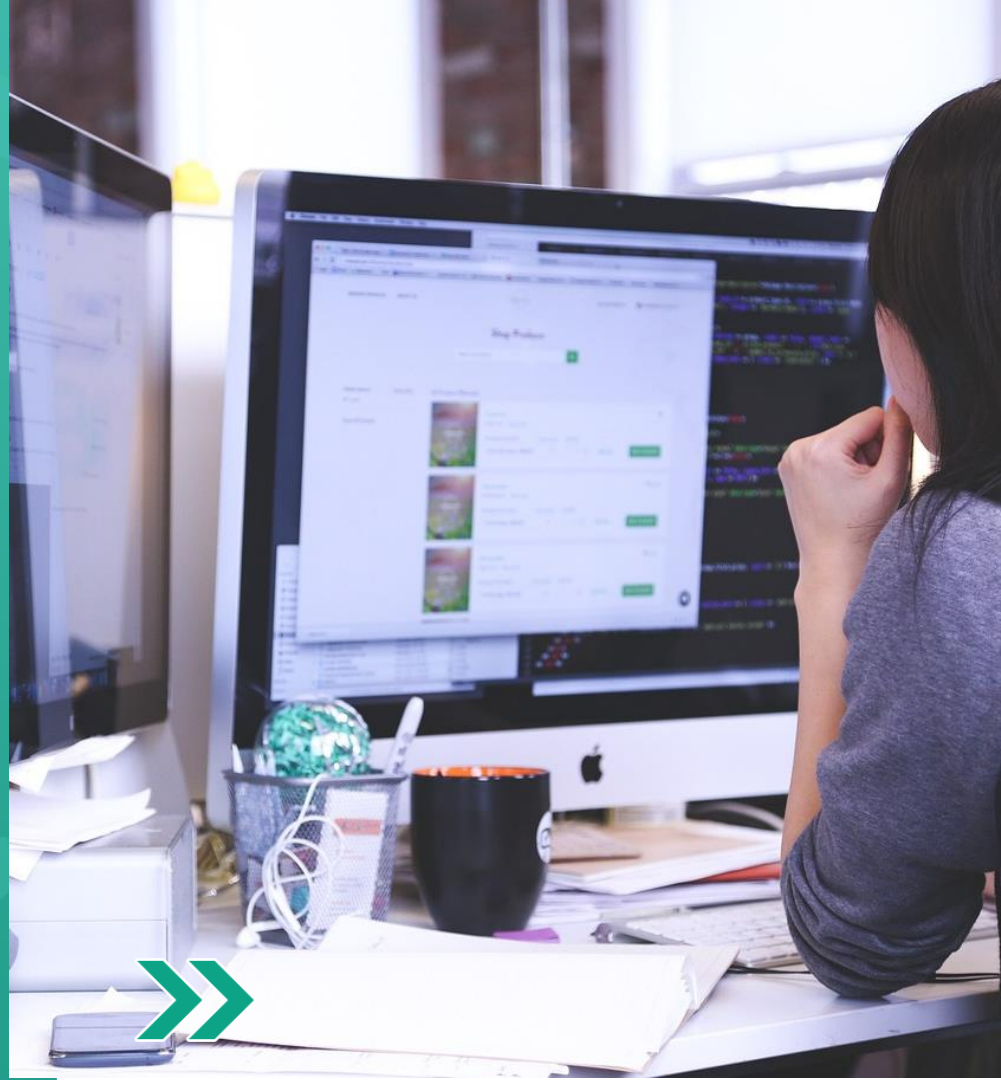
escola britânica de artes criativas & tecnologia

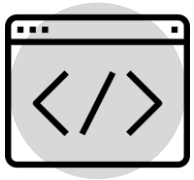
# Profissão: Engenheiro Front-End

escola britânica de artes criativas & tecnologia

# BOAS PRÁTICAS



# React Testing Library



Confira boas práticas da comunidade de Front-End por assunto relacionado às aulas.

- **Conheça os tipos de testes**
- **Escreva os primeiros testes**
- **Explore testes no React**
- **Faça testes com React e Redux**
- **Conheça o Mock Service Worker**

# Conheça os tipos de testes

Acompanhe alguns benefícios ao realizar testes.

- **Garantia de qualidade do software:**
  Testes bem escritos garantem que o código funcione corretamente e conforme o esperado. Isso ajuda a evitar bugs e problemas em produção, melhorando a qualidade do software.

- **Detecção precoce de problemas:**
  Testes permitem detectar problemas e erros no código de forma antecipada, antes que o software seja entregue aos usuários ou integrado a outras partes do sistema.

- **Facilita a refatoração:**
  Com testes em vigor, é mais seguro realizar refatorações no código, pois você pode executar os testes para verificar se as alterações não quebraram funcionalidades existentes.

- **Mantém a confiança no código:**
  Testes automatizados fornecem uma base sólida para ter confiança no código, pois você pode executá-los sempre que houver mudanças no projeto, garantindo que tudo continue funcionando conforme o esperado.

# Conheça os tipos de testes

Acompanhe alguns benefícios ao realizar testes.

- **Documentação executável:**
  Testes servem como uma forma de documentação executável, descrevendo como o código deve se comportar em diferentes cenários. Isso facilita a compreensão do código por outros desenvolvedores.

- **Aumenta a produtividade:**
  Embora escrever testes possa parecer um esforço adicional, no longo prazo, eles podem aumentar a produtividade da equipe, uma vez que os bugs são identificados mais cedo e a manutenção do código é facilitada.

- **Reduz custo e tempo de desenvolvimento:**
  Testes bem escritos ajudam a evitar a necessidade de depurar e corrigir problemas em produção, o que pode economizar tempo e dinheiro.

- **Suporte a metodologias ágeis e devOps:**
  Testes automatizados são uma parte fundamental das práticas ágeis e DevOps, permitindo entregas mais frequentes e confiáveis do software.

# Conheça os tipos de testes

Acompanhe alguns benefícios ao realizar testes.

- **Ajuda na tomada de decisões:**
  Testes podem ajudar a equipe a tomar decisões informadas sobre o software e a identificar gargalos ou problemas no código.

- **Melhoria contínua:**
  Testes promovem uma cultura de melhoria contínua, pois os testes falhando incentivam a equipe a resolver problemas e escrever um código mais robusto.

# Escreva os primeiros testes

Acompanhe algumas dicas sobre o uso da função "expect" no Jest.

**Escreva expectativas descritivas:**
Dê nomes descritivos para suas expectativas (assertions) para que, em caso de falha, seja fácil identificar o que estava sendo testado.

**Negar Expectativas:**
Se necessário, você pode negar uma expectativa usando not para verificar que um resultado não é igual ao esperado.

**Use as Funções Assíncronas do Jest:**
O Jest fornece funções de teste específicas para cenários assíncronos, como test, beforeEach, afterEach, beforeAll, afterAll. Utilize-os para lidar com tarefas assíncronas e garantir que os testes sejam executados adequadamente.

**Mensagens em Falhas de Teste:**
Quando possível, forneça uma mensagem personalizada para a expectativa (combinando com o nome do teste) para que, em caso de falha, você saiba o que estava sendo testado e qual era a expectativa.

# Escreva os primeiros testes

Acompanhe algumas dicas sobre o uso do *plugin* de teste no Jest.

- **Escolha os testes certos:**
  Nem todos os pedaços de código precisam ser testados. Escolha testar os que são mais importantes para o seu projeto e mais propensos a ter erros.
- **Escreva testes claros e concisos:**
  Use nomes de testes significativos e escreva etapas de teste claras e concisas.
- **Organize seus testes em arquivos e pastas:**
  Isso tornará mais fácil navegar por eles e encontrar os testes que você precisa.
- **Execute seus testes regularmente:**
  Faça isso como parte do seu processo de desenvolvimento. Isso ajudará a identificar e corrigir erros no código antes que ele seja liberado para produção.
- **Acompanhe o status de seus testes.**
  Isso ajudará a se manter atualizado sobre o progresso do seu projeto e a identificar quaisquer testes que precisem ser corrigidos.

# Escreva os primeiros testes

Acompanhe algumas dicas sobre o uso do *matchers* no Jest.

- **toBe vs. toEqual:**
  Use toBe para testar igualdade estrita (===) de valores primitivos, como números e strings. Use toEqual para testar igualdade de valores de objetos e arrays.

- **toBeNull vs. toBeUndefined vs. toBeDefined:**
  Use toBeNull para verificar se um valor é nulo, toBeUndefined para verificar se um valor é undefined e toBeDefined para verificar se um valor está definido (ou seja, não é undefined).

- **Truthy e Falsy:**
  Use toBeTruthy para verificar se um valor é avaliado como verdadeiro e toBeFalsy para verificar se um valor é avaliado como falso (por exemplo, null, undefined, 0, false ou uma string vazia).

# Escreva os primeiros testes

Acompanhe algumas dicas sobre o uso do *matchers* no Jest.

- **toContain vs. toHaveBeenCalledWith:**
  Use toContain para verificar se um array ou uma string contém um valor específico. Use toHaveBeenCalledWith para verificar se uma função foi chamada com determinados argumentos.

- **toHaveAttribute vs. toHaveStyle:**
  Use toHaveAttribute para verificar se um elemento possui um determinado atributo com um valor específico. Use toHaveStyle para verificar o estilo CSS de um elemento.

# Explore testes no React

**Render:**
O método "render" normalmente é utilizado em conjunto com outros métodos e funções da Testing Library, como "screen.getByText", "screen.getByRole", "screen.getByTestId", "fireEvent", entre outros. Essas funções são usadas para acessar os elementos renderizados do componente e simular interações do usuário, como cliques, inserções de texto e outros eventos.

# Explore testes no React

Acompanhe alguns exemplos de funções disponíveis em "screen".

- **screen.getByText:**
  Seleciona um elemento com base em seu texto.
- **screen.getByRole:**
  Seleciona um elemento com base no papel/propósito que ele desempenha (por exemplo, botão, caixa de diálogo, *link* etc.).
- **screen.getByTestId:**
  Seleciona um elemento com base no atributo "data-testid" definido no elemento.
- **screen.getByPlaceholderText:**
  Seleciona um elemento de entrada (input) com base em seu texto de placeholder.
- **screen.getByLabelText:**
  Seleciona um elemento com base no texto associado à label do elemento.

# Explore testes no React

Existem diferentes tipos de cobertura de testes. Acompanhe a seguir, algumas das mais comuns.

- **Cobertura de linhas (Line coverage):**
  Mede a porcentagem de linhas de código que são executadas pelos testes. Uma linha de código é considerada coberta se foi executada pelo menos uma vez durante a execução dos testes.

- **Cobertura de ramificações (Branch coverage):**
  Mede a porcentagem de caminhos de execução do código que foram percorridos pelos testes. Isso inclui testar as duas direções em instruções condicionais (if/else, switch) e loops.

- **Cobertura de funções (Function coverage):**
  Mede a porcentagem de funções do código que foram chamadas durante a execução dos testes.

- **Cobertura de instruções (Statement coverage):**
  Mede a porcentagem de instruções individuais que foram executadas pelos testes.

# Faça testes com React e Redux

Acompanhe algumas dicas sobre o uso da função getByText.

- **Seja específico com o texto:**
  Ao usar getByText, tente fornecer o texto exato que você espera encontrar no elemento. Se o texto for ambíguo ou puder ser encontrado em vários elementos, o getByText retornará o primeiro elemento correspondente encontrado, o que pode levar a resultados inesperados.

- **Evite testar a aparência:**
  O getByText é mais eficaz quando usado para testar o conteúdo e o comportamento funcional dos elementos, em vez de sua aparência específica. Focar em testes centrados no comportamento do usuário é uma prática recomendada.

- **Use screen como prefixo:**
  Para melhorar a legibilidade e evitar conflitos de nome, é recomendado usar screen.getByText em vez de apenas getByText. Isso ajuda a identificar claramente que a função está vindo da Testing Library.

# Faça testes com React e Redux

Acompanhe algumas dicas sobre o uso da função getByText.

- **Combine com outras funções de seleção:**
  getByText é apenas uma das muitas funções de seleção fornecidas pela Testing Library. Para cenários mais complexos, você pode combinar getByText com outras funções, como getByRole, getByTestId, getByPlaceholderText, entre outras.

- **Lide com elementos assíncronos:**
  Se o conteúdo que você está buscando é carregado de forma assíncrona (por exemplo, após uma chamada de API), você pode precisar usar opções assíncronas, como findByText, para aguardar a renderização dos elementos.

- **Use expectativas e asserts:**
  Após selecionar o elemento usando getByText, você pode realizar expectativas e asserts para verificar se o elemento foi encontrado corretamente e para testar seu comportamento.

# Faça testes com React e Redux

Acompanhe algumas dicas sobre o uso da função getByText.

- **Seja cuidadoso com traduções:**
  Se a aplicação é traduzida para vários idiomas, tenha em mente que o texto visível nos elementos pode variar de acordo com o idioma. Nesses casos, você pode usar chaves de localização ou atributos data-testid para melhorar a seleção de elementos.

- **Prefira seletores semânticos:**
  Se possível, dê preferência para seletores semânticos como data-testid, role, aria-label, entre outros, em vez de depender exclusivamente do texto visível para selecionar elementos. Isso torna os testes menos frágeis em relação a alterações na interface.

# Faça testes com React e Redux

Acompanhe agora algumas dicas sobre o uso do preloadedState.

- **Defina o estado inicial adequadamente:**
  O preloadedState deve conter o estado inicial da aplicação com valores corretos e válidos para que a aplicação comece em um estado consistente. Certifique-se de que o preloadedState esteja estruturado de acordo com o formato esperado pelos redutores da sua aplicação.

- **Carregue dados iniciais de forma síncrona:**
  O preloadedState é útil para fornecer dados iniciais à aplicação antes que ela inicie. No entanto, ele é adequado para carregar dados de forma síncrona, como informações estáticas, configurações ou dados locais. Evite fazer chamadas assíncronas dentro do preloadedState, pois isso pode levar a atrasos na inicialização da aplicação.

# Faça testes com React e Redux

Acompanhe agora algumas dicas sobre o uso do preloadedState.

- **Não utilize informações sensíveis no preloadedState:** Evite armazenar informações confidenciais ou dados sensíveis no preloadedState, pois eles serão enviados ao cliente junto com o código da aplicação. Isso pode expor informações que não devem ser acessadas pelo usuário final.

- **Use selectors para acessar o preloadedState:** Ao acessar o estado dentro dos componentes, prefira utilizar os seletores (selectors) do Redux para obter os dados necessários. Isso garante que você está acessando o estado de forma segura e centralizada, mesmo se o formato do preloadedState mudar no futuro.

# Conheça o Mock Service Worker

Essa técnica permite isolar o código que está sendo testado de suas dependências externas, criando um ambiente controlado e previsível para os testes. Acompanhe algumas dicas sobre "mockar".

- **Mock apenas o necessário:**
  Não é necessário "mockar" todos os componentes ou dependências em seus testes. Concentre-se nas partes do código que precisam ser isoladas para obter uma cobertura eficaz dos testes.

- **Seja seletivo com os mocks:**
  Ao "mockar" uma dependência, forneça apenas as funcionalidades necessárias para o teste específico. Evite criar "mocks" excessivamente complexos ou que não sejam relevantes para o cenário de teste.

- **Utilize bibliotecas de "mocking":**
  Em vez de criar "mocks" personalizados manualmente, use bibliotecas de "mocking" disponíveis para sua linguagem de programação, como Jest no JavaScript ou Mockito no Java. Essas bibliotecas podem simplificar o processo de criação de "mocks" e fornecer funcionalidades avançadas.

# Conheça o Mock Service Worker

Essa técnica permite isolar o código que está sendo testado de suas dependências externas, criando um ambiente controlado e previsível para os testes. Acompanhe algumas dicas sobre "mockar".

- **Evite "mockar" tudo em testes de integração:**
  Em testes de integração, é preferível usar componentes e dependências reais em vez de "mocks". Dessa forma, você pode garantir que o código integrado esteja funcionando corretamente em conjunto.

- **Atualize os "mocks" conforme o código muda:**
  À medida que o código evolui, verifique se os "mocks" estão atualizados para refletir as mudanças no comportamento real das dependências.

# Conheça o Mock Service Worker

Essa biblioteca é destinada a ambientes de desenvolvimento e teste. Em produção, sua aplicação provavelmente usará um servidor Back-End real para lidar com as solicitações de rede. Acompanhe algumas das principais características e benefícios do Mock Service Worker.

- **Simulação realista:**
  Você pode definir as respostas simuladas usando dados reais ou cenários específicos, garantindo que suas simulações sejam o mais realistas possível.
- **Fácil de configurar:**
  O MSW é fácil de configurar e pode ser usado com diferentes bibliotecas e frameworks front-end, como React, Angular, Vue etc.
- **Independente de ambiente:**
  Funciona tanto em ambientes de desenvolvimento local quanto em ambientes de teste automatizados.
- **Flexibilidade:**
  Permite lidar com situações complexas, como autenticação, erros de rede e respostas condicionais.
- **Evita chamadas reais:**
  Ao usar o MSW, você evita chamadas de rede reais durante o desenvolvimento, o que pode economizar tempo e recursos.

escola britânica de artes criativas & tecnologia

# Bons estudos!