

Trilha 05

Automação de Testes

Instruções para a melhor prática de Estudo

1. **Leia atentamente todo o conteúdo:** Antes de iniciar qualquer atividade, faça uma leitura detalhada do material fornecido na trilha, compreendendo os conceitos e os exemplos apresentados.
2. **Não se limite ao material da trilha:** Utilize o material da trilha como base, mas busque outros materiais de apoio, como livros, artigos acadêmicos, vídeos, e blogs especializados. Isso enriquecerá o entendimento sobre o tema.
3. **Explore a literatura:** Consulte livros e publicações reconhecidas na área, buscando expandir seu conhecimento além do que foi apresentado. A literatura acadêmica oferece uma base sólida para a compreensão de temas complexos.
4. **Realize todas as atividades propostas:** Conclua cada uma das atividades práticas e teóricas, garantindo que você esteja aplicando o conhecimento adquirido de maneira ativa.
5. **Evite o uso de Inteligência Artificial para resolução de atividades:** Utilize suas próprias habilidades e conhecimentos para resolver os exercícios. O aprendizado vem do esforço e da prática.
6. **Participe de debates:** Discuta os conteúdos estudados com professores, colegas e profissionais da área. O debate enriquece o entendimento e permite a troca de diferentes pontos de vista.
7. **Pratique regularmente:** Não deixe as atividades para a última hora. Pratique diariamente e revise o conteúdo com frequência para consolidar o aprendizado.
8. **Peça feedback:** Solicite o retorno dos professores sobre suas atividades e participe de discussões sobre os erros e acertos, utilizando o feedback para aprimorar suas habilidades.

Essas instruções são fundamentais para garantir um aprendizado profundo e eficaz ao longo das trilhas.

1. Cypress

O Cypress é uma ferramenta de automação de testes end-to-end (E2E) voltada principalmente para o teste de aplicações web. Ela permite que desenvolvedores e testadores escrevam e executem testes de forma rápida, fácil e confiável, simulando a interação de um usuário com a aplicação em um navegador.

1.1. Teste Web

Exemplo: Teste criado com o objetivo de validar os elementos na tela de login:

```
it('Meu primeiro teste.', () => {  
  cy.visit('https://www.saucedemo.com');  
  
  cy.get('.login_logo')  
    .then((element) => {  
      expect(element.text()).eq('Swag Labs');  
      expect(element).to.be.visible;  
      expect(element).not.disabled;  
    });  
});
```

1.2. Entendendo o código

Os **asserts** (ou asserções) são verificações usadas para garantir que um determinado valor, elemento ou comportamento esteja correto durante a execução de um teste. No Cypress, as asserções são usadas para validar que o resultado esperado corresponde ao resultado real, garantindo que a aplicação esteja funcionando como esperado.

- Acessa a página **<https://www.saucedemo.com>**.
- Verifica se o elemento com a classe **.login_logo** está presente.
- Confirma que o texto do elemento é **"Swag Labs"**.
- Verifica se o elemento está visível.
- Garante que o elemento não está desabilitado.

1.3. Rodando o Teste

- **Via interface:** Use o comando **npx cypress open** e selecione o teste.
- **Via terminal:** (modo headless): **npx cypress run**

1.4. Resultado do Teste

Exemplo rodando o teste pelo terminal, espere um resultado parecido como este, significa que os testes foram executados com êxito:

(Run Finished)

| Spec | Tests | Passing | Failing | Pending | Skipped |
|---------------------|-------|---------|---------|---------|---------|
| ✓ spec.cy.js | 00:03 | 1 | 1 | - | - |
| ✓ All specs passed! | 00:03 | 1 | 1 | - | - |

1.5. Asserções Implícitas

As asserções implícitas são usadas diretamente nos comandos do Cypress, como `cy.get()`, e verificam automaticamente a condição.

```
describe('Asserções Implícitas. ', () => {
  // Verifica se um elemento está visível
  cy.get('.login-logo').should('be.visible');

  // Verifica se um elemento contém um texto específico
  cy.get('.welcome-message').should('contain', 'Bem-vindo');

  // Verifica se um campo de entrada tem um valor específico
  cy.get('#email').should('have.value', 'teste@email.com');

  // Verifica se um elemento tem uma classe CSS específica
  cy.get('.botao').should('have.class', 'ativo');

  // Verifica se um elemento NÃO contém uma classe
  cy.get('.botao').should('not.have.class', 'desativado');

  // Verifica se um botão está desabilitado
  cy.get('#submit-button').should('be.disabled');

  // Verifica se um elemento NÃO está desabilitado
  cy.get('#submit-button').should('not.be.disabled');
});
```

1.6. Asserções Explícitas

As asserções explícitas são usadas quando precisamos manipular o primeiro elemento antes de validar algo, geralmente dentro de um `.then()`.

```
describe('Asserções Explícitas. ', () => {
  cy.get('.titulo').then((element) => {
    // Verifica se o texto é exatamente igual a "Página Inicial"
    expect(element.text()).to.eq('Página Inicial');

    // Verifica se o elemento está visível
    expect(element).to.be.visible;

    // Verifica se um elemento NÃO tem uma classe específica
    expect(element).not.to.have.class('erro');
  });
});
```

1.7. Testes com API

O Cypress não se limita apenas à automação de testes com manipulação de elementos web, ele também permite a realização de chamadas de APIs diretamente, possibilitando a criação de testes mais robustos e completos. Com essa funcionalidade, é possível validar endpoints REST, enviar requisições GET, POST, PUT e DELETE, e até mesmo manipular tokens de autenticação e dados dinâmicos. Isso torna o Cypress uma ferramenta poderosa para testes end-to-end, permitindo a combinação de testes na interface gráfica com interações diretas na API, garantindo que fluxos complexos sejam testados de forma eficaz e integrada.

Exemplo de uma requisição para um API Fake:

```
describe.only('API - Apenas Testes', () => {
  Add only
  it('Verifica o status da requisição.', () => {
    cy.api({
      method: 'GET',
      url: 'https://jsonplaceholder.typicode.com/posts'
    }).then((response)=>{
      expect(response.status).to.equal(200); // Verifica se a resposta foi 200 OK
    })
  });
});
```

1.8. Preparação de Ambiente

O Cypress é uma ferramenta de automação de testes para aplicações web. Para configurar o ambiente e começar a utilizá-lo, siga os passos abaixo:

1.9. Pré-requisitos

Node.js (versão LTS recomendada) → [Download Node.js](#)

npm (já vem com o Node.js)

Configurações

Nesse momento, devemos ter nossa pasta do projeto já criada, nosso exemplo será **teste-cypress**, através de um editor de texto abra a pasta do projeto, em seguida execute o seguinte comando: **npm init -y**, essa configuração é necessária para criar a estrutura inicial no arquivo **package.json**, ao final rode o comando abaixo:

```
{
  "devDependencies": {
    "cypress": "^13.17.0",
    "cypress-mochawesome-reporter": "^3.8.2",
    "cypress-multi-reporters": "^2.0.4",
    "cypress-plugin-api": "^2.11.2"
  },
  "name": "teste-cypress",
  "version": "1.0.0",
  "main": "cypress.config.js",
  "scripts": { },
  "keywords": [ ],
  "author": "Emerson Amancio",
  "license": "ISC",
  "description": "Projeto para Testes"
}
```

Coloque todas as instruções listadas no arquivo, isso irá instalar todas as dependências que iremos utilizar ao decorrer do projeto, dentro do projeto rode o comando: **npm install**.

1.2.1 Instalar o Cypress

Dentro do seu projeto, execute: **npm install cypress --save-dev** caso você ainda não o tenha.


1.2.2. Abrindo o Cypress

No seu projeto abra o terminal e execute o seguinte comando: **npx cypress open**, esse comando irá executar em background o Cypress e por fim o mesmo vai abrir a dashboard inicial.

Aqui devemos criar as configurações da estrutura no nosso projeto, clique em **E2E Testing**:

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.

● **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**

No próximo passo devemos clicar em prosseguir para finalizar a configuração dos arquivos, segue o exemplo abaixo:

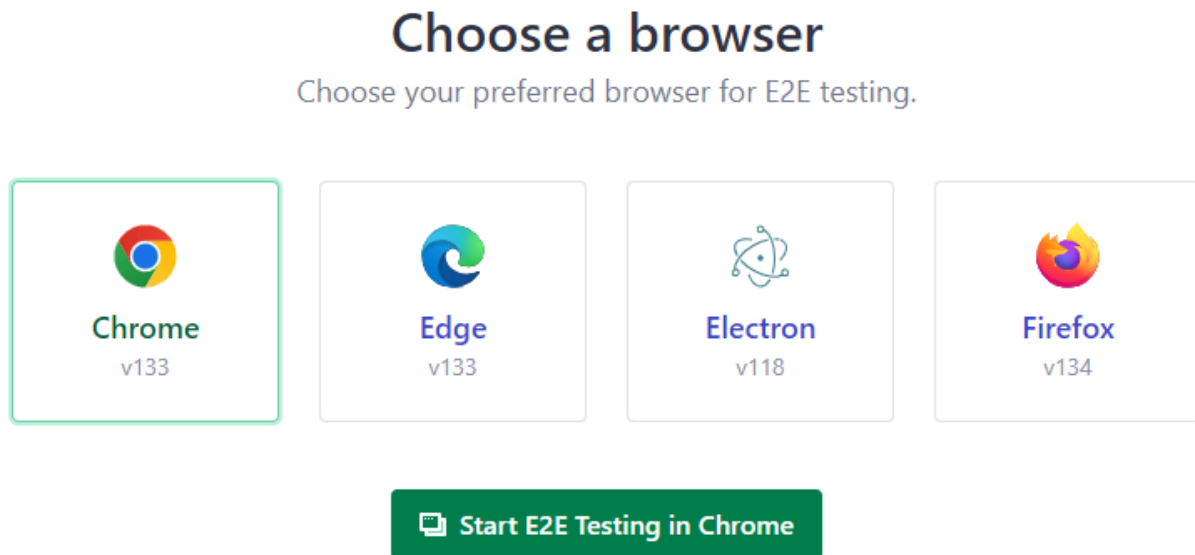
Configuration files

We added the following files to your project:

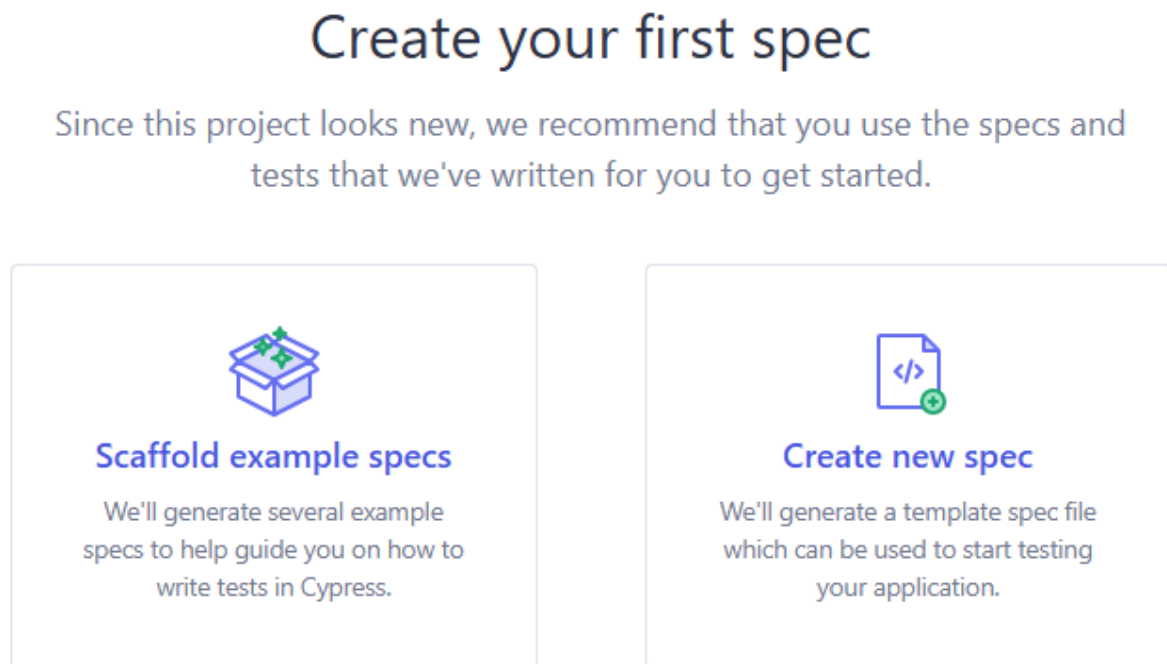
| | | |
|---|---|---|
| ✓ | cypress.config.js The Cypress config file for E2E testing. | ▼ |
| ✓ | cypress\support\e2e.js The support file that is bundled and loaded before each E2E spec. | ▼ |
| ✓ | cypress\support\commands.js A support file that is useful for creating custom Cypress commands and overwriting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

Continue

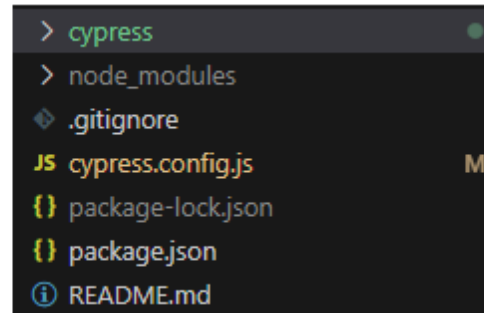
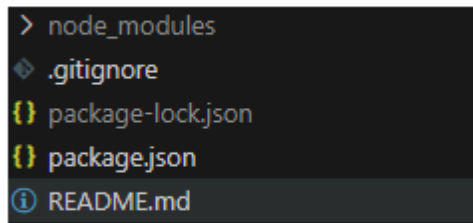
Ao final irá aparecer a opção de selecionar o navegador que desejamos executar os testes:



No próximo passo irá dar duas opções, **Scaffold example specs** e **Create new spec**, a primeira opção criará vários arquivos de exemplos de testes, a segunda irá dar a opção de criarmos o teste.



Abaixo o exemplo final de como ficou nossa estrutura de pastas **antes** e **depois** das configurações:



1.2.3. Estrutura Cypress

cypress/e2e/: Essa pasta contém os testes automatizados.

- **spec.cy.js**: Arquivo que contém os testes E2E escritos em Cypress.

Normalmente, cada arquivo representa um conjunto de testes para uma funcionalidade específica da aplicação.

cypress/support/: Contém arquivos auxiliares para suporte aos testes.

- **commands.js**: Arquivo onde podemos definir comandos personalizados do Cypress.

e2e.js: Arquivo de configuração global para os testes E2E.

- Esse arquivo pode conter hooks, como **beforeEach()**, para executar ações antes de cada teste.

cypress.config.js: Arquivo de configuração principal do Cypress.

- Define configurações globais, como tempo limite de execução, ambiente **baseUrl**, e comportamentos personalizados.

1.2.4. Criando um Teste

Crie um arquivo dentro de **cypress/e2e/**, por exemplo: **meu-teste.cy.js**:

```
it('Meu primeiro teste.', () => {
  cy.visit('https://www.saucedemo.com');

  cy.get('.login_logo')
    .then((element) => {
      expect(element.text()).eq('Swag Labs');
      expect(element).to.be.visible;
      expect(element).not.disabled;
    });
});
```

1.2.5. Entendendo o código

Os **asserts** (ou asserções) são verificações usadas para garantir que um determinado valor, elemento ou comportamento esteja correto durante a execução de um teste. No Cypress, as asserções são usadas para validar que o resultado esperado corresponde ao resultado real, garantindo que a aplicação esteja funcionando como esperado.

- Acessa a página **<https://www.saucedemo.com>**.
- Verifica se o elemento com a classe **.login_logo** está presente.
- Confirma que o texto do elemento é **"Swag Labs"**.
- Verifica se o elemento está visível.
- Garante que o elemento não está desabilitado.

1.2.6. Rodando o Teste

- **Via interface:** Use o comando **npx cypress open** e selecione o teste.
- **Via terminal:** (modo headless): **npx cypress run**

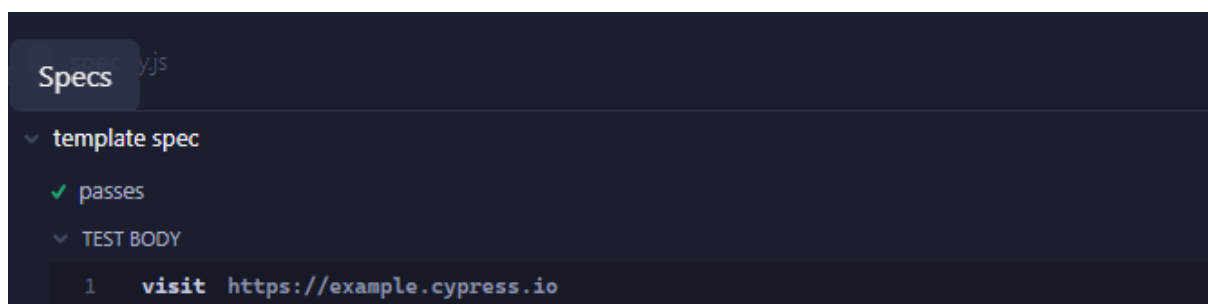
1.2.7. Resultado do Teste

Exemplo rodando o teste pelo terminal, espere um resultado parecido como este, significa que os testes foram executados com êxito:

(Run Finished)

| Spec | Tests | Passing | Failing | Pending | Skipped |
|---------------------|-------|---------|---------|---------|---------|
| ✓ spec.cy.js | 00:03 | 1 | 1 | - | - |
| ✓ All specs passed! | 00:03 | 1 | 1 | - | - |

Exemplo rodando pela interface através do comando **npx cypress open**:



1.2.8. Configuração Adicional

Personalize o arquivo **cypress.config.js** para ajustar as configurações do ambiente de teste, como viewport, timeouts, ou baseUrl:

Configurações de variáveis de ambiente:

- Abaixo configuramos as variáveis bases para conexão com nossos sites ou APIs, também configuramos as demais variáveis para a conexão com o nosso banco de dados (MongoDB):

```
env: {
  mongodb: {
    uri: process.env.MONGODB_URI,
    database: 'test',
    collection: 'carros'
  },
  urlSite: 'https://www.saucedemo.com',
  urlApiWeb: 'https://serverest.dev',
  urlApiLocal: 'http://localhost:3000',
  urlApiMongoDB: 'http://localhost:3000'
}
```

1.2.9. Extensões para o Visual Studio Code

Nas opções de extensão pesquise pelas seguintes extensões abaixo para facilitar a forma que escrevemos os testes:

- Cypress Snippets.
- Add Only.

1.3.1. Gerenciamento de Dados

Os **fixtures** no Cypress são arquivos utilizados para armazenar dados externos (como JSON, CSV ou outros formatos) que podem ser reutilizados em vários testes. O objetivo principal de usar fixtures é separar os dados do código dos testes, tornando os testes mais organizados, reutilizáveis e fáceis de manter.

Criar o Arquivo de Dados (Fixtures)

Agora, dentro da pasta cypress/fixtures/, crie um arquivo chamado **login.json** e adicione os seguintes dados:

cypress/fixtures/login.json:

```
{
  "user": {
    "loginCorreto": {
      "username": "Admin",
      "password": "admin123"
    },
    "loginIncorreto": {
      "username": "user",
      "password": "123456"
    }
  }
}
```

Agora, dentro da pasta **cypress/e2e/**, crie um arquivo chamado **login.cy.js**:

```
it('Testa o comportamento do botão de login (clicável)', () => {
  cy.fixture('userData.json').then((userData) => {
    // Verifica se o botão pode ser clicado sem lançar erro
    cy.get('[name="username"]').type(userData.user.loginCorreto.username);
    cy.get('[name="password"]').type(userData.user.loginCorreto.password);

    cy.get('.oxd-button')
      .should('not.be.disabled') // Verifica se o botão não está desabilitado
      .click();

    // Se as credenciais forem válidas, esperamos que a página de inventário seja carregada
    cy.url().should('include', '/dashboard');
  });
});
```

Essa é uma ótima opção para manipular dados pré-definidos em nossos testes, assim não poluindo a lógica dos testes com informações como os dados que precisamos para testes.

1.3.2. Gerando Relatórios

O Cypress não gera relatórios detalhados nativamente, mas podemos usar bibliotecas como **Mochawesome** para criar relatórios visuais e estruturados dos testes.

Antes de mais nada, configura se as seguintes instruções estão presentes em seu arquivo **package.json**:

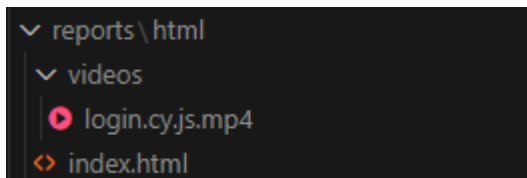
```
"cypress-mochawesome-reporter": "^3.8.2",
"cypress-multi-reporters": "^2.0.4",
```

Agora no arquivo **cypress.config.js** será necessário implementar as seguintes configurações:

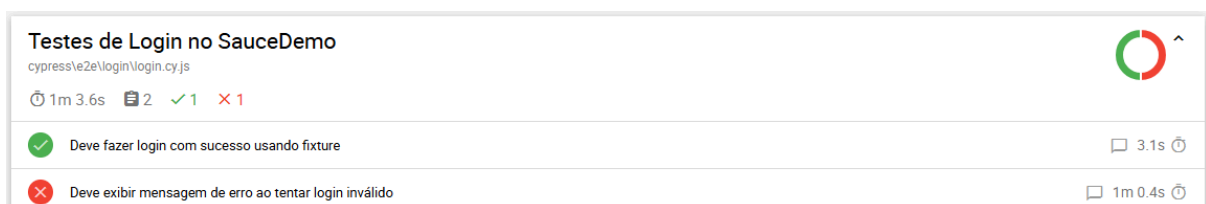
```
const { defineConfig } = require("cypress");
const { configurePlugin } = require('cypress-mongodb');

module.exports = defineConfig({
  reporter: 'cypress-multi-reporters',
  video: true,
  reporterOptions: {
    reporterEnabled: 'cypress-mochawesome-reporter',
    cypressMochawesomeReporterReporterOptions: {
      charts: true,
      reportPageTitle: 'Relatório de Testes',
      embeddedScreenshots: true,
      inlineAssets: true,
      saveAllAttempts: false
    }
  },
  e2e: {
    setupNodeEvents(on, config) {
      require('cypress-mochawesome-reporter/plugin')(on);
      configurePlugin(on);
    },
  },
});
```

Agora basta executar os testes com o comando: **npx cypress run** no seu terminal e esperar o resultado. Ao final dos testes uma pasta chamada **reports** será criada contendo todas as evidências que foram tiradas dos testes, abaixo o exemplo gerado:



Abrindo o arquivo **index.html** será exibido tudo o que foi testado, segue o exemplo abaixo:



1.3.3. Testes de API

O Cypress não se limita apenas à automação de testes com manipulação de elementos web, ele também permite a realização de chamadas de APIs diretamente, possibilitando a criação de testes mais robustos e completos. Com essa funcionalidade, é possível validar endpoints REST, enviar requisições GET, POST, PUT e DELETE, e até mesmo manipular tokens de autenticação e dados dinâmicos. Isso torna o Cypress uma ferramenta poderosa para testes end-to-end, permitindo a combinação de testes na interface gráfica com

interações diretas na API, garantindo que fluxos complexos sejam testados de forma eficaz e integrada.

Exemplo de uma requisição para um API Fake, crie o arquivo de teste **api.cy.js**:

```
describe.only('API - Apenas Testes', () => {
  // Add only
  it('Verifica o status da requisição.', () => {
    cy.api({
      method: 'GET',
      url: 'https://jsonplaceholder.typicode.com/posts'
    }).then((response) => {
      expect(response.status).to.equal(200); // Verifica se a resposta foi 200 OK
    })
  });
});
```

Explicação do Código

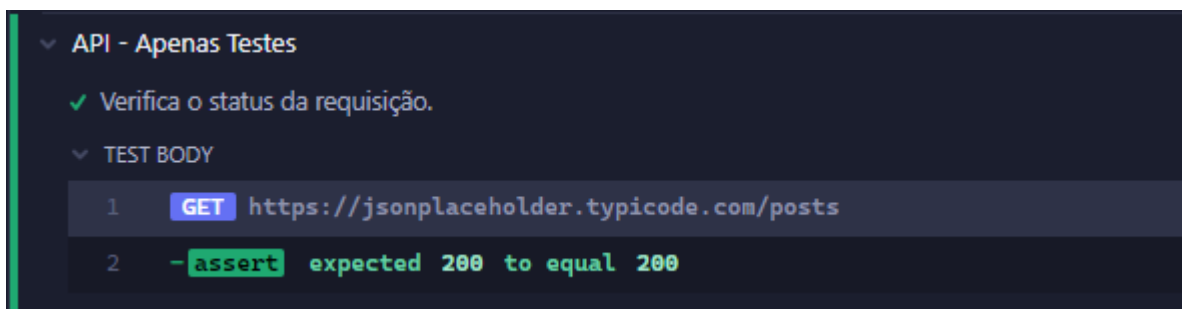
cy.request({ method: 'GET', url: 'https://jsonplaceholder.typicode.com/posts' })

Faz uma requisição GET para a API Fake do **JSONPlaceholder**.

- Verificação do status da resposta (**200 OK**)

Resultado Requisição API

Resultado das linhas de teste na interface do Cypress:



O teste finalizado com sucesso, também validou todas as asserções que pré definimos.

1.3.4. Sessions

A função **cy.session()** no Cypress é essencial para otimizar testes que exigem autenticação, como login em uma API ou sistema web. Ela armazena e reutiliza a sessão do usuário, evitando que cada teste tenha que realizar login do zero.

Primeiro devemos colocar o seguinte trecho de código no arquivo **commands.js**:

```
Cypress.Commands.add("login", (user, pass) => {
  cy.session([user, pass], () => {
    cy.visit(Cypress.env('urlSite'));

    cy.get('.login_logo')
      .then((element) => {
        expect(element.text()).eq('Swag Labs');
        expect(element).to.be.visible;
        expect(element).not.disabled;
      });

    cy.get('[data-test="username"]').type(user);
    cy.get('[data-test="password"]').type(pass);

    cy.get('[data-test="login-button"]').click();
  }, {
    cacheAcrossSpecs: true
  });
});
```

O que esse código faz?

- Visita a página inicial (usando **cy.visit(Cypress.env('urlSite'))**).
- Verifica a presença e visibilidade do logotipo (classe **.login_logo**) para garantir que seja **"Swag Labs"**, que esteja visível e não esteja desabilitado.
- Preenche os campos de usuário e senha (localizados pelos atributos **data-test="username"** e **data-test="password"**).
- Clica no botão de login (**[data-test="login-button"]**).
- Armazena a sessão (por meio de **cy.session(...)**) para que não seja necessário repetir o login a cada teste, graças à opção **cacheAcrossSpecs: true**.

Aqui temos nosso teste, percebe-se que utilizamos do arquivo **commands** para reaproveitar nossa função de login:

```
describe('Testes de Login com API e Sessão', () => {
  beforeEach(() => {
    cy.login("standard_user", "secret_sauce");
  });

  Add only
  it('Deve validar que o login foi realizado com sucesso', () => {
    cy.url().should('include', '/inventory.html'); // Confirma que está logado
    cy.get('.title').should('contain', 'Products'); // Verifica a presença do título da página de produtos
  });

  Add only
  it('Deve manter a sessão entre os testes e acessar o carrinho', () => {
    cy.get('.shopping_cart_link').click(); // Clica no ícone do carrinho
    cy.url().should('include', '/cart.html'); // Confirma que está na página do carrinho
  });
});
```

Imagine que precisamos testar um sistema em que a primeira ação é o login. Após essa autenticação, desejamos verificar dez telas diferentes do sistema. Utilizando o fluxo de sessão do Cypress, é possível executar os testes de todas as telas sem precisar refazer o login para cada uma delas. Sem essa opção, quando um teste terminasse em uma tela e passássemos para a seguinte, o contexto de usuário logado seria perdido. Dessa forma, o teste seria interrompido por não reconhecer que ainda estamos autenticados no sistema.

1.3.5. Commands

Os commands personalizados no Cypress servem para evitar repetição de código, organizar melhor os testes e facilitar a manutenção. Eles permitem encapsular ações comuns dentro de um único comando, tornando os testes mais limpos e reutilizáveis.

No Cypress, já existe um arquivo chamado **cypress/support/commands.js**. Se ele não existir, crie manualmente dentro da pasta support.

Local do arquivo: **cypress/support/commands.js**

```
Cypress.Commands.add('login', (username, password) => {
  cy.visit('https://www.saucedemo.com/'); // Acessa a página

  cy.get('[data-test="username"]').type(username); // Insere usuário
  cy.get('[data-test="password"]').type(password); // Insere senha
  cy.get('[data-test="login-button"]').click(); // Clica no botão de login
});
```

Agora no arquivo de testes colocamos as seguintes instruções:

```
describe('Testes de Login', () => {
  beforeEach(() => {
    cy.fixture('usuarios.json').then((dados) => {
      cy.visit('https://www.saucedemo.com/');
      cy.login(dados.usuarioValido.username, dados.usuarioValido.password);
    });
  });

  Add only
  it('Deve validar que o login foi realizado com sucesso', () => {
    cy.url().should('include', '/inventory.html'); // Confirma que está logado
  });

  Add only
  it('Deve exibir erro para login inválido', () => {
    cy.fixture('usuarios.json').then((dados) => {
      cy.login(dados.usuarioInvalido.username, dados.usuarioInvalido.password);
      cy.get('[data-test="error"]').should('be.visible');
    });
  });
});
```


Em resumo, o código faz:

Carrega dados de usuários a partir de um arquivo JSON (por exemplo, **usuarioValido** e **usuarioInvalido**).

Efetua login no site (usando um comando Cypress customizado **cy.login(...)**).

Valida dois cenários: login de sucesso (checando a URL) e login inválido (checando uma mensagem de erro).

Nessa forma conseguimos reutilizar toda a parte de acesso e login que antes era feito com a duplicidade da lógica. Com isso, apenas realizamos a chamada da função enviando apenas os dados que necessita ser validado.

1.3.6. Page Objects

O Page Object Model (**POM**) é um padrão de design utilizado em testes automatizados para separar a lógica dos testes da interação com os elementos da página. Em vez de escrever seletores diretamente nos testes, criamos uma classe separada que representa a página e seus elementos.

Com POM, cada página da aplicação tem uma classe que encapsula seus elementos e ações, tornando os testes mais organizados, reutilizáveis e fáceis de manter.

Antes de implementar o código vamos definir nossa estrutura de pastas conforme abaixo:

Estrutura do Projeto

```

cypress/
├── e2e/ (onde ficam os testes)
│   └── login.cy.js (arquivo de teste)
├── support/
│   └── pages/ (pasta para armazenar as classes das páginas)
│       └── LoginPage.js (classe para a página de login)

```

No nosso arquivo **LoginPage.js** vamos implementar as seguintes instruções:

```

class Login {
  visitarPagina() { cy.visit(Cypress.env('urlSite')); }

  getUserField() { return cy.get('[data-test="username"]'); }
  getPasswordField() { return cy.get('[data-test="password"]'); }
  getLoginButton() { return cy.get('[data-test="login-button"]'); }

  fillUser(user) { this.getUserField().type(user); }
  fillPassword(password) { this.getPasswordField().type(password); }

  clickLogin() { this.getLoginButton().click(); }

  // Função que combina as ações para realizar o login
  login(username, password) {
    this.fillUser(username);
    cy.log("Nome: "+username);
    this.fillPassword(password);
    cy.log("Senha: "+password);

    this.clickLogin();
  }
}

export default new Login()

```

Explicando o código criado acima:

Função para acesso a página que especificamos no arquivo de configuração do Cypress:

```
visitarPagina() { cy.visit(Cypress.env('urlSite')); }
```

Funções para capturar os campos da nossa interface, neste caso são os campos para input de dados e o botão de login:

```

getUserField() { return cy.get('[data-test="username"]'); }
getPasswordField() { return cy.get('[data-test="password"]'); }
getLoginButton() { return cy.get('[data-test="login-button"]'); }

```

Aqui criamos mais duas funções, ambas fazem as instâncias das funções que criamos anteriormente e com isso enviamos por parâmetro os dados que precisamos que sejam imputados em cada campo:

```

fillUser(user) { this.getUserField().type(user); }
fillPassword(password) { this.getPasswordField().type(password); }

```

Já abaixo criamos a função de click do botão, e a função de login, onde essa função faz as tratativas de cada campo:

```
clickLogin() { this.getLoginButton().click(); }

// Função que combina as ações para realizar o login
login(username, password) {
  this.fillUser(username);
  cy.log("Nome: "+username);
  this.fillPassword(password);
  cy.log("Senha: "+password);

  this.clickLogin();
}
```

Ao final realizamos a exportação da nossa classe de Login, com isso podemos utilizá-la nos arquivos que serão necessário essas ações, neste caso no arquivo de teste do Login:

```
export default new Login()
```

Nosso arquivo de teste **login.cy.js** irá ficar assim:

```
import Login from '../pages/login'

Add only
describe('Testes de Login!', () => {
  beforeEach(() => {
    Login.visitarPagina();
  });

  Add only
  it('Deve realizar o login com sucesso.', () => {
    Login.login("standard_user", "secret_sauce");
  });
});
```

Diferença Page Object x Commands

Tanto o Page Object Model (POM) quanto o uso de Commands (commands.js) são formas de organizar e reutilizar código nos testes do Cypress. No entanto, eles têm propósitos diferentes e são usados de maneiras distintas.

1.3.7. Cypress Cloud

O **Cypress Cloud** (antes conhecido como Cypress Dashboard) é um serviço online que se integra ao framework de testes Cypress para:

- **Armazenar e exibir** relatórios detalhados das execuções de teste (logs, capturas de tela, vídeos, falhas etc.).
- Permitir **acompanhamento** histórico dos testes, com gráficos e registros de quais commits/branches passaram ou falharam.
- **Facilitar** a análise, depuração e colaboração entre os membros da equipe (compartilhando links do dashboard).

Quando você executa seus testes com a flag `--record` e informa a Record Key do seu projeto, o Cypress envia dados dessas execuções para o Cloud, onde você pode visualizá-los num painel web.

Conta e Configurações Cloud

No portal do [Cypress Cloud](#) você deverá criar uma conta e criar um projeto, após feito este processo vá em **Project Settings > General**, terá todas as informações para capturar o **Project ID** e **Record Keys**, essas informações serão necessárias para configurações adiante.

Configuração Projeto

Primeiro será necessário configurar o arquivo `.env` de nosso projeto, crie esse arquivo na pasta raiz do seu projeto e configure com o seguinte escopo mostrado abaixo:

```
CYPRESS_RECORD_KEY=0000-0000-0000-0000-000000000000
CYPRESS_PROJECT_ID=000000
```

Em seguida iremos configurar nosso arquivo `cypress.config.js` para identificar nossa variável do projeto:

```
module.exports = defineConfig({
  projectId: process.env.CYPRESS_PROJECT_ID,
  video: true,
```

Agora devemos certificar de que o **dotenv** está sendo requisitado no arquivo:

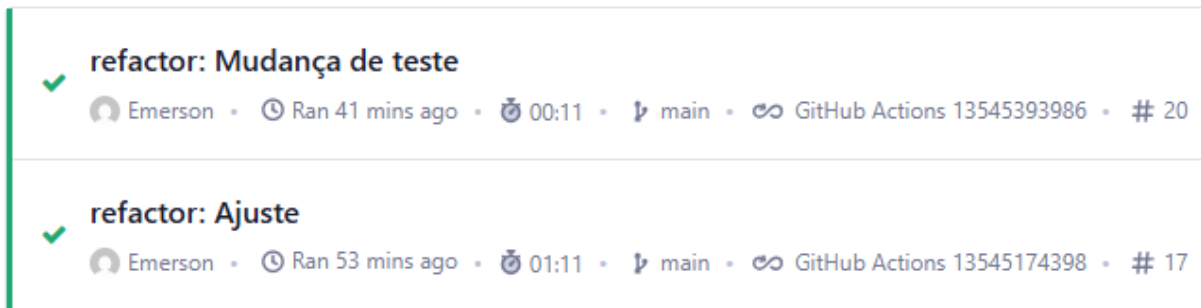
```
require('dotenv').config();
```

Rodando o teste com o comando abaixo:

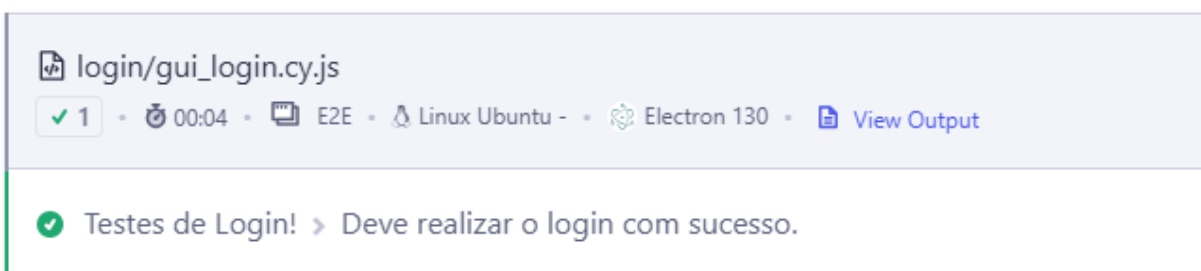
```
npx cypress run --browser chrome --record --key 0000-0000-0000-0000-000000000000
```

Resultados Gerados

Na aba **Latest runs** temos todos os testes que foram executados, você verá algo como:



Selecionando um dos testes que foi executado e a aba **Test Results** teremos todo o fluxo dos testes:



No exemplo listado acima percebemos que houve sucesso no cenário executado, é exibido algumas informações simples como, **tempo de execução**, **total de testes**, **máquina**, etc.

O Cypress possui uma interface bem robusta e intuitiva de se usar, neste mesmo exemplo temos a opção de visualizar o Log que foi executado na própria máquina ou até mesmo ver o teste em forma de vídeo que foi executado. Em resumo ele possui as duas opções principais que seriam **execução via terminal** e **execução via interface**.

2. Jest

Jest é um framework de testes em JavaScript criado pelo Facebook (atualmente, Meta) que roda em Node.js. Ele é amplamente utilizado para testar aplicações front-end (especialmente as que usam React) e também pode ser aplicado no back-end. Ele é mantido pelo Facebook e se destaca por ser rápido, simples e poderoso.

Qual o objetivo do uso do Jest?

- **Automatizar testes:** Garante que o código continua funcionando corretamente após mudanças.

- **Verificar a lógica do código:** Ajuda a testar funções, cálculos e fluxos da aplicação.
- **Testar componentes de frontend e backend:** Compatível com APIs, interfaces e até testes assíncronos.
- **Executar testes rapidamente:** Possui um mecanismo interno de paralelismo que agiliza os testes.

Exemplos de Testes

Criando nossa função para fins de teste, crie um arquivo chamado **math.js** dentro da pasta **src**:

```
function soma(a, b) {
  if(typeof a !== 'number' || typeof b !== 'number') {
    throw new Error("Os parâmetros devem ser números");
  }

  return a + b;
}
```

Agora crie o arquivo de teste dentro da pasta **tests** chamado **math.test.js**:

```
// tests/math.test.js
const { soma } = require('../src/math');

Add only
describe('Função de soma', () => {
  Add only
  it('Deve somar dois números corretamente', () => {
    expect(soma(2, 3)).toBe(5);
  });
  Add only
  it('Deve lançar erro se um dos parâmetros não for número', () => {
    expect(() => soma(2, 'a')).toThrow("Os parâmetros devem ser números");
  });
});
```

O que acontece aqui?

Teste 1

- Chama a função **soma(2, 3)** e verifica se o retorno é 5.
- O método **expect().toBe()** compara o valor retornado com o esperado.
- Se a função **soma(2, 3)** retornar 5, o teste passa.
- Se retornar outro valor, o teste falha.

Teste 2

- Verifica se a função **soma()** lança um erro ao receber um valor inválido.
- O método **expect().toThrow()** espera que a função gere um erro com a mensagem "Os parâmetros devem ser números".
- Se a função **soma(2, 'a')** lançar um erro com essa mensagem, o teste passa.
- Se não lançar erro ou lançar uma mensagem diferente, o teste falha.

2.1. Instalação do Jest

Para instalado, execute este comando no seu projeto: **npm install --save-dev jest**.

Para rodar os testes, coloque as seguintes instruções no seu arquivo **package.json**:

```
"scripts": {  
  "test": "jest"  
},
```

Agora para executar os testes execute o seguinte comando no terminal: **npm test**.

2.2. Gerando Relatórios

O Jest permite gerar relatórios detalhados sobre os testes executados. Esses relatórios são úteis para monitorar a qualidade do código, identificar falhas e compartilhar resultados com a equipe.

Instalar a Dependência

```
npm install --save-dev jest-html-reporter
```

Crie um arquivo chamado **jest.config.js** na raiz do seu projeto e adicione a seguinte configuração:

```
module.exports = {  
  reporters: [  
    'default',  
    [  
      'jest-html-reporter',  
      {  
        pageTitle: 'Relatório de Testes',  
        outputPath: './relatorio.html',  
        includeFailureMsg: true  
      }  
    ],  
  ],  
};
```

O que essa configuração faz?

O Jest continua usando seu relatório padrão, mas agora também gera um relatório HTML no arquivo **relatorio.html**. Se um teste falhar, a mensagem de erro será incluída no relatório.

Exemplo de API

Agora vamos testar a API pública JSONPlaceholder usando Jest + SuperTest.

- Sem precisar criar um servidor
- Testando requisições reais para <https://jsonplaceholder.typicode.com>

Criando nosso arquivo chamado **jsonplaceholder.test.js**:

```
const request = require('supertest');

const API_URL = 'https://jsonplaceholder.typicode.com';

Add only
describe('Testes na API JSONPlaceholder', () => {
  // 1 - Testando uma requisição GET
  Add only
  test('Deve buscar um post pelo ID', async () => {
    const response = await request(API_URL).get('/posts/1');

    expect(response.status).toBe(200); // Verifica se a resposta foi bem-sucedida
    expect(response.body).toHaveProperty('id', 1); // Garante que o post tem o ID correto
    expect(response.body).toHaveProperty('title'); // Verifica se há um título
    expect(response.body).toHaveProperty('body'); // Verifica se há um corpo
  });
});
```

O que esse teste faz?

- Testa uma API real (JSONPlaceholder) sem precisar criar um servidor
- Faz requisições GET,
- Verifica se as respostas contêm os dados esperados
- Garante que os status HTTP estão corretos (200)

Continuação para o método **POST**:


```
// 2 - Testando uma requisição POST
Add only
test('Deve criar um novo post', async () => {
  const newPost = {
    title: 'Post de Teste',
    body: 'Este é um post criado via teste automatizado',
    userId: 1,
  }

  const response = await request(API_URL)
    .post('/posts')
    .send(newPost)
    .set('Content-Type', 'application/json');

  expect(response.status).toBe(201); // Verifica se o post foi criado com sucesso
  expect(response.body).toMatchObject(newPost); // Confirma que os dados enviados são os mesmos retornados
});
```

O que esse teste faz?

- Testa uma API real (JSONPlaceholder) sem precisar criar um servidor
- Faz requisições POST,
- Verifica se o post foi criado com sucesso
- Garante que os status HTTP estão corretos (201)

Continuação para o método **PUT**:

```
// 3 - Testando uma requisição PUT
Add only
test('Deve atualizar um post existente', async () => {
  const updatedPost = {
    id: 1,
    title: 'Título atualizado',
    body: 'Conteúdo atualizado no teste',
    userId: 1
  }

  const response = await request(API_URL)
    .put('/posts/1')
    .send(updatedPost)
    .set('Content-Type', 'application/json');

  expect(response.status).toBe(200); // Verifica se a atualização foi bem-sucedida
  expect(response.body).toMatchObject(updatedPost); // Confirma que o post foi atualizado corretamente
});
```

O que esse teste faz?

- Testa uma API real (JSONPlaceholder) sem precisar criar um servidor
- Faz requisições PUT,
- Verifica se o post foi editado com sucesso
- Garante que os status HTTP estão corretos (200)

Continuação para o método **DELETE**:

```
// 4 - Testando uma requisição DELETE
Add only
test('Deve deletar um post', async () => {
  const response = await request(API_URL).delete('/posts/1');

  expect(response.status).toBe(200); // O JSONPlaceholder retorna 200 para DELETE, mesmo sem deletar de fato
});
```

O que esse teste faz?

- Testa uma API real (JSONPlaceholder) sem precisar criar um servidor
- Faz requisições DELETE,
- Verifica se o post foi deletado com sucesso
- Garante que os status HTTP estão corretos (200)

3. K6

O K6 é uma ferramenta de testes de carga e desempenho desenvolvida pela Grafana Labs. Seu objetivo é ajudar equipes de desenvolvimento e qualidade a simular usuários acessando sistemas simultaneamente, identificando gargalos e garantindo a estabilidade das aplicações. O K6 se destaca por ser de código aberto, baseado em JavaScript para a definição de testes, e altamente otimizado para consumir poucos recursos.

Características do K6

- **Fácil de usar:** Testes escritos em JavaScript, facilitando a adoção por desenvolvedores.
- **Leve e eficiente:** Implementado em Go, consome menos recursos que ferramentas tradicionais como JMeter.
- **Integração com DevOps:** Suporta CI/CD, podendo ser integrado a pipelines no GitHub Actions, Jenkins, GitLab CI, entre outros.
- **Coleta de métricas:** Exporta dados para ferramentas como Grafana, Prometheus e InfluxDB.
- **Execução distribuída:** Possibilidade de rodar em contêineres e Kubernetes para testes em larga escala.

Vantagens do K6

Foco em desempenho: Projetado para executar testes de carga de forma eficiente, sem sobrecarregar a infraestrutura de testes.

Flexibilidade: Suporta diversos tipos de testes: stress, smoke, endurance, spike e carga.

Fácil automação: Pode ser integrado com pipelines DevOps para execução automática.

Desenvolvimento moderno: Utiliza JavaScript para definição de cenários de teste, tornando-o acessível para desenvolvedores frontend e backend.

3.1. Configuração do ambiente

A instalação do K6 varia conforme o sistema operacional:

- **Linux (Ubuntu):** No seu terminal rode o comando: **sudo snap install k6**
- **Windows:** Se estiver usando o Chocolatey, execute o comando: **choco install k6**
- Também é possível realizar a instalação pelo próprio executável do K6 para Windows, acessando pelo link: [K6 Installer](#)
- **macOS:** Se estiver usando o Homebrew, execute o comando: **brew install k6**

Validando a Instalação

Verificando se o K6 foi instalado corretamente, execute o comando no terminal: **k6 version**
Se a instalação estiver correta, o terminal exibirá a versão do K6.

Exemplos de Testes

Aqui veremos um exemplo simples do uso de um endpoint para uma API online, crie um arquivo chamado **teste_api.js** e adicione o seguinte código:

Importação de Módulos

```
import http from 'k6/http';  
import { check, sleep } from 'k6';
```

Configuração do Teste

```
export let options = {  
  vus: 1,  
  duration: '10s'  
};
```

Configuração adicional completa do exemplo API

```
export default function () {  
  const url = 'https://jsonplaceholder.typicode.com/posts/1';  
  
  const res = http.get(url);  
  
  check(res, {  
    'status é 200': (r) => r.status === 200,  
    'o corpo não está vazio': (r) => r.body.length > 0  
  });  
  
  sleep(1);  
}
```

Execução do Teste

```
const url = 'https://jsonplaceholder.typicode.com/posts/1';  
  
const res = http.get(url);
```

- O usuário virtual **acessa um endpoint de API pública**, simulando a requisição de um **recurso específico** (nesse caso, um get de um blog fictício).

Validação da Resposta

```
check(res, {  
  'status é 200': (r) => r.status === 200,  
  'o corpo não está vazio': (r) => r.body.length > 0  
});
```

- **Status HTTP:** Confirma que a API responde com código **200 (sucesso)**.
- **Conteúdo da resposta:** Garante que a API **retornou um corpo válido** (não vazio).

Intervalo Entre Requisições

```
sleep(1);
```

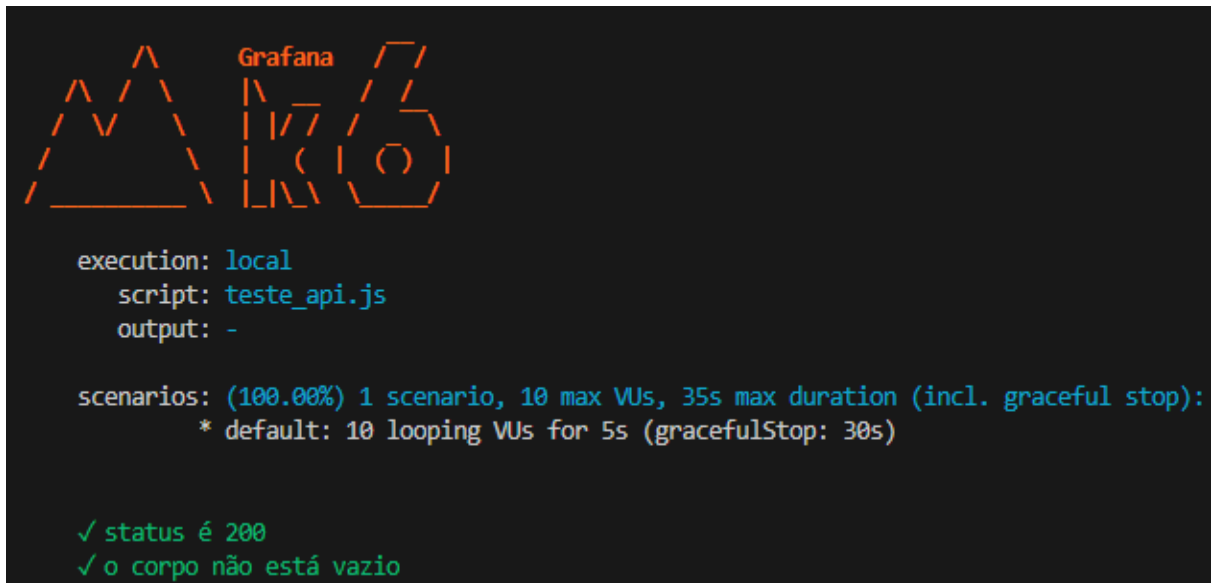
- O usuário **espera 1 segundo** antes de repetir o teste.

Como Executar o Teste

No terminal, navegue até a pasta onde o arquivo está salvo e execute o comando:

```
k6 run teste_basico.js
```

Resultado do Teste da API



```

execution: local
  script: teste_api.js
  output: -

scenarios: (100.00%) 1 scenario, 10 max VUs, 35s max duration (incl. graceful stop):
  * default: 10 looping VUs for 5s (gracefulStop: 30s)

✓ status é 200
✓ o corpo não está vazio

```

Principais Métricas Geradas

- **Latência e tempo de resposta:** http_req_duration, http_req_waiting, p(90), p(95).
- **Erros e estabilidade:** http_req_failed, vus, iterations.
- **Capacidade de carga:** vus_max, http_reqs.
- **Consumo de rede:** data_received, data_sent.

As métricas irão aparecer logo abaixo do exemplo acima no terminal de comando.

O Que Acontece Durante a Execução

O K6 inicia com **1 usuário virtual**.

Esse usuário faz uma **requisição GET** para a API **a cada segundo**, por **10 segundos**.

O teste verifica se a API responde corretamente (**status 200**) e se retorna um **conteúdo válido**.

O K6 gera um relatório com **tempo de resposta**, **taxas de erro** e **outras métricas**.

3.2. Relatórios

Além de ser possível criar uma gama ampla de testes, o mesmo oferece suporte para a geração do relatório ao final de cada teste realizado. Crie um arquivo chamado **teste_relatorio.js** ou implementa-o em seu código já existente:

Importação de Módulos

```

import { htmlReport } from "https://raw.githubusercontent.com/benc-uk/k6-reporter/main/dist/bundle.js";

export function handleSummary(data) {
  return {
    "summary.html": htmlReport(data)
  };
}

```

Observações Finais

Após rodar os testes será gerado um arquivo conforme o nome que você deu, nesse nosso exemplo o nome é **Summary.html**, será gerado na raiz de onde o teste se encontra.

3.2. Métricas

Os testes de performance realizados com o K6 geram diversas métricas que ajudam a entender como uma aplicação se comporta sob diferentes cargas de usuários. Essas métricas são fundamentais para identificar gargalos, problemas de latência e o impacto do aumento de requisições na API ou sistema.

Principais Objetivos da Análise de Métricas

- Identificar tempos de resposta altos e otimizar a performance da aplicação.
- Detectar pontos de falha quando a API recebe um alto volume de requisições.
- Verificar limites de capacidade e entender quantos usuários simultâneos o sistema suporta.
- Medir taxas de erro para garantir que a aplicação não retorne falhas sob carga.
- Avaliar a escalabilidade do sistema, garantindo que ele suporte um crescimento de tráfego.

Principais Métricas

http_req_duration

O que é?: Essa métrica indica quanto tempo, em média, uma requisição demora para ser processada pela API, desde o envio até a resposta.

Como analisar?

- Se o tempo médio (avg) estiver alto (exemplo: > 1s), a API pode estar sobrecarregada ou mal otimizada.
- O percentil 95 (p(95)) indica o tempo máximo de resposta para 95% das requisições. Se for alto, pode haver instabilidade.
- Se a máxima (max) estiver muito maior que a média, significa que há picos de lentidão.

http_reqs

O que é?: Mede quantas requisições o servidor está processando por segundo.

Como analisar?

- Se esse valor for baixo e o sistema estiver lento, o servidor pode estar recusando conexões.
- Se esse valor cai muito ao longo do tempo, o servidor pode estar ficando sobrecarregado.

http_req_connecting

O que é?: Mede o tempo necessário para o cliente se conectar ao servidor.

Como analisar?

- Se esse valor for alto (exemplo: > 200ms), pode haver problemas na infraestrutura da rede.
 - Se a conexão varia muito, pode indicar instabilidade no servidor.
-

http_req_failed

O que é?: Mede quantas requisições falharam devido a timeouts, status HTTP de erro (500, 404, 403) ou problemas de rede.

Como analisar?

- Se a taxa for alta (> 2%), pode haver problemas na aplicação ou um limite de requisições atingido.
 - Se a taxa for 0%, o servidor está respondendo bem sob carga.
-

vus e vus_max

O que é?: Mostra quantos usuários virtuais (VUs - Virtual Users) estão rodando no teste.

Como analisar?

- Se o sistema começar a falhar ao aumentar os VUs, pode haver limitação na infraestrutura.
- Testes podem ser feitos com diferentes cargas para verificar quando o sistema começa a falhar.

3.4. Tipos de Testes

Vamos criar um teste de performance básico com K6, onde simularemos múltiplos usuários acessando uma API pública (JSONPlaceholder). Esse teste irá medir:

- Tempo de resposta das requisições
- Taxa de erro das requisições
- Capacidade da API para suportar múltiplos usuários simultâneos

Criando o arquivo **test_performance.js**:

```
import http from 'k6/http';
import { check, sleep } from 'k6';

// Configurações do teste
export let options = {
  stages: [
    { duration: '10s', target: 10 }, // 10 usuários simultâneos em 10s
    { duration: '30s', target: 50 }, // Sobe para 50 usuários em 30s
    { duration: '20s', target: 10 } // Reduz para 10 usuários em 20s
  ],
  thresholds: {
    http_req_duration: ['p(95)<500'], // 95% das requisições devem ser < 500ms
    http_req_failed: ['rate<0.01'] // Menos de 1% de erro permitido
  }
}

export default function () {
  let res = http.get('https://jsonplaceholder.typicode.com/posts/1');

  // Validações
  check(res, {
    'Status é 200': (r) => r.status === 200,
    'Tempo de resposta < 500ms': (r) => r.timings.duration < 500,
  });

  sleep(1); // Simula o tempo de espera entre requisições
}
```

O que esse teste faz?

- Simula usuários simultâneos crescendo de 10 → 50 → 10 ao longo do tempo.
- Faz requisições GET para um endpoint da API.
- Verifica tempo de resposta e se o status retornado é 200.
- Dorme 1 segundo entre cada requisição para simular comportamento real de usuários.

Teste de Carga

O teste de carga verifica como um sistema se comporta sob um alto volume de requisições, ajudando a identificar gargalos e limitações de infraestrutura.

Criando o arquivo **test_load.js**:


```
import http from 'k6/http';
import { check, sleep } from 'k6';

export let options = {
  stages: [
    { duration: '30s', target: 50 }, // Sobe para 50 usuários em 30s
    { duration: '1m', target: 100 }, // Aumenta para 100 usuários em 1 minuto
    { duration: '2m', target: 200 }, // Mantém 200 usuários por 2 minutos
    { duration: '1m', target: 100 }, // Reduz para 100 usuários em 1 minuto
    { duration: '30s', target: 50 } // Diminui para 50 usuários em 30s
  ],
  thresholds: {
    http_req_duration: ['p(95)<1000'], // 95% das requisições devem ser < 1s
    http_req_failed: ['rate<0.02'] // Menos de 2% de falhas são aceitáveis
  }
}

export default function () {
  let res = http.get('https://jsonplaceholder.typicode.com/posts');

  // Validações
  check(res, {
    'Status é 200': (r) => r.status === 200,
    'Tempo de resposta < 1000ms': (r) => r.timings.duration < 1000,
  });

  sleep(1); // Simula tempo entre as requisições
}
```

O que esse teste faz?

- Simula um aumento gradual de usuários (50 → 100 → 200).
- Mantém 200 usuários simultâneos por 2 minutos.
- Verifica se o tempo de resposta se mantém abaixo de 1s.
- Analisa se a API falha com alto volume de requisições.

4. Playwright

O Playwright é uma ferramenta de automação de testes de código aberto desenvolvida pela Microsoft. Ele permite testar aplicações web e APIs de forma eficiente e confiável, suportando múltiplos navegadores e dispositivos.

Qual o objetivo do uso do Playwright?

- Automatizar testes de UI (Interface do Usuário) em aplicações web.
- Executar testes em diferentes navegadores (Chromium, Firefox, WebKit).
- Realizar testes em múltiplos dispositivos e contextos (desktop, mobile, headless).
- Testar APIs diretamente, sem precisar de outra ferramenta.
- Realizar testes visuais, de acessibilidade e de desempenho.

4.1. Instalação do Playwright

Crie um projeto chamado **teste-playwright** e rode o seguinte comando:

```
npm init playwright@latest
```

Agora basta seguir o passo a passo da configuração:

- Seleção da linguagem
- Nome da pasta onde ficarão os testes, por padrão o nome é tests
- Adicionar ou não o arquivo de pipelines (Workflow), por padrão é false
- Instalar o navegador Playwright, por padrão é true

Fluxo completo do que foi selecionado neste exemplo:

```
✓ Do you want to use TypeScript or JavaScript? · JavaScript
✓ Where to put your end-to-end tests? · tests
✓ Add a GitHub Actions workflow? (y/N) · false
✓ Install Playwright browsers (can be done manually via 'npx playwright install')? (Y/n) · true
Initializing NPM project (npm init -y)...
```

Após as configurações iniciais o Playwright será instalado no seu projeto.

4.2. Tags Playwright

As tags são usadas para organizar e filtrar testes, facilitando a execução de testes específicos sem precisar rodar toda a suíte de testes, algumas características do uso:

- Executar apenas testes específicos sem rodar todos.
- Organizar diferentes tipos de testes (Ex: @login, @api, @checkout).
- Melhorar a performance dos testes rodando apenas os necessários.
- Facilitar integração com CI/CD, permitindo rodar apenas testes críticos.

Para rodar um teste com tag basta usar o comando: **npx playwright test --grep '@login'**.

4.3. Actions

As Actions são interações que o Playwright executa na página, como clicar, preencher campos, pressionar teclas e navegar. Elas simulam ações de um usuário real.

Exemplos de Actions no Playwright:

```
await page.goto('https://www.saucedemo.com/'); // Navegar até um site
await page.click('[data-test="login-button"]'); // Clicar em um botão
await page.fill('[data-test="username"]', 'standard_user'); // Preencher um campo
await page.press('[data-test="password"]', 'Enter'); // Pressionar tecla
```

4.4. Assertions

As Assertions (Asserções) são validações usadas para garantir que a página se comporta como esperado. Elas verificam se textos, URLs, elementos e estados estão corretos após as ações.

Exemplos de Assertions no Playwright:

```
await expect(page).toHaveURL('https://www.saucedemo.com/inventory.html'); // Verifica a URL
await expect(page.locator('[data-test="login-button"]')).toBeVisible(); // Verifica se um botão está visível
await expect(page.locator('.title')).toHaveText('Products'); // Verifica o texto de um elemento
```

Escrevendo os Testes

Dentro da pasta tests crie o arquivo **login.spec.js**:

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

Add only
test.beforeEach(async ({ page }) => {
  await page.goto('https://www.saucedemo.com/');
});

Add only
test('Login com sucesso. @login', async ({ page }) => {
  await page.locator('[data-test="username"]').click();
  await page.locator('[data-test="username"]').fill('standard_user');
  await page.locator('[data-test="username"]').press('Tab');
  await page.locator('[data-test="password"]').fill('secret_sauce');
  await page.locator('[data-test="login-button"]').click();

  const texto = await page.waitForSelector('text=Products');
  await texto.scrollIntoViewIfNeeded();
});
```

O que esse teste faz?

Passos de Configurações Iniciais

- beforeEach executa um passo antes de cada teste.
 - Neste caso, abre o site SauceDemo antes de executar qualquer teste.
- Define um teste de login com a descrição "Login com sucesso. @login".
 - @login pode ser usado para filtrar testes específicos no Playwright.

Passos de Preenchimento dos campos

- Clica no campo de usuário.
- Preenche com "standard_user".

- Pressiona a tecla Tab para mover para o próximo campo.
- Preenche a senha com "secret_sauce".
- Clica no botão de login.

Passos de validações

- Espera o elemento "Products" aparecer na tela, indicando que o login foi bem-sucedido.
- Rola a página até esse elemento, se necessário.

Teste de API

Crie o arquivo **api.spec.js**:

```
const { test, expect } = require('@playwright/test');

Add only
test('Deve buscar um post via API. @api', async ({ request }) => {
    const response = await request.get('https://jsonplaceholder.typicode.com/posts/1');

    expect(response.status()).toBe(200);

    const body = await response.json();

    expect(body).toHaveProperty('id', 1);
    expect(body).toHaveProperty('title');
});
```

O que este teste faz?

- Faz uma requisição GET para a API JSONPlaceholder.
- Verifica se a resposta tem status 200.
- Valida se o corpo da resposta contém os campos esperados.

Exibindo os Relatórios

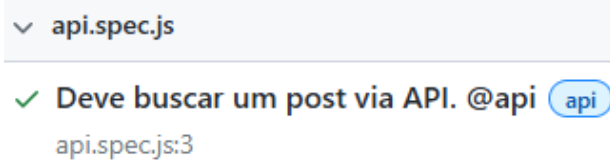
Após um teste será gerado o relatório, podendo abrir com o comando: **npx playwright show-report**, abaixo o exemplo após rodar um teste:

```
Running 1 test using 1 worker
  1 passed (985ms)

To open last HTML report run:

npx playwright show-report
```

Ao abrir o relatório será exibido todo resumo do que foi testado, exemplo a seguir:



Ao clicar sobre, será exibido todos os passos que levaram à execução final do teste.

4.5. Playwright Interface

Além de poder executar os testes via terminal, também é possível realizar os testes via interface que o próprio Playwright possui integrado, basta rodar o seguinte comando via terminal: **npx playwright test --ui**.

4.6. Playwright Gerador de Testes

O Playwright possui uma funcionalidade chamada **Codegen**, que permite gravar suas interações na página e gerar automaticamente o código dos testes enquanto você navega. Isso torna o processo mais dinâmico e intuitivo, ideal para quem quer criar testes sem precisar escrever todo o código manualmente.

Via terminal rode o comando: **npx playwright codegen https://www.seusite.com/** isso abrirá um navegador controlado pelo Playwright e um editor de código ao lado. Agora basta navegar pelo site que o código será gerado dinamicamente e após o código gerado você poderá copiar o mesmo e implementar em seu projeto.

5. Faker.js

O **Faker.js** é uma biblioteca JavaScript amplamente utilizada para gerar dados falsos ou fictícios de maneira rápida e fácil. Ele é útil principalmente em ambientes de desenvolvimento e teste, onde dados realistas são necessários para simular cenários e validar funcionalidades, mas o uso de dados reais seria impróprio ou inviável.

Principais Recursos do Faker.js

- Gera dados como nomes, endereços, números de telefone, textos, datas, valores financeiros, entre outros.
- Suporte para várias localizações (idiomas e formatos específicos de cada país).
- Fácil integração em projetos JavaScript e Node.js.

Por que usar Faker.js?

1. **Testes de Software:** Gera dados de entrada para testes automatizados.
2. **Prototipação:** Preenche interfaces de usuário com dados realistas.
3. **Desenvolvimento de APIs:** Simula respostas de APIs durante o desenvolvimento.

4. **Evitando Dados Reais:** Reduz riscos relacionados ao uso de dados sensíveis ou privados.

Instalação

Para usar o Faker.js, você pode instalá-lo via npm:

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

Uso Básico

Importação da biblioteca:

```
import { faker } from '@faker-js/faker';
```

Comandos básicos de geração de informações:

```
const randomName = faker.person.fullName();
const randomEmail = faker.internet.email();
```

Com esse exemplo agora será possível usar esses dados fictícios em alguma lógica que você precisa que os dados sempre sejam diferentes.

Lista de Atividades - Automação de Testes - Cypress

Desafio 1: Testes de Login

Objetivo: O aluno deve validar se a página de login do SauceDemo está funcionando corretamente, garantindo que os campos e botões estejam visíveis e interativos, além de verificar mensagens de erro ao inserir dados inválidos.

O que será feito?

- Acessar o site [OrangeHrm](https://www.orangehrm.com/).
- Validar a presença dos elementos da tela de login (campos de usuário e senha, botão de login, logo do site).
- Testar o funcionamento dos campos de entrada (verificar se é possível digitar nos campos).
- Testar o comportamento do botão de login (verificar se ele pode ser clicado).
- Verificar mensagens de erro ao inserir dados incorretos.

O que o aluno deve observar?

Verificando os Elementos na Tela

- O site carrega corretamente?
- Os campos de usuário e senha estão visíveis?
- O botão de login aparece corretamente na tela?
- O título da página está correto?

Testando a Interatividade dos Campos

- O aluno consegue digitar no campo de usuário?
- O aluno consegue digitar no campo de senha?
- O botão de login pode ser clicado?

Testando o Login com Dados Inválidos

- O site exibe uma mensagem de erro ao tentar fazer login com usuário ou senha errados?
- A mensagem de erro desaparece quando o aluno tenta corrigir os dados e tentar novamente?

O que será avaliado?

- Código organizado e comentado.
- Uso de Page Objects.
- Código executável e funcional.

Os acadêmicos deverão entregar os seguintes itens:

- Scripts de testes desenvolvidos no repositório GitHub.
- Documentação explicativa sobre o processo de desenvolvimento e execução dos testes.

Boa sorte e bom aprendizado!

Desafio 2: Chamadas de API

Objetivo: O aluno deverá explorar a API pública [Json Placeholder](#) realizando testes automatizados no Cypress para os quatro principais métodos de requisição HTTP:

GET: Buscar informações

POST: Criar um novo dado

PUT: Atualizar um dado existente

DELETE: Remover um dado

O que será feito?

- Realizar uma consulta (GET) para obter uma lista de posts e validar se os dados estão corretos.
- Cria um novo post (POST) e verificar se ele foi salvo corretamente.
- Editar um post existente (PUT) e confirmar que a atualização foi feita com sucesso.
- Delete um post (DELETE) e validar que ele foi removido.

O que será avaliado?

- Código organizado e comentado.
- Uso de Commands.
- Uso de Page Objects.
- Código executável e funcional.

Os acadêmicos deverão entregar os seguintes itens:

- Scripts de testes desenvolvidos no repositório GitHub.
- Documentação explicativa sobre o processo de desenvolvimento e execução dos testes.

Boa sorte e bom aprendizado!

Atividades Parte 2

Os alunos deverão implementar dois tipos de testes:

Teste de Integração (API): Valida o comportamento dos métodos GET e POST da API pública [Json Placeholder](https://jsonplaceholder.typicode.com/).

Teste Unitário: Cria uma função simples que manipula os dados retornados da API e valida seu comportamento.

Criar um teste de integração para um método GET para:

```
//https://jsonplaceholder.typicode.com/posts/1
```

Critérios de validação:

- O código de resposta deve ser 200.
- O corpo da resposta deve conter uma lista com as propriedades id, title e body.

Criar uma função para manipular os dados da API

Critérios de validação:

- A função deve receber um array de posts e retornar apenas os títulos.
- O teste unitário deve verificar se a função retorna os títulos corretamente.

Interpretação dos Resultados

Após rodar os testes, os alunos devem responder às seguintes perguntas:

- O teste GET passou com status correto (200)?
- Os dados da API continham as propriedades esperadas (id, title, body)?
- A função extrairTítulos() retornou os títulos corretamente?
- Se algum teste falhou, qual pode ter sido a causa?
- O que pode ser melhorado nos testes para aumentar a confiabilidade?

Os acadêmicos deverão entregar os seguintes itens:

- Scripts de testes desenvolvidos no repositório GitHub.
- Documentação explicativa sobre o processo de desenvolvimento e execução dos testes.

Boa sorte e bom aprendizado!

Lista de Atividades - Automação de Testes - K6

Esta atividade tem como objetivo avaliar o conhecimento dos alunos sobre a ferramenta K6, utilizada para testes de performance, carga e estresse.

A atividade contém:

Os alunos devem criar e executar testes de carga para os métodos GET e POST usando o K6 e a API pública [Json Placeholder](https://jsonplaceholder.typicode.com/). (**Consulte a documentação e cuide com as configurações de requisição**).

O que será avaliado?

- Criação de um teste GET para buscar um post e validar o tempo de resposta.
- Criação de um teste POST para enviar um novo post e verificar se ele foi criado corretamente.
- Execução dos testes em um cenário de carga de 10 usuários simultâneos durante 1 minuto.
- Interpretação dos resultados do teste e análise da performance da API.

Instruções para o Desafio

Criar um teste GET para acessar o endpoint:

```
//https://jsonplaceholder.typicode.com/posts/1
```

Critérios de validação:

- O código de resposta deve ser 200.
- O tempo de resposta deve ser inferior a 500ms em 95% das requisições.

Criar um teste POST para enviar um novo post para o endpoint:

```
//https://jsonplaceholder.typicode.com/posts
```

Critérios de validação:

- O código de resposta deve ser 201 (indica que o recurso foi criado com sucesso).
- O tempo de resposta deve ser inferior a 500ms em 95% das requisições.

Interpretação dos Resultados

Após rodar o teste, os alunos devem analisar os resultados e responder às seguintes perguntas:

- O tempo de resposta do método GET ficou abaixo de 500ms em 95% das requisições?
- O método POST respondeu corretamente com status 201?
- A API conseguiu manter a performance com 50 usuários simultâneos?
- A taxa de falhas (http_req_failed) ficou abaixo de 1%?
- Quais possíveis melhorias poderiam ser feitas para otimizar a API e reduzir tempos de resposta?

Os acadêmicos deverão entregar os seguintes itens:

- Scripts de testes desenvolvidos no repositório GitHub.
- Documentação explicativa sobre o processo de desenvolvimento e execução dos testes.

Boa sorte e bom aprendizado!

Lista de Atividades - Automação de Testes - Jest

Nesta atividade, os alunos deverão implementar testes de API REST utilizando o Jest, testando os métodos GET e POST na API pública [Json Placeholder](https://jsonplaceholder.typicode.com/).

A atividade contém:

Os alunos deverão implementar dois tipos de testes:

Teste de Integração (API): Valida o comportamento dos métodos GET e POST da API pública [Json Placeholder](https://jsonplaceholder.typicode.com/)..

Teste Unitário: Cria uma função simples que manipula os dados retornados da API e valida seu comportamento.

Criar um teste de integração para um método GET para:

```
//https://jsonplaceholder.typicode.com/posts/1
```

Critérios de validação:

- O código de resposta deve ser 200.
- O corpo da resposta deve conter uma lista com as propriedades id, title e body.

Criar uma função para manipular os dados da API

Critérios de validação:

- A função deve receber um array de posts e retornar apenas os títulos.
- O teste unitário deve verificar se a função retorna os títulos corretamente.

Interpretação dos Resultados

Após rodar os testes, os alunos devem responder às seguintes perguntas:

- O teste GET passou com status correto (200)?
- Os dados da API continham as propriedades esperadas (id, title, body)?
- A função extrairTítulos() retornou os títulos corretamente?
- Se algum teste falhou, qual pode ter sido a causa?
- O que pode ser melhorado nos testes para aumentar a confiabilidade?

Os acadêmicos deverão entregar os seguintes itens:

- Scripts de testes desenvolvidos no repositório GitHub.
- Documentação explicativa sobre o processo de desenvolvimento e execução dos testes.

Boa sorte e bom aprendizado!

Lista de Atividades Extra - Automação de Testes - Playwright

Crie um projeto implementando tudo que aprendeu sobre esse framework. Lambesse das boas práticas, código limpo, estruturado e publicado no gitHub.

Boa sorte e bom aprendizado!