RESUMO DA AULA

Porque você deveria testar seu código

Nesta se o, vamos explorar a import ncia de aplicar testes automatizados mesmo para c digos aparentemente simples e que parecem n o necessitar de cobertura de testes. comum, especialmente para desenvolvedores iniciantes, achar que certos trechos de c digo s o t o bvios que n o precisam ser testados. No entanto, veremos como um c digo pode escalar em complexidade e como a falta de testes desde o in cio pode gerar grandes dificuldades no futuro. Exemplo pr tico - Lista de compras Para ilustrar a import ncia dos testes, vamos usar um exemplo pr tico de uma lista de compras que representa um pedido de itens em uma loja. Vamos analisar como um simples c lculo de valor total pode, rapidamente, se tornar mais complexo. Passo 1: Criando a lista de compras e a fun o de c lculo inicial No exemplo inicial, temos um array de itens que representa os produtos do nosso pedido: const meuPedido = { itens: [{ nome: 'Po o de energia', valor: 100 }, { nome: 'Espada longa', valor: 3000 },], }; A fun o para calcular o valor total dos itens no pedido bastante simples: const calcularValorPedido = (pedido) => pedido.itens.reduce((anterior, atual) => anterior + atual.valor, 0); const resultado = calcularValorPedido(meuPedido); console.log(resultado); Esse c digo direto e f cil de entender. Ao ser executado, ele soma os valores de cada item no array meuPedido.itens e retorna o total, que exibido no console. Neste ponto, voc pode pensar: "Essa fun o t o simples, eu realmente preciso escrever testes para isso?" A resposta sim. Apesar de parecer bvia agora, qualquer altera o futura pode adicionar mais complexidade, e a fun o pode crescer rapidamente. Vamos ver isso na pr tica. Passo 2: Adicionando complexidade ao c digo Agora, imagine que recebemos um novo requisito: se o valor total dos produtos for maior que 500 e o pedido incluir a taxa de entrega, o usu rio ganha a entrega gr tis. Com essa altera o, nosso pedido fica assim: const meuPedido = { itens: [{ nome: 'Po o de energia', valor: 100, quantidade: 2 }, { nome: 'Espada longa', valor: 3000, quantidade: 1 }, { nome: 'Entrega', valor: 30, entrega: true },], }; E nossa fun o de c lculo agora

precisa lidar com mais condi es: const calcularValorPedido = (pedido) => { const valorProdutos = pedido.itens .filter(produto => !produto.entrega) .reduce((prev, curr) => prev + curr.valor * curr.quantidade, 0); const entrega = pedido.itens.find(produto => produto.entrega); if (valorProdutos > 500) { return valorProdutos; } else { return valorProdutos + entrega.valor; } }; const resultado = calcularValorPedido(meuPedido); console.log(resultado); Agora, a fun o est mais complexa. Ela verifica se o pedido inclui a entrega e se o valor total dos produtos ultrapassa 500. Se sim, a entrega gratuita. Caso contr rio, ela soma o valor da entrega ao total. Complexidade crescente: por que testar desde o in cio? Inicialmente, t nhamos um nico cen rio: somar os valores dos itens. Agora, com a nova funcionalidade, temos dois caminhos diferentes no c digo: Caminho 1: Se o valor total dos produtos for maior que 500, n o cobramos pela entrega. Caminho 2: Se o valor for menor, somamos o valor da entrega ao total. Mesmo com essa pequena altera o, j temos um c digo mais dif cil de manter na cabe a e mais propenso a erros. E se mais funcionalidades forem adicionadas? Se novas regras de descontos, condi es de frete, ou c lculos diferentes para produtos espec ficos entrarem na l gica? Cada nova camada de complexidade tornaria essa fun o mais dif cil de entender e propensa a falhas. Passo 3: Testar desde o come o poupa problemas futuros Quando o c digo est simples, f cil pensar que adicionar testes desnecess rio. No entanto, fun es como essa crescem rapidamente em complexidade. Se n o implementarmos testes no in cio, prov vel que, conforme a fun o se torne mais complexa, acabemos evitando a adi o de testes por achar o processo muito demorado ou complicado. Por que testar desde o come o? Testes garantem que mesmo pequenas altera es n o gerem regress es. Ajudam a documentar o comportamento esperado do c digo. D o seguran a para refatorar e otimizar o c digo no futuro. Eliminam a necessidade de manter todas as regras de neg cios na mem ria. Facilitam a detec o de erros em cen rios complexos. Resumo A moral dessa se o simples: "Comece a testar o c digo desde o in cio". Mesmo que pare a bvio e simples agora, no futuro, quando o c digo estiver cheio de regras, exce es e condi es especiais, voc vai agradecer por ter criado uma boa base de testes. Os testes libertam sua mente para focar em novos desafios, pois voc n o precisar se preocupar constantemente com a possibilidade de cada pequena altera o

quebrar algo. Al m disso, ajudam a reduzir o estresse de apagar inc ndios e consertar bugs no c digo que poderiam ter sido evitados desde o come o.