PATTERN OBJECT PAGE

como organizar e manter código de testes automatizados

wederson machado

trilhadequalidade.com.br

ÍNDICE

Quem sou eu? — 03
Definições e conceitos básico — 04
Estrutura Básica do Object Page — 07
Componentização e Reutilização — 14
Melhores Práticas — 23
Testes Avançados com Object Page — 28
Conclusão — 37

INÍCIO

QUEM SOU EU?

Olá! Meu nome é Wederson, um carioca apaixonado pela vida, nascido nas ensolaradas terras do Rio de Janeiro em 02 de fevereiro de 1991. Desde cedo, fui cativado pelo mundo da tecnologia, onde encontrei minha paixão e propósito.

Ao longo dos anos, desenvolvi uma afinidade especial com a área técnica, liderando equipes com entusiasmo e comprometimento. São mais de 5 anos de experiência como líder, guiando e inspirando colegas na jornada do sucesso profissional.

Minha trajetória no universo da tecnologia começou aos 19 anos, e desde então, tenho vivido e respirado esse mundo dinâmico e desafiador. Além disso, minha vida é preenchida com amor pelos meus amigos, pelos animais, especialmente pelos cachorros, e pela energia revigorante da praia.
Neste Ebook, compartilho com você minha expertise, na esperança de inspirar e agregar valor ao seu caminho. Juntos, vamos explorar as possibilidades infinitas desse fascinante universo digital. Seja bem-vindo(a) à nossa jornada!

DEFINIÇÃO E CONCEITOS BÁSICOS

O **Object Page Pattern** é uma abordagem de design estrutural usada para **organizar e manter** o código de teste automatizado, especialmente em ambientes de testes de interface de usuário (UI).

Nesse padrão, cada página ou componente significativo da aplicação é representado como uma classe ou objeto separado. Cada uma dessas classes contém métodos que interagem com os elementos da página, encapsulando a lógica de navegação e interação dentro dessas classes.

BENEFÍCIOS DE USAR EM TESTES AUTOMATIZADOS

Organização: Mantém o código de teste limpo e bem estruturado, facilitando a localização e manutenção dos testes.

Reutilização: Permite a reutilização de métodos e componentes entre diferentes testes, reduzindo a duplicação de código.

Manutenção: Facilita a atualização dos testes quando há mudanças na UI, uma vez que as alterações são feitas em um único lugar.

Leitura e Compreensão: Torna os testes mais legíveis e compreensíveis, mesmo para quem não escreveu originalmente o código.

COMPARAÇÃO COM OUTRAS ABORDAGENS DE ESTRUTURAÇÃO DE TESTES

Scripted Tests: Abordagem onde cada teste é escrito como um script linear, que pode se tornar difícil de manter e compreender à medida que cresce.

Modular Tests: Divisão dos testes em módulos menores, mas pode ainda haver repetição de lógica de navegação e interação.

Object Page Pattern: Foco na encapsulação da lógica de UI em classes de página, promovendo a reutilização e manutenção eficiente.

POR QUE USAR O OBJECT PAGE PATTERN NO CYPRESS?

Organização e Manutenção de Código de Teste
Com o Object Page Pattern, o código de teste é organizado de maneira que cada página ou componente da aplicação é representado por uma classe específica. Isso torna mais fácil manter e atualizar os testes quando há mudanças na UI.

Em vez de modificar cada script de teste individualmente, as alterações podem ser feitas na classe correspondente, propagando-se automaticamente para todos os testes que utilizam essa classe.

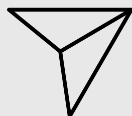
Reutilização de Componentes e Funcionalidades
O Object Page Pattern permite a criação de métodos reutilizáveis para interagir com elementos da UI. Esses métodos podem ser usados em múltiplos testes, reduzindo a duplicação de código e facilitando a implementação de mudanças. Por exemplo, um método que preenche um formulário de login pode ser reutilizado em todos os testes que envolvem autenticação, garantindo consistência e economizando tempo.

Facilidade de Leitura e Compreensão dos Testes
Testes escritos utilizando o Object Page Pattern são mais legíveis e fáceis de entender. A separação clara entre a lógica de navegação/interação e os próprios testes permite que novos membros da equipe ou mesmo outras partes interessadas (como gerentes de produto) compreendam facilmente o que cada teste está fazendo. Isso também facilita a revisão e depuração dos testes.

TÁ CURTINDO?

NÃO DEIXE DE INTERAGIR

CURTA



COMPARTILHE

COMENTE

ESTRUTURA BÁSICA DO OBJECT PAGE

A criação de classes de página é um componente essencial do Object Page Pattern, onde cada classe representa uma página específica ou um componente da aplicação. A estrutura básica de uma classe de página no Cypress inclui métodos para interagir com elementos da página, como campos de entrada, botões e links.

Componentes Essenciais de uma Classe de Página:

- Métodos de Navegação:Métodos que encapsulam a navegação para a página específica, como visit(), que utiliza o comando cy.visit() para acessar a URL da página.
- Seletores de Elementos:Seletores para localizar elementos na página, como campos de entrada, botões e links. Utiliza o comando cy.get() para selecionar esses elementos.
- Métodos de Ação:Métodos que encapsulam ações do usuário, como preencher campos de entrada (type()), clicar em botões (click()), selecionar opções de dropdown (select()) e outras interações.

Estrutura Básica

LoginPage.js

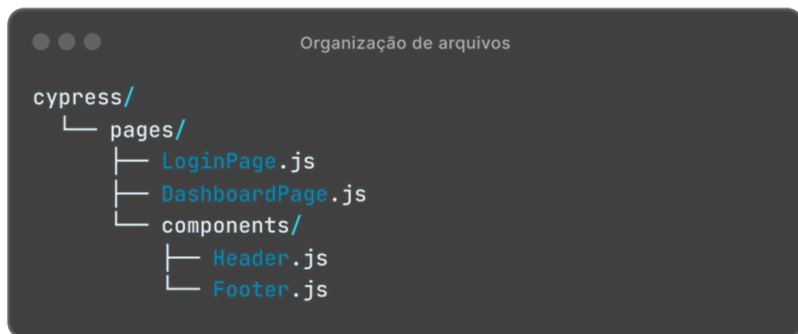
```
// cypress/pages/LoginPage.js
class LoginPage {
    visit() {
        cy.visit('/login');
    }

    fillEmail(email) {
        cy.get('input[name="email"]').type(email);
    }

    fillPassword(password) {
        cy.get('input[name="password"]').type(password);
    }

    submit() {
        cy.get('button[type="submit"]').click();
    }
}

export default new LoginPage();
```

CONVENÇÕES DE NOMENCLATURA E ORGANIZAÇÃO DE ARQUIVOS

Para manter o código organizado e fácil de manter, é importante seguir convenções de nomenclatura e organização de arquivos. Aqui estão algumas práticas recomendadas:

ESTRUTURA BÁSICA DO OBJECT PAGE

1. **Nomenclatura de Classes e Arquivos**:
    - Use nomes descritivos e consistentes para classes e arquivos, refletindo a página ou componente que representam.
    - As classes de página devem ter nomes em CamelCase e serem armazenadas em arquivos com a mesma convenção. Por exemplo, a classe para a página de login deve ser chamada LoginPage e armazenada em LoginPage.js.
2. **Estrutura de Pastas:**
    - Organize as classes de página dentro de uma pasta pages no diretório cypress.
    - Se o projeto for grande e complexo, considere criar subpastas dentro de pages para diferentes seções da aplicação.
3. **Componentização**:
    - Para componentes reutilizáveis, crie uma pasta components dentro da pasta pages e organize os componentes lá.
    - Cada componente deve ser uma classe separada que pode ser importada e utilizada em diferentes páginas.



EXEMPLO DE UMA CLASSE DE PÁGINA SIMPLES

A seguir, um exemplo de uma classe de página simples para a página de login de uma aplicação. Essa classe inclui métodos para navegar até a página, preencher campos de email e senha e clicar no botão de login.

LoginPage.js

```
// cypress/pages/LoginPage.js
class LoginPage {
    visit() {
        cy.visit('/login');
    }

    fillEmail(email) {
        cy.get('input[name="email"]').type(email);
    }

    fillPassword(password) {
        cy.get('input[name="password"]').type(password);
    }

    submit() {
        cy.get('button[type="submit"]').click();
    }
}

export default new LoginPage();
```

Neste exemplo, a classe LoginPage encapsula toda a lógica necessária para interagir com a página de login. Os métodos visit(), fillEmail(), fillPassword() e submit() tornam os testes mais limpos e reutilizáveis,

já que qualquer mudança na UI da página de login pode ser atualizada apenas nesta classe.

Uso da Classe LoginPage em um Teste:

login.spec.js

```
// cypress/integration/login.spec.js
import LoginPage from '../pages/LoginPage';

describe('Login Test', () => {
    it('should log in successfully', () => {
        LoginPage.visit();
        LoginPage.fillEmail('user@example.com');
        LoginPage.fillPassword('password');
        LoginPage.submit();

        // Verificação de sucesso
        cy.url().should('include', '/dashboard');
    });
});
```

Esse exemplo demonstra como utilizar a classe LoginPage em um teste, tornando o teste mais legível e fácil de manter. Ao seguir essas práticas, você garante que seu código de teste seja organizado, reutilizável e fácil de compreender, facilitando a manutenção e a escalabilidade dos testes automatizados.

EXEMPLO DE UMA CLASSE DE PÁGINA SIMPLES

Para utilizar uma classe de página em seus testes, você deve criar uma instância dessa classe e

exportá-la. Isso facilita a reutilização e mantém o código organizado.

Exemplo:

LoginPage.js

```
// cypress/pages/LoginPage.js
class LoginPage {
    visit() {
        cy.visit('/login');
    }

    fillEmail(email) {
        cy.get('input[name="email"]').type(email);
    }

    fillPassword(password) {
        cy.get('input[name="password"]').type(password);
    }

    submit() {
        cy.get('button[type="submit"]').click();
    }
}

export default new LoginPage();
```

Neste exemplo, a classe LoginPage é instanciada e a instância é exportada para uso em outros arquivos.

USO DE UMA INSTÂNCIA DE CLASSE EM TESTES

Depois de criar e exportar a instância da classe de página, você pode importá-la e utilizá-la em seus testes para interagir com a UI da aplicação.

Exemplo:

login.spec.js

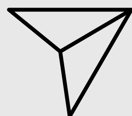
```
// cypress/integration/login.spec.js
import LoginPage from '../pages/LoginPage';

describe('Login Test', () => {
    it('should log in successfully', () => {
        LoginPage.visit();
        LoginPage.fillEmail('user@example.com');
        LoginPage.fillPassword('password');
        LoginPage.submit();

        // Verificação de sucesso
        cy.url().should('include', '/dashboard');
    });
});
```

Neste teste, a instância LoginPage é usada para navegar até a página de login, preencher os campos de email e senha, e submeter o formulário, tornando o teste mais organizado e fácil de ler.

TÁ CURTINDO?

NÃO DEIXE DE INTERAGIR

CURTA



COMPARTILHE

COMENTE

COMPONENTIZAÇÃO E REUTILIZAÇÃO

Componentes reutilizáveis são partes da interface do usuário que podem ser usadas em diferentes páginas ou partes de uma aplicação. No contexto do Object Page Pattern, esses componentes são encapsulados em classes separadas, permitindo sua reutilização em várias páginas de teste. Essa abordagem promove a DRY (Don't Repeat Yourself) e facilita a manutenção do código de teste.

Benefícios dos Componentes Reutilizáveis::

- **Redução da Duplicação**: Elimina a necessidade de duplicar código para componentes comuns em várias páginas.

- **Manutenção Simplificada**: Alterações em um componente afetam todas as páginas que o utilizam, necessitando atualização em apenas um lugar.

- **Código Limpo e Organizado**: Facilita a leitura e compreensão do código, mantendo as interações com a UI bem definidas e separadas.

EXEMPLO DE COMPONENTE REUTILIZÁVEL

Vamos criar um componente de cabeçalho que pode ser utilizado em várias páginas de uma aplicação.

Header.js

```
// cypress/pages/components/Header.js
class Header {
    getLogo() {
        return cy.get('.logo');
    }

    getUserProfile() {
        return cy.get('.user-profile');
    }

    logout() {
        cy.get('.logout-button').click();
    }
}

export default new Header();
```

Neste exemplo, a classe Header encapsula métodos para interagir com elementos do cabeçalho, como o logo, o perfil do usuário e o botão de logout.

Uso do Componente em uma Página:

Vamos utilizar o componente Header na página do dashboard.

DashboardPage.js

```
// cypress/pages/DashboardPage.js
import Header from './components/Header';

class DashboardPage {
    header = Header;

    visit() {
        cy.visit('/dashboard');
    }

    getWelcomeMessage() {
        return cy.get('.welcome-message');
    }
}

export default new DashboardPage();
```

Aqui, a página DashboardPage inclui o componente Header, permitindo o acesso aos métodos definidos na classe Header.

Exemplo de Teste Utilizando o Componente:

Finalmente, vamos criar um teste que interaja com o dashboard e utilize o componente de cabeçalho.

dashboard.spec.js

```
// cypress/integration/dashboard.spec.js
import { LoginPage } from '../pages';
import DashboardPage from '../pages/DashboardPage';

describe('Dashboard Test', () => {
    before(() => {
        LoginPage.visit();
        LoginPage.fillEmail('user@example.com');
        LoginPage.fillPassword('password');
        LoginPage.submit();
    });

    it('should display the welcome message', () => {
        DashboardPage.visit();
        DashboardPage.getWelcomeMessage().should('contain',
'Welcome');
    });

    it('should display the user profile in the header', ()
=> {

DashboardPage.header.getUserProfile().should('be.visible');
    });

    it('should log out successfully', () => {
        DashboardPage.header.logout();
        cy.url().should('include', '/login');
    });
});
```

Neste exemplo, os testes utilizam o componente Header através da página DashboardPage, mostrando a reutilização e organização proporcionadas pelo uso de componentes reutilizáveis. Isso demonstra como encapsular a lógica de UI em componentes específicos pode

simplificar a escrita e manutenção de testes automatizados.

INTEGRAÇÃO DE COMPONENTES EM PÁGINAS

A integração de componentes reutilizáveis dentro de classes de página é um aspecto crucial do Object Page Pattern. Esta prática permite uma modularização eficiente, onde cada página da aplicação pode usar componentes comuns sem duplicação de código. Essa abordagem melhora a manutenção, legibilidade e reutilização do código de teste.

Passos para Integrar Componentes em Páginas:

- **Definir e Exportar o Componente Reutilizável**:
  - Crie classes para componentes reutilizáveis, encapsulando a lógica de interação com elementos específicos da UI.
  - Exporte as instâncias dessas classes para que possam ser usadas em outras partes do código.

- **Importar o Componente na Classe de Página**:
  - Importe o componente reutilizável na classe de página onde será utilizado.
  - Crie uma instância do componente ou referencie diretamente a instância exportada.

- **Utilizar o Componente na Classe de Página:**
  - Utilize os métodos e propriedades do componente dentro dos métodos da classe de página para realizar ações específicas.

Exemplo: Integração de um Componente de Cabeçalho (Header) em uma Página

<u>Passo 1</u>: Definir e Exportar o Componente Reutilizável

Header.js

```
// cypress/pages/components/Header.js
class Header {
    getLogo() {
        return cy.get('.logo');
    }

    getUserProfile() {
        return cy.get('.user-profile');
    }

    logout() {
        cy.get('.logout-button').click();
    }
}

export default new Header();
```

<u>Passo 2</u>: Importar o Componente na Classe de Página

DashboardPage.js

```
// cypress/pages/DashboardPage.js
import Header from './components/Header';

class DashboardPage {
    header = Header;

    visit() {
        cy.visit('/dashboard');
    }

    getWelcomeMessage() {
        return cy.get('.welcome-message');
    }
}

export default new DashboardPage();
```

Neste exemplo, a classe DashboardPage importa o componente Header e o associa a uma propriedade chamada header.

Passo 3: Utilizar o Componente na Classe de Página

dashboard.spec.js

```
// cypress/integration/dashboard.spec.js
import { LoginPage } from '../pages';
import DashboardPage from '../pages/DashboardPage';

describe('Dashboard Test', () => {
    before(() => {
        LoginPage.visit();
        LoginPage.fillEmail('user@example.com');
        LoginPage.fillPassword('password');
        LoginPage.submit();
    });

    it('should display the welcome message', () => {
        DashboardPage.visit();
        DashboardPage.getWelcomeMessage().should('contain',
'Welcome');
    });

    it('should display the user profile in the header', ()
=> {

DashboardPage.header.getUserProfile().should('be.visible');
    });

    it('should log out successfully', () => {
        DashboardPage.header.logout();
        cy.url().should('include', '/login');
    });
});
```

Neste teste, utilizamos o componente Header através da propriedade header da classe DashboardPage. Isso permite:

- **Reutilização de Métodos**: Métodos como getUserProfile() e logout() são definidos uma vez no componente Header e reutilizados em qualquer página que importe esse componente.

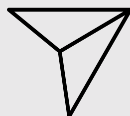
- **Organização**: A lógica de interação com o cabeçalho está centralizada no componente Header, facilitando a manutenção e a atualização do código.

Integrar componentes reutilizáveis em classes de página é uma prática poderosa que promove a organização e a manutenção do código de teste. Seguindo este padrão, podemos garantir que as interações com a interface do usuário sejam encapsuladas de forma modular e reutilizável, resultando em testes mais limpos e fáceis de manter.

TÁ CURTINDO?

NÃO DEIXE DE INTERAGIR

CURTA



COMPARTILHE

COMENTE

1. MANUTENÇÃO E ORGANIZAÇÃO

Estruturação de Arquivos e Pastas para Facilitar a Manutenção

Uma estrutura de arquivos e pastas bem organizada é fundamental para manter a clareza e a eficiência do projeto. No contexto do Object Page Pattern, é recomendável seguir uma estrutura hierárquica e intuitiva que reflita a organização lógica do aplicativo. Uma possível estrutura pode ser:

Estrutura de Arquivos

```
cypress/
  └── integration/
        ├── login.spec.js
        ├── dashboard.spec.js
  └── pages/
        ├── LoginPage.js
        ├── DashboardPage.js
        └── components/
              ├── Header.js
              └── Footer.js
  └── support/
        └── commands.js
        └── index.js
```

Boas Práticas para Nomear e Organizar Classes de Página e Componentes:

- **Nomenclatura Descritiva**: Utilize nomes claros e descritivos para classes de página e componentes, refletindo sua funcionalidade. Por exemplo, LoginPage para a página de login e Header para o componente de cabeçalho.

- **Consistência**: Mantenha uma convenção de nomenclatura consistente, como CamelCase para classes (LoginPage) e kebab-case para arquivos (login-page.js).

- **Segregação de Responsabilidades:** Separe componentes e páginas em arquivos distintos para promover a modularidade e facilitar a manutenção.

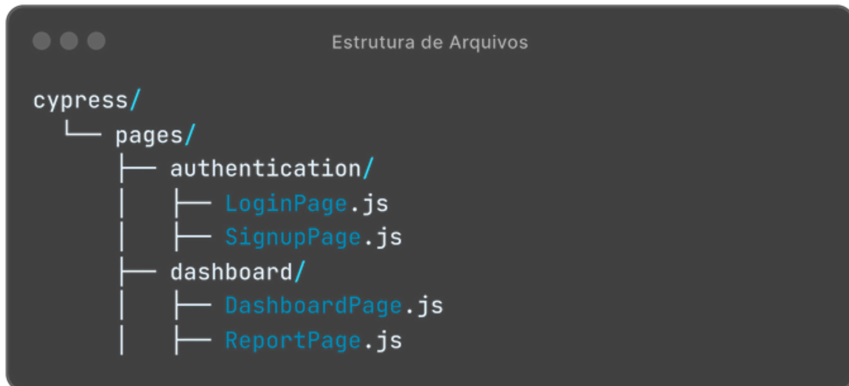
2. ESCALABILIDADE

Como Escalar a Estrutura de Object Page para Projetos Maiores

À medida que o projeto cresce, é crucial que a estrutura do Object Page Pattern seja escalável. Para isso, adote as seguintes estratégias:

- **Modularização**: Divida a aplicação em módulos menores e mais gerenciáveis, como diferentes áreas funcionais (autenticação, dashboard, relatórios).

- **Subpastas:** Use subpastas dentro da pasta pages para organizar páginas relacionadas. Por exemplo:




```
cypress/
  └── pages/
      ├── authentication/
      │   ├── LoginPage.js
      │   ├── SignupPage.js
      ├── dashboard/
      │   ├── DashboardPage.js
      │   ├── ReportPage.js
```

Estratégias para Manter a Coesão e Reduzir a Complexidade

- **Componentização:** Encapsule funcionalidades comuns em componentes reutilizáveis para evitar a duplicação de código e melhorar a coesão.

- **Funções Utilitárias:** Crie funções utilitárias e helpers para ações repetitivas, facilitando a reutilização e a manutenção.

3. REUTILIZAÇÃO E DRY (DON'T REPEAT YOURSELF)

Maximizar a Reutilização de Componentes

- **Componentes Genéricos:** Desenvolva componentes genéricos que podem ser usados em várias partes da aplicação. Por exemplo, um componente de Button que pode ser estilizado e configurado conforme necessário.

- **Biblioteca de Componentes**: Mantenha uma biblioteca de componentes reutilizáveis dentro do projeto, documentando suas funcionalidades e modos de uso.

Evitar Duplicação de Código com Funções Utilitárias e Helpers

- **Utilitários Comuns**: Crie funções utilitárias para operações comuns, como formatação de datas, geração de IDs únicos, ou manipulação de strings.

- **Helpers de Teste**: Desenvolva helpers de teste para encapsular interações complexas ou repetitivas com a UI. Por exemplo, um helper para preencher e submeter formulários.

Exemplo de Função Utilitária:

Função utilitária

```
// cypress/support/utils.js
export function formatDate(date) {
    const options = { year: 'numeric', month: '2-digit',
day: '2-digit' };
    return new Date(date).toLocaleDateString('en-US',
options);
}
```

Uso de Helpers em Testes:

Helpers

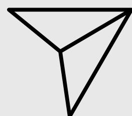
```
// cypress/support/commands.js
Cypress.Commands.add('login', (email, password) => {
    cy.visit('/login');
    cy.get('input[name="email"]').type(email);
    cy.get('input[name="password"]').type(password);
    cy.get('button[type="submit"]').click();
});
```

Integrando essas melhores práticas em seu projeto de testes automatizados, você garante um código mais organizado, eficiente e fácil de manter, facilitando a escalabilidade e a reutilização ao longo do tempo.

TÁ CURTINDO?

NÃO DEIXE DE INTERAGIR

CURTA



COMPARTILHE

COMENTE

trilhadequalidade.com.br

TESTES AVANÇADOS COM OBJECT PAGE

Em aplicações reais, muitos cenários de teste envolvem a navegação e interação entre várias páginas. Utilizando o Object Page Pattern, é possível estruturar esses testes de maneira organizada e eficiente, garantindo que cada página e seus componentes sejam reutilizáveis e mantidos de forma coesa.

Benefícios de Utilizar o Object Page Pattern para Testes Multi-Página:

1. **Reutilização de Componentes**: Permite a reutilização de componentes e métodos comuns em várias páginas, como formulários de login, barras de navegação e rodapés.
2. **Facilidade de Manutenção**: Centraliza a lógica de cada página em suas respectivas classes, facilitando a atualização e a manutenção do código.
3. **Organização e Clareza**: Mantém o código dos testes organizado, tornando os testes mais fáceis de ler e entender.

ESTRUTURA BÁSICA PARA TESTES MULTI-PÁGINA

Para ilustrar como realizar testes avançados que envolvem navegação e interação entre várias páginas, vamos considerar um cenário simples onde um usuário faz login, navega até o dashboard, e então acessa uma página de relatórios.

Passo 1: Definir Classes de Página

LoginPage.js

```
// cypress/pages/LoginPage.js
class LoginPage {
    visit() {
        cy.visit('/login');
    }

    fillEmail(email) {
        cy.get('input[name="email"]').type(email);
    }

    fillPassword(password) {
        cy.get('input[name="password"]').type(password);
    }

    submit() {
        cy.get('button[type="submit"]').click();
    }
}

export default new LoginPage();
```

DashboardPage.js

```
// cypress/pages/DashboardPage.js
class DashboardPage {
    visit() {
        cy.visit('/dashboard');
    }

    goToReports() {
        cy.get('a[href="/reports"]').click();
    }

    getWelcomeMessage() {
        return cy.get('.welcome-message');
    }
}

export default new DashboardPage();
```

ReportPage.js

```
// cypress/pages/ReportPage.js
class ReportPage {
    visit() {
        cy.visit('/reports');
    }

    getReportTitle() {
        return cy.get('h1');
    }
}

export default new ReportPage();
```

Passo 2: Criar Testes que Interagem Entre Múltiplas Páginas

multiPage.spec.js

```
// cypress/integration/multiPage.spec.js
import LoginPage from '../pages/LoginPage';
import DashboardPage from '../pages/DashboardPage';
import ReportPage from '../pages/ReportPage';

describe('Multi-Page Interaction Test', () => {
    it('should log in, navigate to dashboard, and then to
reports page', () => {
        // Step 1: Realiza login
        LoginPage.visit();
        LoginPage.fillEmail('user@example.com');
        LoginPage.fillPassword('password');
        LoginPage.submit();

        // Step 2: Verifica navegação para Dashboard
        DashboardPage.visit();
        DashboardPage.getWelcomeMessage()
                    .should('contain', 'Welcome');

        // Step 3: Verifica navegação para Report
        DashboardPage.goToReports();

        // Step 4: Verifica página de Report
        ReportPage.getReportTitle()
                  .should('contain', 'Reports');
    });
});
```

Neste exemplo, o teste cobre o fluxo completo desde o login até a navegação entre páginas e a verificação de elementos específicos em cada página.

Principais Elementos do Teste Multi-Página:

1. **Login**: Usa métodos definidos na classe LoginPage para realizar ações de login.
2. **Dashboard:** Navega para o dashboard e verifica a presença de uma mensagem de boas-vindas.
3. **Relatórios**: Navega para a página de relatórios e verifica o título da página.

Utilizando o Object Page Pattern para testes que envolvem navegação e interação entre múltiplas páginas, é possível estruturar testes complexos de maneira organizada e eficiente. Esta abordagem promove a reutilização de componentes, facilita a manutenção e melhora a clareza dos testes, permitindo que você construa uma suíte de testes robusta e sustentável para aplicações de qualquer tamanho.

TESTES DE INTEGRAÇÃO COMPLEXOS

Os testes de integração complexos são essenciais para garantir que diferentes partes de uma aplicação funcionem corretamente juntas. Esses testes envolvem a criação de cenários que simulam o uso real da aplicação, com várias etapas e verificações ao longo do processo. Utilizando o Object Page Pattern, esses cenários podem ser estruturados de forma modular e reutilizável.

Criação de Cenários de Teste Mais Complexos Envolvendo Várias Etapas e Verificações

Para criar cenários de teste complexos, siga estas etapas:

1. **Defina o Fluxo de Trabalho**: Mapeie todas as etapas do fluxo de trabalho que o teste deve cobrir, desde o início até a conclusão.
2. **Utilize Classes de Página**: Utilize classes de página para encapsular a lógica de cada etapa. Isso facilita a manutenção e a leitura do teste.
3. **Combine Ações e Verificações**: Combine ações de interação com verificações de estado em cada etapa do fluxo.

Exemplo: Cenário de Teste Complexo

Vamos criar um cenário onde um usuário faz login, adiciona um item ao carrinho, realiza o checkout e verifica a confirmação do pedido.

CartPage.js

```
// cypress/pages/CartPage.js
class CartPage {
    visit() {
        cy.visit('/cart');
    }

    addItemToCart(item) {
        cy.get(`[data-test="${item}"]`).click();
    }

    goToCheckout() {
        cy.get('button.checkout').click();
    }
}

export default new CartPage();
```

CheckoutPage.js

```
// cypress/pages/CheckoutPage.js
class CheckoutPage {
    fillShippingDetails(details) {

cy.get('input[name="address"]').type(details.address);
        cy.get('input[name="city"]').type(details.city);

cy.get('input[name="postalCode"]').type(details.postalCode);
    }

    placeOrder() {
        cy.get('button.place-order').click();
    }
}

export default new CheckoutPage();
```

OrderConfirmationPage.js

```
// cypress/pages/OrderConfirmationPage.js
class OrderConfirmationPage {
    getOrderConfirmationMessage() {
        return cy.get('.order-confirmation');
    }
}

export default new OrderConfirmationPage();
```

Agora que já criamos as nossas classes principais, iremos importá-las no nosso teste de integração complexo para que elas nos auxiliem.

(veja na próxima página)

complexIntegration.spec.js

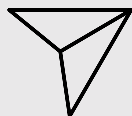
```
// cypress/integration/complexIntegration.spec.js
import LoginPage from '../pages/LoginPage';
import CartPage from '../pages/CartPage';
import CheckoutPage from '../pages/CheckoutPage';
import OrderConfirmationPage from
'../pages/OrderConfirmationPage';

describe('Complex Integration Test', () => {
    it('should complete a purchase successfully', () => {
        // Step 1: Realiza Login
        LoginPage.visit();
        LoginPage.fillEmail('user@example.com');
        LoginPage.fillPassword('password');
        LoginPage.submit();

        // Step 2: Adiciona item ao carrinho
        CartPage.visit();
        CartPage.addItemToCart('item1');
        CartPage.goToCheckout();

        // Step 3: Preencha os detalhes de envio e faça o pedido
        CheckoutPage.fillShippingDetails({
            address: '123 Main St',
            city: 'Springfield',
            postalCode: '12345'
        });
        CheckoutPage.placeOrder();

        // Step 4: Verifica confirmação do pedido
        OrderConfirmationPage.getOrderConfirmationMessage()
        .should('contain', 'Pedido confirmado!');
    });
});
```

Neste exemplo, o teste cobre um fluxo de trabalho complexo que inclui login, adição de item ao carrinho, checkout e verificação de confirmação de pedido.

TÁ CURTINDO?

NÃO DEIXE DE INTERAGIR

CURTA



COMPARTILHE

COMENTE

trilhadequalidade.com.br

CONCLUSÃO

RECAPITULAÇÃO DOS PRINCIPAIS PONTOS ABORDADOS

Neste ebook, exploramos diversas facetas do uso do Cypress com o Object Page Pattern, desde os conceitos básicos até práticas avançadas para a criação de testes robustos e eficientes. Recapitulando os principais pontos:

- **Definição e Conceitos Básicos**: Introduzimos o Object Page Pattern, destacando sua importância para a organização e manutenção de testes automatizados.

- **Benefícios**: Discutimos como essa abordagem facilita a reutilização de componentes, melhora a clareza do código e contribui para a escalabilidade dos testes.

- **Criação de Classes de Página**: Demonstramos como estruturar e implementar classes de página, seguindo convenções de nomenclatura e organização de arquivos.

- **Componentes Reutilizáveis**: Exploramos a definição e o uso de componentes reutilizáveis para evitar duplicação de código e maximizar a eficiência.

- **Interação Entre Múltiplas Páginas**: Abordamos a criação de testes complexos que envolvem navegação e interação entre várias páginas, mantendo a coesão e a clareza.

- **Testes de Integração Complexos**: Mostramos como criar cenários de teste complexos que cobrem fluxos completos de trabalho, garantindo que todas as partes da aplicação funcionem harmoniosamente.

PRÓXIMOS PASSOS

Para continuar aprimorando suas habilidades e a eficácia de seus testes automatizados, considere as seguintes práticas avançadas:

- **Refatoração Contínua**: Regularmente revise e refatore seus testes para incorporar novas melhores práticas, manter o código limpo e eliminar duplicações desnecessárias.

- **Cobertura de Testes**: Aumente a cobertura dos testes, garantindo que todos os cenários críticos e fluxos de usuário sejam validados, incluindo casos extremos e de borda.

- **Integração Contínua**: Integre seus testes automatizados em pipelines de CI/CD para que eles sejam executados automaticamente em cada commit, detectando problemas de forma precoce.

- **Performance de Testes**: Monitore e otimize a performance dos testes automatizados, minimizando o tempo de execução sem comprometer a cobertura e a confiabilidade.

- **Testes de Segurança**: Adicione testes de segurança para verificar vulnerabilidades comuns e garantir que a aplicação esteja protegida contra ataques conhecidos.

- **Documentação e Treinamento**: Documente suas estratégias de teste e pratique o treinamento contínuo da equipe, promovendo a adoção de boas práticas e o compartilhamento de conhecimento.

Adotando essas práticas avançadas, você não apenas aprimora a qualidade dos seus testes automatizados, mas também contribui para a construção de um ciclo de desenvolvimento de software mais robusto, eficiente e sustentável.

OBRIGADO!

ME SIGA NAS REDES SOCIAIS

WEDERSON-MACHADO

@TRILHADEQUALIDADE

TRILHADEQUALIDADE.COM.BR