01

Jasmine Aula 3

Transcrição

[00:00] Olá, bem-vindo ao capítulo 3 do curso de testes em JavaScript com Jasmine. Até agora você aprendeu a escrever testes e entendeu que testes são um cenário, uma ação, uma validação. Pensou em classes de equivalência, escrever um único teste pra equivalência, um teste mais simples. E essa bateria de testes está crescendo, você já viu que isso é legal, que você está testando caminhos diferentes do seu código, então é natural que você tenha muito o código de teste nesse momento.

[00:37] É até bastante comum que no mundo real você tenha até mais códigos de teste do que de produção, porque você viu que um if no código de produção gera 4, 5, 6 linhas no código de teste. Então, o ponto é que dado que essa bateria cresce muito rápido, o código cresce muito rápido, você tem que escrever um código fácil de ser mantido, do mesmo jeito que o código feio te atrapalha na produção, ele também te atrapalha no teste. Então, vamos lá, de volta aqui com o nosso código.

[01:04] Dá uma olhada na nossa "ConsultaSpec". Se você terminou todos os exercícios está maior do que o meu, mas veja, eu tenho linha repetida em todos eles. Essa aqui é o var Guilherme. Em todos os meus códigos aqui eu escrevo "var Guilherme", repetidamente o mesmo código. Qual o problema disso? O problema é que imagina que essa linha mude, eu tenho que passar mais um parâmetro aqui, um outro número qualquer. Eu vou ter que mudar isso em todos os lugares. Estranho, problemático, código repetido você já sabe dos problemas, eu nem preciso ficar discutindo muito sobre isso.

[01:40] Então, eu vou mostrar pra vocês como fazer para resolver, como fazer para escrever esse Guilherme uma única vez e deixar de repetir. Eu preciso desse Guilherme em todos os métodos de teste, eu vou precisar que essa variável tenha um escopo maior e seja enxergada por todos os testes. Eu vou ter uma função qualquer, essa ideia e essa função qualquer que vai fazer para mim lá o Guilherme new paciente.

[02:08] Inicializar coisas num teste é comum. É comum que vários testes, na mesma bateria, dentro do mesmo describe, tenham, por exemplo, um mesmo objeto paciente, ou tenha um começo igual. Então, o Jasmine já espera isso e ele dá para nós o que ele chama de "beforeEach". O "beforeEach" recebe uma função, que é exatamente essa função que escrevemos, e o que ele faz? Ele vai executar essa função antes de cada teste. Para mostrar até eu vou escrever aqui "console.log("beforeEach")".

[02:37] Ou seja, antes de rodar esse teste "não deve cobrar nada se for um retorno" ele roda o beforeEach. Antes de rodar o segundo teste ele roda o beforeEach, antes de rodar o terceiro teste ele roda o beforeEach. Voltando pro nosso browser, olha ali, três beforeEach. Um para cada teste. Essa é a ideia, vou apagar o "console.log" que é feito, eu não preciso dele. Essa é a ideia, se você tem um código repetido em todos os testes, isola isso e faça o uso do beforeEach.

[03:06] Eu, em particular, sempre tenho beforeEach no meu código. Todo o texto, todo código repetido vai para ele. A primeira boa dica é: use beforeEach. A segunda dica, que eu usei describe lá em cima, mas eu posso ter outros subníveis de describe se isso me ajudar a descrever melhor meus testes. Então, por exemplo, imagina que eu tenho uma bateria de testes que são consultas do tipo retorno, eu poderia agrupá-los em outro describe.

[03:40] Vou levar esse it aqui para dentro. Dar um tab pra ficar bonito. Então, consultas do tipo retorno, eu vou colocar um outro subnível aqui que vai ser consultas com procedimentos, porque esses dois testes embaixo fazem os testes dos dois procedimentos para mim, eu vou levar os dois. E o que isso muda? O teste é o mesmo, a diferença é o relatório, dá uma olhada, consulta, consultas do tipo retorno, consultas com procedimentos, e ele vai só separando melhor o relatório.

[04:17] Agora eu usei os describes não só para separar os testes de cada uma das minhas classes, mas também para separar comportamentos maiores, responsabilidades maiores dentro de uma classe em particular. Você pode ter describes alinhados e ele te dá um relatório mais amigável pra isso. Se você tiver conjuntos de testes que são semelhantes, até faz algum sentido agrupar em describes, esse relatório é em português, então é legível e vai te ajudar caso seu código tenha algum problema, vai ficar mais fácil de entender esse problema.

[04:56] Próximo passo. Dê uma olhada no código que usamos para criar um paciente. New paciente Guilherme, 28 anos, 72 quilos, 1.80 de altura, quatro parâmetros. Parece fácil. Agora imagina uma entidade paciente que fosse mais complicado, com dez campos para endereço, carteirinha de vacinação, número do convênio, um monte de informação. Imagina só em todo lugar você tem que dar um new e passar estado completo.

[05:30] Pensa o seguinte, aqui eu estou usando o paciente para testar a consulta, mas o paciente é provavelmente uma entidade importante no meu sistema, então, eu vou ter esse new paciente em outros lugares. Toda vez que minha entidade mudar, eu vou ter que mudar em vários lugares. Se ela é complicado de ser criada, eu vou ficar passando um monte de parâmetro pra ela, que às vezes eu nem preciso, nem quero. Eu passo o nome Guilherme porque é obrigatório, mas o nome pouco importa para esse teste.

[05:55] Eu tenho uma entidade que é grande, é complicada de ser criada, que tem dados que eu não preciso usar o tempo inteiro e eu quero facilitar minha vida, com é que fazemos? Essa aqui é uma cópia do que fazemos em linguagens como C#, em Java, e que para mim faz total sentido também em JavaScript. Veja só o que eu vou fazer, no meu source aqui eu vou criar um outro aqui e vou chamar de "PacienteBuilder".

[06:18] E o nome dele já diz, o objetivo vai ser facilitar a criação de pacientes, a palavra builder vem lá do padrão de projeto Builder, se você não conhece ele direitinho faz o nosso curso de padrão de projeto em Java, em C#, que você vai entender o que é o builder. Mas a ideia é facilitar a criação de objeto. Qual? O objeto paciente. Primeira coisa que eu vou fazer é definir valores padrões para cada um dos atributos.

[06:44] Então, o nome vai Guilherme. A idade vai ser 28, o peso vai ser 72, e a altura vai ser 1.80. Eu vou criar a classe. "var clazz", "return clazz", e o primeiro método que eu vou fazer nele é o método que todo builder tem, que chamamos de "controi". Esse método vai retornar pra nós um paciente com os dados que ele tem salvo, o nome, idade, peso e altura. Esse código aqui já funciona, então se eu for aqui no meu "ConsultaSpec", ao invés de eu fazer new paciente eu posso fazer "new PacienteBuilder().constroi()".

[07:28] Ele já vai me dar um paciente com os dados que eu preciso. Muito mais simples. Em todo lugar agora que eu precisar de um paciente, ao invés de eu fazer new paciente e passar os dados fixos, eu passo paciente builder ponto constrói e ele vai passar para mim, Guilherme, com um peso específico e tal. "Mas aqui na minha classe paciente, cadê os testes de paciente, vou precisar variar o peso, a altura, preciso mudar", é fácil.

[07:49] Vamos aqui no nosso builder e da métodos para isso. Então, por exemplo, se a idade é importante e precisa mudar, eu posso criar um método que eu vou chamar de "comIdade", que eu vou passar um valor qualquer aqui, e esse cara vai fazer assim, "idade = valor". Que eu vou até fazer um "return this". Já explico o porquê. "Ah, preciso mudar o peso", "comPeso : function (valor)", e eu mudo o peso. O peso é igual ao valor que chegou, return this.

[08:16] Percebeu o que eu estou fazendo? Estou criando métodos no meu builder que modificam os valores padrão. Então, o ponto é, se eu estou aqui no meu consulta spec e eu preciso mudar o peso do paciente, eu faço com peso e passo 50, ".constroi". Se precisa mudar a idade, com idade de 12 anos, ".constroi". Eu posso mudar, mas se não for importante eu tenho um valor padrão. Essa é a sacada do builder, ele facilita a criação de objetos pro teste e nele eu ponho qualquer método que me ajude.

[08:44] Então, "comIdade" para mudar a idade, "comPeso", etc, mas eu sei que tem valores padrão, que não estão preenchidos de graça para mim, essa é a ideia. Agora, código mesmo porque aqui no meu builder eu retornei this, pra

conseguir escrever de maneira fluente, como chamamos, então, eu faço com peso 10 ponto, que ele vai me deixar invocar o método constrói, porque o método "comPeso" retorna o this, que é o próprio objeto. Então, ele vai ter um método constrói.

[09:11] Vamos ver se funciona. No "SpecRunner" eu não posso esquecer de importar o "PacienteBuilder". Não achou o "PacienteBuilder" porque eu errei a pasta, coloquei no source. Idealmente tem que estar na pasta spec, que tem mais a ver com os teste. Poderíamos mover esse código pra lá. Vou ter que mover aqui pelo Finder, porque o sublime não está movendo. Então, vou pegar a pastinha source, "PacienteBuilder" eu vou levar para a pastinha spec e fica melhor. O que é de teste na pasta de teste.

[09:57] Volto aqui pro "SpecRunner", aqui ponho spec de volta, volto pro meu teste, rodo. Tudo continua passando, já estávamos esperando isso. Isso é o que nós chamamos de test data builder. São builders para dados de teste, para cenários de teste. O que ganho com isso? De novo, fica fácil criar cenários se a minha classe paciente um dia mudar, eu não vou ter que sair mudando em todo lugar fazia "new paciente" no meu teste. Eu vou mudar simplesmente aqui no meu builder.

[10:33] Vou acrescentar um parâmetro a mais aqui, então, var novo parâmetro. Vou colocar lá no método constrói e assim por diante. Vai ficar fácil, eu mantenho em um lugar só. Então, veja só que é um pouquinho do que falamos de encapsulamento, a lógica de criar um paciente está encapsulado no meu teste data builder. Então, nesse capítulo eu discuti com você boas práticas para você escrever código de teste de qualidade. Falei pra você não repetir código, mostrei pra você o beforeEach, que vai ser executado antes de cada teste.

[11:06] "Ah, mas eu tenho comportamentos que vou utilizar em vários outros teste também, como eu aproveito?" Nada te impede de criar outras classes de teste que tenham lá algum comportamento que você precisa. Lembre-se: favoreça a testabilidade, facilite seu teste. Isso vai implicar em escrever infraestrutura de testes. Não tenha vergonha e não tenha medo disso, escreva o código que te ajudar pro teste, pra te ajudar a manter o código de testes.

[11:33] Mostrei o beforeEach, mostrei o describe, que você pode ter vários describes alinhados e isso te dá um relatório mais elegante, mais formatado e mostrei para vocês aí o "PacienteBuilder". O test data Builder, que é o nome genérico do padrão de projeto, que são classes que ajudam a montar cenários de teste. A vantagem: encapsular o processo de criação de objetos de teste em um único lugar, fica fácil de utilizar, fica fácil evoluir, facilitando a manutenção do teste.

[12:04] Tudo que você puder fazer para escrever um código de testes simples e fácil de ser mantido, faça. Cuide do código do seu teste. Então, era isso que eu queria falar nessa terceira aula para vocês. Obrigado e até a próxima aula.