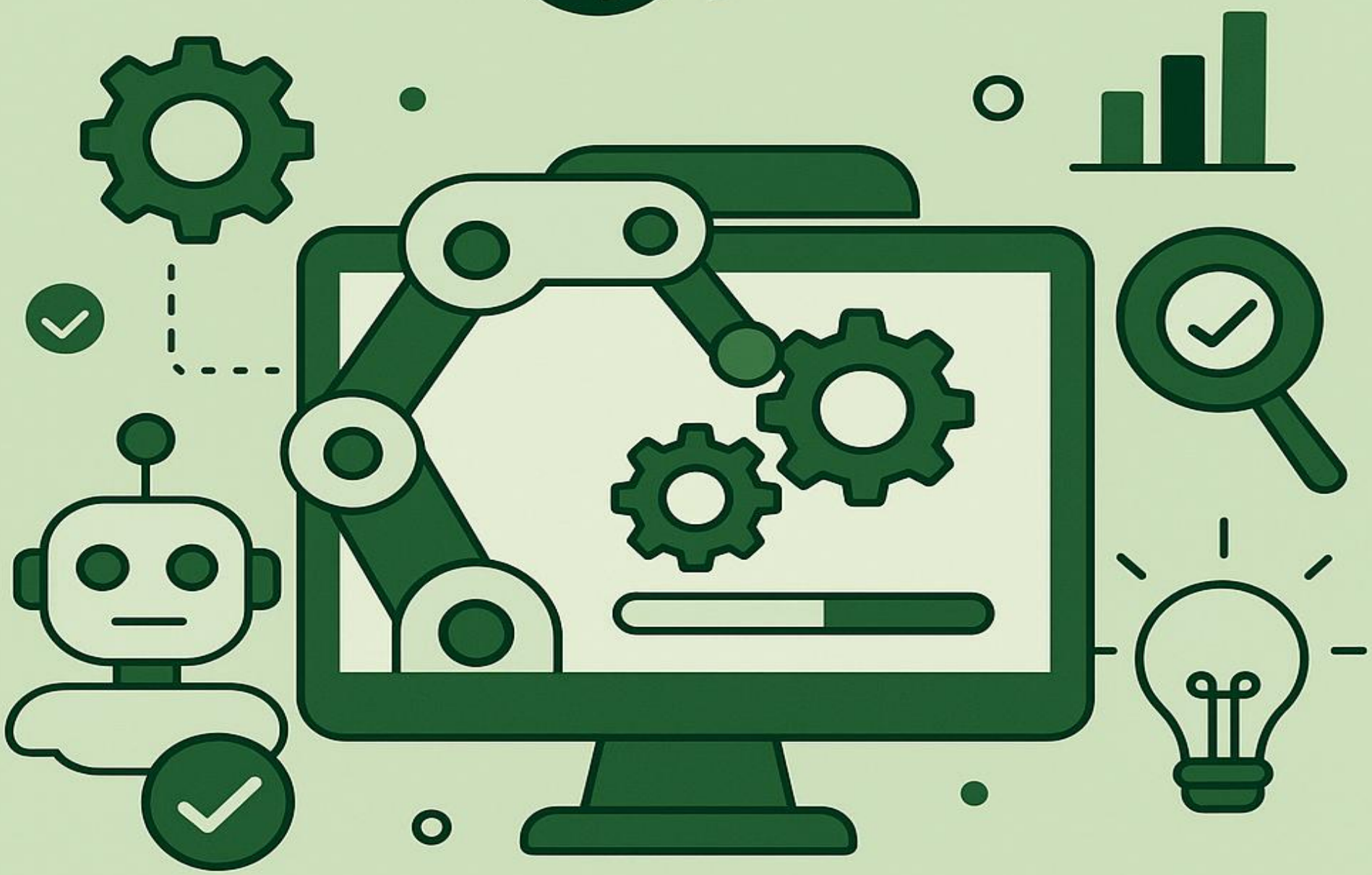
DOMINE A AUTOMAÇÃO INTELIGENTE

COM cypress



cypress

INTRODUÇÃO

Simplificando os Testes de Softwares

A automação de testes é uma das chaves para desenvolver software com qualidade, rapidez e confiança. Com o Cypress, você pode criar testes modernos, visuais e inteligentes — ideais para validar fluxos críticos como login, cadastro e navegação.

💡 Este eBook é um guia prático e direto para quem quer dominar a automação de testes do jeito certo.

01

CONCEITOS DE AUTOMAÇÃO

A automação de testes ajuda times a validar funcionalidades mais rápido, reduzir erros e entregar software com mais qualidade. Em vez de testar tudo manualmente, scripts automatizados assumem o trabalho repetitivo.

CONCEITOS DE AUTOMAÇÃO

Por que automatizar

- Reduz erros manuais
- Aumenta a produtividade
- Facilita regressões e refatorações
- Garante estabilidade em deploys contínuos

O papel do Cypress

O Cypress é um framework moderno de **testes end-to-end em JavaScript**, fácil de instalar e com execução visual.
Ele roda **diretamente no navegador**, mostrando cada passo do teste em tempo real.

02

INTRODUÇÃO AO CYPRESS

O Cypress se destaca por ser rápido, moderno e fácil de usar, principalmente por rodar dentro do navegador.

INTRODUÇÃO AO CYPRESS

O **Cypress** é um framework de testes end-to-end baseado em JavaScript. Ele roda diretamente no navegador, permitindo observar a execução dos testes em tempo real e depurar facilmente quando algo falha.

🎯 Exemplo de Teste de Login

javascript — Copiar código

```javascript
describe('Teste de Login', () => {
  it('Deve fazer login com sucesso', () => {
    cy.visit('https://meusistema.com/login')
    cy.get('#email').type('usuario@teste.com')
    cy.get('#senha').type('123456')
    cy.get('button[type="submit"]').click()
    cy.contains('Bem-vindo, usuário!').should('be.visible')
  })
})
```

💡 ***Dica:*** o Cypress já aguarda automaticamente os elementos ficarem visíveis, então raramente é necessário usar `wait()`.

✅ **Estrutura de Comandos**

- `cy.visit()` → navega
- `cy.get()` → seleciona elementos
- `cy.contains()` → busca por texto
- `cy.click()` → clica
- `cy.type()` → digita

03

SELETORES DE ELEMENTO

Os seletores de elemento permitem que você direcione um elemento HTML específico com base em seu nome de tag. Eles são simples e diretos. Vamos ver alguns exemplos:

SELETOR POR ID

O **seletor por ID** é uma das maneiras **mais rápidas e precisas** de localizar um elemento em uma página. Cada elemento no HTML pode ter um atributo único chamado id, que serve como um identificador exclusivo.
No Cypress, usamos o **símbolo #** para selecionar por **ID**, assim como no **CSS**.

Formulário de login:

javascript — Copiar código

```
cy.get('#email').type('admin@teste.com')
cy.get('#senha').type('123')
cy.get('#btnEntrar').click()
```

🧩 **Vantagens**

- **Alta performance:** o Cypress encontra o elemento rapidamente.
- **Legibilidade:** o código fica claro mesmo para quem não é desenvolvedor.
- **Confiabilidade:** IDs raramente mudam, o que evita que testes quebrem.

⚠️ **Cuidados**

- Verifique se o **ID é realmente único**. Se o mesmo ID aparecer em mais de um elemento, o Cypress pode selecionar o errado.
- **Evite IDs dinâmicos**, como id="input_1234" que muda a cada renderização.

SELETOR POR CLASSE

O **seletor por classe** é uma das formas mais comuns de localizar elementos em uma página web. No HTML, o atributo **class** é usado para definir o estilo visual de um elemento (como cor, tamanho, posição), mas também pode ser usado para identificar componentes na automação de testes.
No Cypress, você utiliza um **ponto (.)** antes do nome da classe — o mesmo padrão usado em CSS.

Clicar no botão de salvar:

javascript — Copiar código

```
cy.get('.btn-salvar').click()
```

🧩 **Vantagens**

- **Simples e intuitivo:** segue a lógica do CSS.
- **Fácil de aplicar**: praticamente todo elemento visual tem uma classe.

⚠️ **Cuidados**

- **Classes nem sempre são únicas:** Muitas vezes, várias partes do sistema usam a mesma classe (ex: .btn em todos os botões).
  Isso pode causar erros de seleção, clicando no elemento errado.
- **Mudanças de layout podem quebrar testes:** Desenvolvedores front-end costumam alterar classes com frequência ao ajustar estilos (ex: .btn-login → .botao-entrar).
- **Evite classes genéricas:** Se o código tiver .input, .form, .container, o teste pode se tornar instável e impreciso.

SELETOR POR ATRIBUTO

O **seletor por atributo** é considerado o **método** mais moderno e recomendado para localizar elementos em aplicações web, especialmente em ambientes corporativos. Ele é usado para encontrar elementos HTML com base em atributos personalizados, como data-test, data-cy ou data-qa. Esses atributos são criados exclusivamente para automação de testes e não afetam o layout, o CSS nem a funcionalidade do sistema.

```javascript
cy.get('[data-test="botao-enviar-form"]').click()
```

Copiar código

🧩 **Vantagens**

- **Estabilidade máxima:** alterações visuais não afetam o teste.
- **Legibilidade:** o código reflete exatamente o que o usuário faz.
- Compatível com qualquer framework front-end (React, Angular, Vue etc.)

⚠️ **Cuidados e boas práticas**

- Padronize o nome do atributo em toda a equipe(data-test, data-cy, data-qa).
- Evite usar nomes genéricos — prefira algo descritivo:
  - ❌ [data-test="button"]
  - ✅ [data-test="botao-login"]
- Evite espaços e caracteres especiais no valor do atributo.

SELETOR POR TEXTO

O **seletor por texto** é uma das formas mais **naturais e humanas** de encontrar elementos na tela.
Em vez de procurar por IDs ou classes, ele busca pelo **texto visível** que o usuário vê no navegador.
No Cypress, usamos o comando **cy.contains()** para isso.
Esse seletor é especialmente útil para **validar mensagens**, **rótulos**, **botões** e **textos exibidos após ações**.

```javascript
cy.contains('Entrar').click()
cy.contains('Bem-vindo, usuário!').should('be.visible')
```

Copiar código

🧩 **Vantagens**

- **Leitura natural:** o código fica quase como uma frase em português**.**
  Exemplo: “Clique em ‘Entrar’ e veja ‘Bem-vindo’.”
- **Ótimo para validações visuais**, como mensagens de sucesso ou erro.
- **Independe de atributos HTML** — ideal quando você não tem acesso ao código-fonte da página

⚠️ **Cuidados e boas práticas**

- **Mudanças no texto** (acentos, maiúsculas, traduções ou espaços) podem quebrar o teste.
- **Evite depender** apenas do texto em sistemas multilíngues.
- Se houver vários elementos com o mesmo texto (ex: vários botões “Editar”), o Cypress selecionará o primeiro encontrado — use contexto hierárquico para resolver isso.

SELETORES HIERÁRQUICOS

O **seletor hierárquico** é usado quando você quer **encontrar um elemento específico dentro de outro elemento**.
Ele é extremamente útil quando existem **vários elementos semelhantes** na página (como formulários, cards, ou botões repetidos).
Em outras palavras:

💬 *Você "entra" em uma área específica da página (pai) e procura algo dentro dela (filho).*

🧱 **Como funciona**

No Cypress, combinamos dois comandos:

- cy.get() → seleciona o elemento pai (o contêiner principal)
- .find() → busca um ou mais elementos dentro desse pai

Procurar elementos dentro de um contêiner. A estrutura fica assim:

```javascript
cy.get('elemento-pai').find('elemento-filho')
```

Quando você precisa navegar dentro de um container.

```javascript
cy.get('.lista-produtos')
  .find('li')
  .contains('Notebook Dell')
  .click()
```

SELETORES HIERÁRQUICOS

Localiza o classe de e-mail e, internamente, encontra o campo de e-mail e digita o seu e-mail.

```javascript
cy.get('.form-login').find('input[name="email"]').type('teste@teste.com')
```

💡 *Use quando o mesmo elemento aparece em diferentes áreas da tela.*

🧠 **Quando usar Seletores Hierárquicos**

- Quando há **elementos repetidos** com o mesmo nome ou classe (ex: vários botões “Salvar”).
- Para testar **componentes específicos** dentro de páginas complexas.
- Para **organizar** o escopo dos testes, tornando-os mais legíveis.

🧩 **Vantagens**

- **Evita ambiguidades** quando há múltiplos elementos iguais.
- Deixa os testes **mais organizados e sem conflito de seletores**.
- **Aumenta a legibilidade** ao mostrar claramente onde o elemento está.

⚠️ **Cuidados**

- **Não exagere nas camadas**. Evite cadeias muito longas como:

```javascript
cy.get('.container .form .box .input .field .email')
```

Isso torna o teste frágil e difícil de manter.

- Prefira definir um ponto de entrada claro (por exemplo .form-login) e usar .find() ou .within() de forma simples.

04

GHERKIN E TESTES BDD

O Gherkin é uma linguagem simples que descreve comportamentos do sistema de forma legível para todos — desenvolvedores, QAs e gestores.

GHERKIN E TESTES BDD

Gherkin e BDD são conceitos de desenvolvimento ágil que trabalham juntos: **BDD** (Behavior Driven Development) é uma metodologia que foca no comportamento do sistema. **Gherkin** é a linguagem que descreve esses comportamentos de forma simples, conectando times técnicos e não técnicos e servindo como base para testes automatizados.

✍️ **Estrutura Gherkin**

- **Given (Dado)** → contexto inicial
- **When (Quando)** → ação executada
- **Then (Então)** → resultado esperado

📘 **Exemplo Gherkin de Login**

gherkin — Copiar código

```gherkin
Feature: Login de usuário

  Scenario: Login com credenciais válidas
    Given que o usuário está na página de login
    When ele informa email e senha válidos
    Then ele deve ver a mensagem "Bem-vindo"
```

💻 **Implementação no Cypress**

javascript — Copiar código

```javascript
Given('que o usuário está na página de login', () => {
  cy.visit('/login')
})

When('ele informa email e senha válidos', () => {
  cy.get('#email').type('usuario@teste.com')
  cy.get('#senha').type('123456')
  cy.get('button[type="submit"]').click()
})

Then('ele deve ver a mensagem {string}', (mensagem) => {
  cy.contains(mensagem).should('be.visible')
})
```

05

BOAS PRÁTICAS PROFISSIONAIS

O uso de Page Objects e Fixtures (dados externos) é uma prática recomendada na automação de testes para melhorar a organização, a legibilidade, a manutenção e a reutilização do código.

BOAS PRÁTICAS PROFISSIONAIS
PAGE OBJECT

O **Page Object** é um padrão de design onde cada página (ou parte significativa) da interface do usuário (GUI) em um sistema web é representada por uma classe ou um objeto no código de automação.

🧠 Page Objects

Organize seus seletores e ações em classes separadas.

javascript — Copiar código

```javascript
class LoginPage {
  email = '#email';
  senha = '#senha';
  botao = '[data-test="login"]';

  login(usuario, senha) {
    cy.get(this.email).type(usuario)
    cy.get(this.senha).type(senha)
    cy.get(this.botao).click()
  }
}

export default new LoginPage();
```

💻 Uso no teste

javascript — Copiar código

```javascript
import loginPage from '../pages/LoginPage'

describe('Login com sucesso', () => {
  it('Deve exibir mensagem de boas-vindas', () => {
    cy.visit('/login')
    loginPage.login('user@teste.com', '123')
    cy.contains('Bem-vindo').should('be.visible')
  })
})
```

BOAS PRÁTICAS PROFISSIONAIS
FIXTURES

Fixtures são usados para preparar o ambiente de teste, incluindo dados de teste (como dados de login, informações de formulários, etc.), para que o teste seja executado de forma consistente e isolada.

📁 Onde ficam os fixtures

Por padrão, o Cypress cria uma pasta chamada:

```bash
cypress/fixtures
```

Dentro dela, você pode criar arquivos .json que armazenam informações que serão usadas nos testes.

⚙️ Uso de Fixtures (dados externos)

Arquivo de dados (usuario.json):

```json
// fixtures/usuario.json
{
  "email": "user@teste.com",
  "senha": "123"
}
```

🧠 Teste usando o fixture:

```javascript
describe('Login com Fixture', () => {
  it('Deve fazer login usando dados externos', () => {
    cy.fixture('usuario').then((dados) => {
      cy.visit('https://meusistema.com/login')
      cy.get('#email').type(dados.email)
      cy.get('#senha').type(dados.senha)
      cy.get('button[type="submit"]').click()
      cy.contains('Bem-vindo, usuário!').should('be.visible')
    })
  })
})
```

06

TESTES DE API COM CYPRESS

Testar APIs com Cypress envolve o uso do comando cy.request() para enviar requisições HTTP (GET, POST, etc.) e do comando cy.intercept() para simular ou interceptar respostas de API.

TESTES DE API COM CYPRESS

O **Cypress** não serve apenas para testes de interface (UI) — ele também é uma ótima ferramenta para **testar APIs** de forma rápida, simples e eficiente.
Com ele, é possível **enviar requisições HTTP** (como GET, POST, PUT e DELETE), **validar respostas** e até **criar dados** antes de executar testes na interface.

💬 *Pense nos testes de API como a base da automação: você garante que o "motor" da aplicação está funcionando antes de testar a "carroceria" (a interface).*

⚙️ **Comando Principal: cy.request()**

O comando cy.request() é usado para fazer chamadas HTTP diretas à API.

🧩 **Sintaxe básica:**

javascript — Copiar código

```javascript
cy.request({
  method: 'GET',
  url: 'https://api.meusistema.com/usuarios'
})
```

TESTANDO UM GET SIMPLES

📋 Exemplo 1 — Testando um GET simples

javascript — Copiar código

```javascript
describe('API - Listar Usuários', () => {
  it('Deve retornar a lista de usuários com sucesso', () => {
    cy.request('GET', 'https://api.meusistema.com/usuarios')
      .then((response) => {
        expect(response.status).to.eq(200)
        expect(response.body).to.have.property('usuarios')
        expect(response.body.usuarios).to.be.an('array')
      })
  })
})
```

💡 O que está acontecendo aqui:

- cy.request() envia uma requisição GET para a API.
- response.status verifica se a resposta foi **200 (OK)**.
- response.body contém o conteúdo retornado pela API.

TESTANDO UM POST SIMPLES

🔐 Exemplo 2 — Testando um POST (Login)

Agora, um teste realista de login por API — algo muito comum em automações.

javascript — Copiar código

```javascript
describe('API - Login', () => {
  it('Deve autenticar o usuário com sucesso', () => {
    cy.request({
      method: 'POST',
      url: 'https://api.meusistema.com/login',
      body: {
        email: 'usuario@teste.com',
        senha: '123456'
      }
    }).then((response) => {
      expect(response.status).to.eq(200)
      expect(response.body).to.have.property('token')
      expect(response.body.token).to.be.a('string')
    })
  })
})
```

✅ **Dica:**

O teste acima pode ser usado antes de testes de UI para **gerar o token de login automaticamente** — evitando precisar "digitar" o login toda vez.

TESTANDO UM PUT SIMPLES

🛠️ **Exemplo 3 — Testando um PUT — Atualizando Usuário**

Imagine que você já tem um usuário cadastrado na API e deseja **atualizar o nome e o e-mail** dele.

javascript Copiar código

```javascript
describe('API - Atualizar Usuário (PUT)', () => {
  it('Deve atualizar os dados do usuário com sucesso', () => {
    const usuarioAtualizado = {
      nome: 'Rafael Atualizado',
      email: 'rafael.atualizado@teste.com'
    }

    cy.request({
      method: 'PUT',
      url: 'https://api.meusistema.com/usuarios/123',
      body: usuarioAtualizado
    }).then((response) => {
      expect(response.status).to.eq(200)
      expect(response.body.nome).to.eq('Rafael Atualizado')
      expect(response.body.email).to.eq('rafael.atualizado@teste.com')
    })
  })
})
```

💡 **Explicação**

- **method**: 'PUT' → indica que queremos atualizar um recurso existente.
- **url** → deve incluir o **ID do usuário** que será alterado.
- **body** → contém os novos dados que serão enviados.
- **expect()** → valida se a resposta foi bem-sucedida e se os campos foram realmente atualizados.

✨ *Dica:* sempre teste campos específicos para garantir que a atualização aconteceu.

TESTANDO UM DELETE SIMPLES

🗑️ Exemplo 4 — Testando um DELETE — Removendo Usuário

Agora, vamos remover o mesmo usuário criado anteriormente.

javascript — Copiar código

```javascript
describe('API - Deletar Usuário (DELETE)', () => {
  it('Deve excluir o usuário com sucesso', () => {
    cy.request({
      method: 'DELETE',
      url: 'https://api.meusistema.com/usuarios/123'
    }).then((response) => {
      expect(response.status).to.eq(200)
      expect(response.body.message).to.eq('Usuário removido com sucesso!')
    })
  })
})
```

💡 Explicação

- **method**: 'DELETE' → indica a remoção do recurso.
- **url** → inclui o ID do item a ser excluído.
- **expect(response.status).to.eq(200)** → garante que a operação foi concluída com sucesso.
- A mensagem retornada (message) pode variar conforme a API — sempre confira a documentação.

VALIDAÇÃO APÓS O DELETE

🧠 Exemplo 5 — Validação após o DELETE

É uma boa prática **verificar se o item foi realmente removido**.
Você pode fazer isso com uma nova requisição GET logo em seguida:

javascript Copiar código

```javascript
cy.request({
  method: 'DELETE',
  url: 'https://api.meusistema.com/usuarios/123'
}).then(() => {
  cy.request({
    method: 'GET',
    url: 'https://api.meusistema.com/usuarios/123',
    failOnStatusCode: false
  }).then((res) => {
    expect(res.status).to.eq(404)
  })
})
```

⚙️ *failOnStatusCode: false evita que o Cypress interrompa o teste ao receber um erro 404 (que, neste caso, é esperado).*

USANDO O TOKEN GERADO

🧠 Exemplo 6 — Usando o Token Gerado

Você pode guardar o **token JWT** e reutilizá-lo em outras chamadas de API:

javascript — Copiar código

```javascript
let token

before(() => {
  cy.request('POST', 'https://api.meusistema.com/login', {
    email: 'usuario@teste.com',
    senha: '123456'
  }).then((res) => {
    token = res.body.token
  })
})

it('Deve listar os dados do perfil usando token', () => {
  cy.request({
    method: 'GET',
    url: 'https://api.meusistema.com/perfil',
    headers: { Authorization: `Bearer ${token}` }
  }).then((res) => {
    expect(res.status).to.eq(200)
    expect(res.body).to.have.property('nome')
  })
})
```

💬 *Essa técnica é útil para fluxos autenticados, onde várias requisições dependem de login.*

VALIDAÇÃO DE RESPOSTAS

🧩 Exemplo 7 — Validação de Respostas

O Cypress permite validar qualquer campo do retorno da API:

```javascript
expect(response.body.nome).to.eq('Rafael Sousa')
expect(response.body.idade).to.be.greaterThan(18)
expect(response.body.email).to.include('@')
```

📘 Vantagens de Testar APIs com Cypress

- **Sem dependência da interface:** os testes rodam mais rápido.
- **Identificação rápida de erros:** você sabe se o problema está na API ou no front-end.
- **Integração com UI:** é possível usar resultados da API dentro de testes visuais.
- **Fluxos completos:** dá para cadastrar, autenticar e validar dados sem precisar clicar em nada.

⚠️ Boas Práticas

- Evite chamar APIs externas reais — use **ambientes de teste ou mocks**.
- Centralize URLs e tokens em variáveis de ambiente (cypress.config.js).
- Use **fixtures** para armazenar dados de requisições repetidas.
- Nomeie os testes de forma clara, como api_login.spec.js ou api_usuarios.spec.js.

07

ERROS COMUNS E COMO EVITÁ-LOS

Mesmo desenvolvedores experientes cometem erros ao automatizar testes com Cypress. Saber quais são os mais comuns e como evitá-los é essencial para garantir testes mais rápidos, estáveis e confiáveis.

ERROS COMUNS E COMO EVITÁ-LOS

Ao começar a trabalhar com Cypress, é normal encontrar alguns erros que se repetem entre os desenvolvedores. Esses problemas geralmente estão ligados à **sincronia dos comandos**, ao **uso incorreto de seletores**, ou à **falta de boas práticas** na estrutura dos testes. Conhecer essas armadilhas é essencial para criar automações mais estáveis e confiáveis.

🧭 Resumo Rápido

Erro Comum	Consequência	Como Evitar
Esperar tempo fixo (`cy.wait()`)	Testes lentos e instáveis	Use `should()` e `timeout`
Seletores frágeis	Testes quebram após mudanças na UI	Use `data-cy` ou `data-testid`
Testes dependentes	Erros em cadeia	Use `beforeEach()` e isole cenários
Não validar respostas	Falhas passam despercebidas	Sempre use `expect()`
Misturar UI e API	Testes confusos	Separe responsabilidades
Ignorar logs e prints	Dificulta debug	Use `screenshots` e `videos`
Código repetido	Testes longos e confusos	Use comandos personalizados

💬 Termos de Erro Comuns

- **Timeout** — O elemento demorou demais para aparecer.
- **Element not found** — Seletor incorreto ou elemento não existe.
- **Cross-origin error** — Tentativa de acessar site diferente da base URL.
- **Assertion failed** — A validação (expect) não passou.

⚙️ Dica Final:

Testes estáveis vêm da **clareza e consistência** — se cada teste tem um propósito único e usa boas práticas, você evita 90% dos problemas mais comuns.

08

GLOSSÁRIO RÁPIDO

Um guia rápido para fixar os principais conceitos

GLOSSÁRIO RÁPIDO

📖 **Glossário Cypress e Automação**

Termo	Significado
Cypress	Framework de automação de testes end-to-end
Gherkin	Linguagem para escrever cenários legíveis
BDD	Behavior-Driven Development (desenvolvimento guiado por comportamento)
Fixture	Arquivo JSON usado para armazenar dados de teste
Page Object	Padrão de organização para páginas e componentes
Selector	Forma de identificar elementos na página
cy.get()	Seleciona um elemento HTML
cy.request()	Faz uma requisição HTTP
beforeEach()	Executa um bloco antes de cada teste
assert / should	Validações (asserções) em Cypress

CONCLUSÃO

CONCLUSÃO

⚙️ **Automação: qualidade em movimento**

A automação de testes é o caminho para garantir **qualidade, velocidade e segurança** no desenvolvimento de software.
Ela reduz erros manuais, acelera entregas e dá mais confiança a cada atualização.
🔁 *Automatizar é transformar tempo gasto em tempo ganho.*

💡 **Cypress: moderno e eficiente**

O **Cypress** simplifica a automação com:
•Execução em tempo real 🧭
•Relatórios visuais 📊
Com ele, os testes se tornam **parte natural do desenvolvimento**, não um obstáculo.

🧩 **BDD e Gherkin: colaboração que gera clareza**

O **BDD** e o **Gherkin** aproximam equipes técnicas e de negócio, descrevendo comportamentos de forma simples e legível.
Assim, todos entendem o que o sistema deve fazer antes mesmo de escrever o código.
💬 *Comportamento claro gera testes melhores.*

OBRIGADO!