01

Jasmine Aula 5

Transcrição

[00:00] Olá, bem vindo ao quinto capítulo do curso de testes em JavaScript com Jasmine. Para começar o capítulo, eu vou melhorar um pouquinho a funcionalidade que nós temos e eu já explicou o porquê. O que eu preciso fazer é a consulta tem uma data também, lembra que tem aquela história de retorno? Eu vou colocar uma data e hoje que eu só tenho esse método preço, eu preciso colocar outros métodos pra conseguir pegar os dados da minha consulta.

[00:34] Então, por exemplo, eu vou "getNome", que vai me devolver aqui o nome do paciente, a variável chama-se paciente. Vou criar um "getProcedimentos", que me devolve a lista de procedimentos. Vou criar "isParticular", que me devolve se é particular ou não, vou criar um "isRetorno", que me devolve se é retorno ou não. A variável chama retorno e, por fim, um "GetData", que me devolve a data.

[01:17] O que eu quero implementar agora é a função de agendamento. Agendamento de retorno. Então, como ela funciona? Ela vai pegar a data da consulta e vai somar 20 dias, o retorno é sempre 20 dias para frente. É a regra de negócio que eu quis inventar aqui. Só que dessa vez vamos fazer diferente, ao invés de sair programando uma classe agendamento vamos começar primeiro pelo teste. Essa vai ser a ideia.

[01:45] Começar pelo teste, escrever o teste e depois que o teste estiver escrito, eu vou escrever o código de produção. Ao invés começar pelo source eu vou começar pelo spec, e eu vou chamar essa classe de "AgendamentoSpec.js". Eu vou descrever um agendamento. Meu teste vai começar aqui como "deve agendar para 20 dias depois". Function. Vou criar um paciente, "var gui = new PacienteBuilder().constroi()".

[02:31] "var agenda = new Agendamento()". E veja só que esse aqui não vai funcionar e eu já sei, não existe a classe agendamento ainda, mas não tem problema. "var consulta = new Consulta", vou passar o Guilherme, nenhum procedimento, falso, falso e vou passar uma data aqui, 2014, 7, 1. Um detalhe, mês sete no JavaScript é agosto, porque janeiro é zero, é pra confundir mesmo. É assim que funciona no JavaScript.

[03:00] Na sequência, eu vou fazer assim, "var novaConsulta = agenda.para(consulta). Esse é o método que eu quero criar. Vou criar um método chamado "para" que vai ser consulta e vai me devolver uma outra consulta. O método não existe ainda, eu estou imaginando aqui na minha cabeça, colocando em código. Não tem problema. O meu assert vai ser assim, eu espero que a data dessa nova consulta, em string, para comparar mais fácil, seja igual a "new Date(2014, 7, 21)".

[03:31] Vinte dias para frente, não tem segredo. Salva tudo. Vamos no "SpecRunner", vamos colocar aqui, "AgendamentoSpec". Já sabe o que vai acontecer, vai dar erro agendamento is not defined, era esperado. Vamos começar a programar. Vou na classe source, vou criar aqui o agendamento JavaScript. Function Agendamento. "var clazz", padrão nosso, não tem segredo. E vamos lá, o método se chama "para", e eu sei que ele recebe uma consulta.

[04:16] Eu preciso implementar ele para fazer o teste passar, e eu vou escrever o código mais simples que eu conseguir imaginar para fazer esse teste passar. O código mais simples vai ser retornar na data fixa mesmo. Então, eu sei que eu estou esperando 2014, 7, 21, nova consulta é igual a consulta com o nome do mesmo paciente do anterior. Os mesmos procedimentos, se é particular, o retorno é true, nova data. Retorna nova consulta. Esse é o nosso código. Vamos lá no "SpecRunner" de novo.

[04:58] Vou completar aqui com "Agendamento.js". E olha o que acontece, o teste passa, implementamos com sucesso. Meu teste passou, mas eu estou triste, porque dá uma olhada aqui no meu código de produção, cadê o agendamento? Tem uma data fixa, horrível isso, você sabe que não é assim no mundo real, mas tudo bem, valeu a pena como primeiro

código. Justamente para vermos nosso teste verde, mas agora que está verde eu vou refatorar meu código, pra tirar esse código fixo daí.

[05:31] Como eu vou fazer? Em JavaScript eu não tenho um jeito muito legal de fazer contas com datas, então, o que vou fazer aqui é o seguinte, a nova data vai ser exatamente igual a data anterior, getData. Só que vou fazer getTime, o getTime devolve para nós a data em milissegundos, desde 1970, aquele formato padrão de data. Eu vou somar com 20 dias em milissegundos.

[05:59] Essa variável eu vou ter que criar, que vai ser "1000 * 60 * 60 * 24 * 20". Vinte dias vezes horas por dia, vezes minutos, vezes segundos, vezes milissegundos. O Código está mais genérico agora, meu teste continua passando, só que agora eu sei que o meu código vai funcionar para qualquer data. Então, esse primeiro teste passa, a funcionalidade de agendar para 20 dias está ok e o teste está verde. Agora vou para a próxima.

[06:35] E a próxima funcionalidade é que o agendamento tem que ser 20 dias depois, mas se cair no fim de semana ele vai ter que empurrar pro próximo dia útil. Porque o meu consultório não trabalha de sábado e domingo. Eu vou fazer a mesma coisa, vou começar pelo teste, vamos lá. It "deve pular fins de semana". Function. Aqui é a mesma coisa, var consulta, é uma nova consulta.

[07:05] Preciso desse gui, já vou usar aqui o beforeEach, que já vimos no capítulo passado, para facilitar minha vida. Escrever menos código. Essas duas linhas aqui serão iguais e eu vou colocar no beforeEach. Var gui, var agenda, tab dois pra ficar mais bonito o código. Então, aqui agora eu vou só criar consulta. New consulta gui, vazio, falso, falso, nova data em 2014, mês 6, porque começa no zero, 30.

[07:41] Se você olhar no calendário, esse é um dia normal, só que se você somar 20 dias, vai dar justo no fim de semana, eu não escolhi essa data à toa, eu escolhi uma data que eu sei que 20 dias para frente vai dar fim de semana. A nova consulta vai ser gerada pela agenda e o meu assert vai ser o seguinte, se eu pegar a data, ponto to string ponto toEqual, eu sei que essa data tem que ser igual não só 20 dias para frente, mas tem que ser igual a 2014, 6, 21, que é o próximo dia útil, depois do 30 do 6, vai ser o 21 do 7.

[08:28] Lembra que começa no zero, ".toString". Se eu vou dar esse teste ele vai falhar. Ele estava esperando que fosse 21 de julho, mas veio 20 de julho. O teste falhou porque era esperado, eu estava esperando isso, já que o meu código ainda não trata essa ideia de fim de semana. O teste falhar era o que eu estava esperando mesmo.

[08:55] Então, qual vai ser minha implementação? Minha implementação vai ser assim, eu calculei a data, só vou fazer loop, enquanto for fim de semana eu vou somando dias nela. Então, dá uma olhada, "while(novaData.getDay() == 0", ou seja, é um domingo, e aqui é a API do JavaScript, tem que conhecer, não tem jeito. Ou se for um domingo ou sábado, a nova data é igual a new data, "novaData.getTime() + umDiaEmMillisegundo". Fiz o loop.

[09:31] Falta criar essa variável aqui, que eu vou colocar como mil vezes 60 vezes, vezes 60, 24. Um dia em milissegundos. Exatamente o que está aqui. Vamos ver se eu rodar o teste? Ótimo, passou. Então, meu código agora está funcionando, empurrando 20 dias para frente, se cai em um dia de semana perfeito, e vinte dias para frente se cai em um fim de semana ele vai procurar o próximo dia útil disponível.

[09:58] Ótimo, está tudo funcionando, mas meu coração está apertado porque eu estou com um pouco de código repetido, dá pra melhorar. Isso aqui é um dia em milissegundos, aquele código daquela conta é feia, quanto mais bonito eu conseguir deixar, melhor. Refatorei, rodo o teste de novo, continua passando, refatorei com sucesso meu código de produção. Meu teste me avisou, se eu tivesse feito uma besteira, como errar o nome de da variável aqui, o código irá falhar. Muito bom.

[10:24] De novo aquela parte da segurança, o teste passou, ótimo. Agora, quero discutir com vocês o que fizemos aqui. Começamos pelo teste, primeira coisa escrevemos um teste. O teste falhou porque eu já estava esperando isso. E o que

eu fiz? Eu fiz uma o meu teste passar, eu fiz ele passar da maneira mais simples que eu podia, no primeiro código, em particular, até retornei a data fixa, e o teste passou. Eu refatorei, ou seja, melhorei o meu código de produção. No primeiro caso eu tirei a data fixa e fiz ela ficar genérica, e agora fiz refatoração simples porque eu tinha a conta matemática repetida.

[11:02] Escrevi um teste, fiz o teste passar, refatorei, escrevi um teste, fiz o teste passar, refatorei. Se eu tivesse uma outra funcionalidade eu ia repetir isso. E esse ciclo, é o que nós chamamos de TDD, Test-Driven Development. Desenvolvimento Guiado pelos Testes. TDD é uma prática que está muito famosa na indústria, todo mundo fala disso, comece pelos testes.

[11:27] O ponto é o que eu ganho? Quando eu começo pelo teste, eu primeiro penso no requisito. Porque a hora que eu estou escrevendo o teste, e veja comigo, a hora que você como estava aqui comigo escrevendo o teste deve agendar para 20 dias depois, ou deve pular o fim de semana, você não estava pensando em implementação, você só estava pensando no que o requisito tinha que fazer. Então, dado essa data essa era a saída, dado essa data essa era a saída e assim por diante. A implementação não estava na cabeça.

[11:49] A implementação veio na cabeça a hora que eu vi o teste ficar vermelho e eu fui implementar. Então, eu penso mais um requisito. Eu separo meu pensamento mental em primeiro penso no requisito, depois a implementação. Dá uma ordem para o meu raciocínio. Tem bastante estudo que mostra que quando você faz TDD, por você pensar mais no requisito, você escreve mais teste do que o cara que não faz TDD e deixa para escrever o teste depois.

[12:12] Então, meu código ele sai mais testado naturalmente. Segundo ponto, ele vai até me ajudar a pensar no meu código. Quando escrevo esse primeiro teste aqui, começamos "agenda.para(consulta)", que devolve uma nova consulta. Isso é lição design, sou eu rabiscando as minhas primeiras lições de design. E se eles tivessem ficado ruins, feias, se eu não tivesse gostado da API que eu desenhei, eu simplesmente ia mudar e essa mudança não ia me custar, porque o código de produção ainda nem existe.

[12:41] Então, tem muita gente que fala que o TDD ajuda você até a escrever um código melhor, justamente porque ela deixa você brincar, experimentar mais com um custo mais baixo. No mundo tradicional, se você escreve um caminhão de código e depois resolve mudar a interface da sua classe, essa mudança vai custar mais, porque você já escreveu mais código.

[12:59] Essas são as vantagens do TDD, seu código sai testado, melhor testado porque você escreve mais código, você pensa mais no requisito, você tem esse tipo de feedback, o design. Aqui no curso eu falo pouco, mas eu, em particular, já escrevi um livro sobre isso, dá olhada lá na casa do código, tenho meu livro sobre TDD, onde eu falo muito, foi meu trabalho de mestrado, então é um assunto que eu discuti lá com uma certa profundidade.

[13:23] TDD é uma prática bem legal, e é por isso que ela ficou bastante famosa. Vale a pena você pensar em eventualmente praticar a TDD. E veja que nem é lá tão difícil assim. Comece pelo teste. Escreveu o teste, começa a implementar o seu código, faz ele passar da maneira mais simples, volta para o código de produção, refatora. Refatorou? O teste está verde? Passa para a próxima funcionalidade. Isso é TDD, não tem segredo.

[13:51] Nesse capítulo eu também refatorei, usei meu beforeEach para facilitar a manutenção do meu teste. Discuti isso em capítulos passados. Aqui mostrei o TDD pra vocês, espero que vocês tenham gostado, espero que vocês tenham entendido. A regra de negócio é um pouco complicada, o exemplo que eu dei, nesse monte de conta aqui que eu tive que fazer, porque a API de JavaScript é chata. Você pode até conhecer uma outra API de JavaScript mais legal para fazer data, ótimo, muda o código de produção e roda teste.

[14:20] Se passou é porque sua implementação está funcionando, essa é a vantagem de ter teste. Podemos mexer na implementação à vontade, sem muito medo de ser feliz. Então, esse é o nosso curso de testes em JavaScript. É uma introdução a testes, eu quis mostrar para vocês que escrever testes é fácil, em JavaScript, em particular, vai muito de

você programar com qualidade em JavaScript, que é uma coisa que a comunidade está aprendendo agora. Porque JavaScript é uma linguagem hoje de primeiro nível. Madura, todo mundo quer usar, todo mundo usa. Está todo mundo agora discutindo boas práticas na linguagem.

[14:52] Então, se vocês escreve um código de qualidade, eu tenho certeza que vai ser fácil testar. Mostrei pra vocês o Jasmine, que é uma opção bem legal, bem fácil de ser utilizada. Só que do Jasmine aprendemos pouca coisa, porque ele é simples. É o describe, é o it, o beforeEach, não tem muito segredo. É usar o SpecRunner para rodar os meus testes. É bem simples.

[15:13] Falei de TDD para vocês, prática importante e é bom ter no seu cinto de segurança. Falei para vocês de manutenção de código de teste. Então, escreva teste fácil de ser mantido. Falei para vocês para vocês ao máximo tentarem isolar código que lida com coisas de interface, de código, de regra de negócio. É por isso que aquele frameworks como o AngularJS eles têm ficado populares, porque eles forçam essa separação e isso facilita a escrita do seu teste.

[15:41] Então, esse é o curso de teste em JavaScript, é uma introdução ao assunto. Tem muita coisa para pesquisar, muita coisa para estudar, mas tenho certeza que com o que eu falei aqui você consegue testar 90%, 95% de seus códigos. Lembra que testar tem que ser fácil, se não está fácil é porque o seu código está meio complicado.

[16:00] Eu espero que você tenha gostado, qualquer dúvida vai lá pro Fórum, abra uma dúvida que eu e toda a comunidade do Alura que hoje é gigante, vai ajudar você. Espero que tenha gostado, obrigado por ficar comigo por nesses cinco capítulos, eu espero de coração que você tenha gostado. Um abraço.