No arquivo **cypress.config.js**:

|  |
| --- |
| const { defineConfig } = require("cypress");  module.exports = defineConfig({  e2e: {  baseUrl: 'http://localhost:3000'  },  env: {  **mobileViewportWidthBreakpoint:** 420  },  **viewportWidth: 1200,**  **viewportHeight: 990,**  }); |

Acabamos de definir uma variável de ambiente. Mas, você deve estar se perguntando porque fizemos isso nesse arquivo e não no **cypress.env.json**.

**- cypress.config.js:** as informações que queremos que as pessoas tenham acesso, armazenamos nesse arquivo.

**- cypress.env.json:** os dados sensíveis nesse arquivo.

Em ***pagina.cy.js***:

|  |
| --- |
| import { **isMobile** } from '../support/utils'  describe('Testando múltiplas páginas', () => {  it('Deve conseguir acessar a página de cartões', ()=>{  cy.login(Cypress.env('email'), Cypress.env('senha'));  cy.visit('/home');  cy.location('pathname').should('eq','/home');  if (**isMobile()**) {  *cy.getByData('menu-burguer').should('be.visible');*  *cy.getByData('menu-burguer').click();*  *cy.getByData('menu-lateral').find('a').eq(2).click();*  } else {  *cy.getByData('app-home').find('a').eq(2).click();*  }  cy.getByData('titulo-cartoes').should('exist').and('have.text', 'Meus cartões');  cy.location('pathname').should('eq', '/home/cartoes');  })  }) |

Para mudar o tamanho do Viewport (Mobile) no teste basta digitar no terminal:

***npx cypress open --config viewportWidth=375,viewportHeight=667***

Dessa forma vai entrar na condição do isMobile qdo verdadeiro.

-------

**Usando variáveis de ambiente**

Ao fazer a manutenção nos testes de uma aplicação que você ficou responsável, você se depara com o seguinte código:

|  |
| --- |
| describe('Teste usando variáveis de ambiente', () => {  it('Deve visitar a página correta', () => {  *// CÓDIGO AQUI*  cy.title().should('eq', Cypress.env('TITULO'))  })  }) |

Este teste deve visitar uma página e verificar se o título dela está correta.

Você observou que no arquivo cypress.config.js tem duas variáveis de ambiente, uma chamada URL e a outra de TITULO.

Qual código deve ser inserido no lugar do comentário ***// CÓDIGO AQUI*** para completar o teste?

***cy.visit(Cypress.env('URL'))***

Correta! Pois é a função utilizada para visitar uma página em Cypress e recebe a URL como parâmetro. URL essa que é recuperada do arquivo de configuração que possui salvas as variáveis de ambientes.

-------

**Tamanho da janela**

|  |
| --- |
| describe('Testando viewport', () => {  it('Deve ajustar o viewport para a largura correta', () => {  cy.viewport(1280, 720)  cy.visit('https://www.exemplo.com')  cy.window().then((win) => {  const { innerWidth } = win  expect(innerWidth).to.eq(1280)  })  })  }) |

O comando **cy.viewport** define a largura e a altura do viewport, o comando cy.visit abre o site e, em seguida, o tamanho do viewport é verificado para garantir que foi ajustado corretamente.

-------

**Para saber mais: variáveis de ambiente no Cypress**

As **Variáveis de Ambiente no Cypress** são uma forma de definir valores que podem ser utilizados em nossos testes, tais como **URLs, credenciais de login, chaves de API**, entre outros, sem precisar expor essas informações diretamente no código. Isso é extremamente útil para proteger informações sensíveis, garantindo a segurança do nosso aplicativo.

Para configurar as variáveis de ambiente, podemos utilizar diferentes métodos, como definir variáveis no arquivo **cypress.json**, no arquivo **.env** ou em uma ferramenta de gerenciamento de variáveis de ambiente.

Podemos utilizar essas variáveis em nossos testes por meio do comando **Cypress.env(),** que retorna o valor da variável definida. Por exemplo, podemos utilizar **Cypress.env('URL\_BASE')** para acessar a URL base do nosso aplicativo.

Além disso, as variáveis de ambiente também podem ser utilizadas para customizar nossos testes em diferentes ambientes, tais como ambiente de desenvolvimento, teste ou produção, sem precisar modificar o código.

Exemplo de configuração de variáveis de ambiente no arquivo **cypress.json**:

|  |
| --- |
| {  "baseUrl": "https://myapp.com",  "env": {  "USERNAME": "user1",  "PASSWORD": "password1"  }  } |

Exemplo de utilização das variáveis de ambiente em nossos testes:

|  |
| --- |
| describe('Login', () => {  beforeEach(() => {  cy.visit('/')  })  it('should login with valid credentials', () => {  cy.get('#username').type(Cypress.env('USERNAME'))  cy.get('#password').type(Cypress.env('PASSWORD'))  cy.get('#login-button').click()  cy.url().should('include', '/dashboard')  })}) |

-------

**Flaky tests**

**retry()**

Flaky tests, ou testes instáveis, são testes automatizados que apresentam resultados inconsistentes, ou seja, que passam em alguns momentos e falham em outros sem nenhuma mudança significativa no código. Isso pode ser causado por vários fatores, como problemas de sincronização com a interface do usuário, tempo de espera insuficiente, alterações no ambiente de execução ou em dependências externas.

Para minimizar a ocorrência de testes instáveis, é importante adotar boas práticas de programação e de teste, como evitar a dependência excessiva da ordem de execução, definir um tempo de espera adequado e manter o ambiente de teste consistente.

Para lidar com testes instáveis no Cypress, existem algumas abordagens possíveis. Uma delas é a utilização de comandos específicos do Cypress, como o .**retry()**.

O **.retry()** permite que o teste seja executado novamente caso ele falhe, até um número máximo de vezes.

Outra abordagem para lidar com testes instáveis é a reexecução automática de testes falhos. Se quiser saber mais sobre isso pode consultar a documentação neste link.

<https://docs.cypress.io/guides/guides/test-retries>

-------