02

Testes Automatizados

Bem vindo ao curso de testes automatizados em Javascript. Testar é tarefa fundamental nos dias de hoje. O que mais vemos por aí são sistemas de software com bugs. E bugs, como sabemos, causam prejuízos e dores de cabeça enormes.

Javascript, em particular, é uma linguagem que tem crescido demais no mercado. Uma das razões para tal é que ela roda em qualquer browser. Desenvolvedores web então, abusam dessa vantagem e fazem uso extenso da linguagem, para todos os fins possíveis.

Neste curso, você vai ver que testes em Javascript são fáceis; a parte difícil é conseguir isolar todo o código que lida com o HTML das regras de negócio. Aqui não faremos testes como aqueles que abrem o browser e clicam nas coisas; testaremos as regras de negócio!

Para começar, vamos escrever uma classe em Javascript que, dado uma lista de números, ele nos devolve o maior e menor elementos dessa lista. O algoritmo é o convencional: passearemos pelo array de números, e guardaremos o maior e o menor de todos.

O código abaixo faz isso. Vamos salvá-lo em um arquivo `MaiorEMenor.js` :

```
function MaiorEMenor() {

    var menor;
    var maior;

    var clazz = {

        encontra : function(nums) {

            menor = Number.MAX_VALUE;
            maior = Number.MIN_VALUE;

            nums.forEach(function(num) {
                if(num < menor) menor = num;
                else if(num > maior) maior = num;
            });
        },

        pegaMenor : function() { return menor; },
        pegaMaior : function() { return maior; }
    };

    return clazz;
}
```

Sem segredo. Se rodarmos esse código em um HTML qualquer de exemplo, e passarmos um array de números pra ele, a saída é a esperada: `15 e 5` . Veja:

```
<script>
var algoritmo = new MaiorEMenor();
algoritmo.encontra([5,15,7,9]);
```

```
console.log(algoritmo.pegaMaior());
console.log(algoritmo.pegaMenor());
</script>
```

Tudo funcionando. Isso quer dizer então que podemos colocar esse código em produção? Errado! Veja só o próximo teste:

```
<script>
var algoritmo = new MaiorEMenor();
algoritmo.encontra([9,8,7,6]);

console.log(algoritmo.pegaMaior());
console.log(algoritmo.pegaMenor());
</script>
```

A saída agora é estranha. Ou seja, o algoritmo não funcionou para um caso específico. E se isso estivesse em produção? Quanto será que esse bug não custaria?

A solução pra isso, todos já sabemos: **testar software**. Agora a pergunta é: por que não testamos? Não testamos, porque o teste manual custa caro, demora, é chato e trabalhoso. A solução pra isso? **Escrever testes automatizados**.

Um teste automatizado é um teste que é executado pela máquina. A máquina, ao contrário de um ser humano, não acha chato e trabalhoso, não demora, e não cobra caro para tal. Daqui pra frente, aprenderemos a fazer com que a máquina nos ajude a testar.

E fazer isso nem é tão complicado. No código acima, já fizemos boa parte de um teste automatizado. Todo teste é dividido em três partes: cenário, ação e validação. O cenário é o que daremos de entrada para o método que queremos testar (no caso, 5, 15, 7, 9, por exemplo). A ação é o método que queremos testar (no caso, `encontra()`). A validação é a hora que verificamos se a saída bate com o que estávamos esperando (no caso, olhando o `console.log`).

A única parte que não está automatizada é a validação, afinal um ser humano ainda precisa olhar a saída. É justamente isso que vamos melhorar. E, para facilitar nossa vida, usaremos um framework que nos dirá, de maneira mais elegante, qual "console.log" falhou, e onde ele está.

Em Javascript, o mais popular deles é o **Jasmine**. Você pode baixar o Jasmine no próprio site dele no github (http://jasmine.github.io/). Ao baixá-lo, você verá que ele já vem com alguns códigos de exemplo.

Veja o `SpecRunner.html`, por exemplo. Ele importa as bibliotecas do Jasmine, que estão em `/lib`. Em seguida, ele importa os javascripts de exemplo `Player.js` e `Song.js`, que estão em `/src`. Por fim, ele importa as `Specs`, que são os nossos testes.

Para facilitar, usaremos essa mesma estrutura ao longo do curso. Colocaremos todos nossos códigos Javascript na pasta `/src`, e todos nossos testes na pasta `spec`.

Começaremos por colocar o `MaiorEMenor.js` dentro da pasta `/src`. E aí criaremos nosso primeiro teste, no arquivo `MaiorEMenorSpec.js`, dentro do diretório `spec`.

A primeira coisa que devemos colocar nesse arquivo é um `describe`. Ele serve para descrever o que vamos testar. No caso, como vamos testar o MaiorEMenor, colocaremos isso nele. Veja que ele recebe uma função como segundo parâmetro. Acostume-se com isso, no Jasmine (bem como no Javascript), passaremos funções para todos os lados:

```
describe("Maior e Menor", function() {

});
```

Dentro dessa função, agora colocaremos cada um dos nossos testes. No primeiro deles, faremos exatamente o primeiro cenário que fizemos hoje: uma sequência de números fora de ordem. Para isso, colocaremos dentro desse describe, uma função chamada `it`. Nela passaremos o "nome do teste", ou seja, o quê aquele teste faz, e em seguida uma função com o teste.

Veja:

```
describe("Maior e Menor", function() {

    it("deve entender numeros em ordem nao sequencial", function() {
        var algoritmo = new MaiorEMenor();
        algoritmo.encontra([5,15,7,9]);

    })
});
```

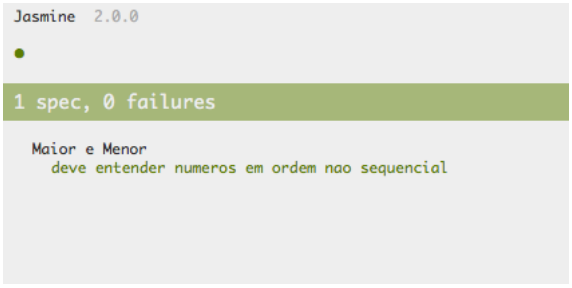
Tudo certo até agora. Criamos um teste, passamos um cenário e invocamos uma ação. Falta agora a parte da validação. E é aí que vamos usar o Jasmine. Frameworks de teste possuem métodos específicos para fazer a validação. No caso do Jasmine, precisamos conhecer o método `expect` que, como o próprio nome diz, serve para dizermos quais são as nossas expectativas com o resultado final. Sabemos que o valor maior é 15, e o menor é 5. Então colocaremos isso no teste:

```
describe("Maior e Menor", function() {

    it("deve entender numeros em ordem nao sequencial", function() {
        var algoritmo = new MaiorEMenor();
        algoritmo.encontra([5,15,7,9]);

        expect(algoritmo.pegaMaior()).toEqual(15);
        expect(algoritmo.pegaMenor()).toEqual(5);
    });
});
```

Ótimo. Vamos executá-lo. Para isso, abra o `SpecRunner.html` e acrescente as duas linhas abaixo, que incluem a classe e o teste no runner:

```
<script type="text/javascript" src="src/MaiorEMenor.js"></script>
<script type="text/javascript" src="spec/MaiorEMenorSpec.js"></script>
```

Agora abra esse arquivo HTML no browser, e veja a saída.



Nosso teste passa. Mas agora vamos ao próximo cenário, que sabemos que não funciona. Números em ordem decrescente. Para isso, basta adicionarmos outro `it` dentro do mesmo `describe` com o novo cenário:

```
it("deve entender numeros em ordem decrescente", function() {
        var algoritmo = new MaiorEMenor();
        algoritmo.encontra([9,8,7,6]);

        expect(algoritmo.pegaMaior()).toEqual(9);
        expect(algoritmo.pegaMenor()).toEqual(6);
});
```

Dessa vez o teste falha:

O problema está na implementação. Veja que temos um `else` sobrando. Vamos corrigir a implementação:

```
function MaiorEMenor() {

    var menor;
    var maior;

    var clazz = {

        encontra : function(nums) {

            menor = Number.MAX_VALUE;
            maior = Number.MIN_VALUE;

            nums.forEach(function(num) {
                if(num < menor) menor = num;
                if(num > maior) maior = num;
```

```
            });
        },

        pegaMenor : function() { return menor; },
        pegaMaior : function() { return maior; }
    };

    return clazz;
}
```

O teste agora passa. Veja só: o teste roda MUITO rápido. Não conseguimos nem vê-lo rodando. Isso quer dizer que podemos executá-lo o tempo todo.

O que ganhamos com isso? Segurança. Se algum desenvolvedor, por descuido, fizer alguma alteração errada nessa classe, o teste rapidamente pegará. E é o que acontece no dia-a-dia, certo? O desenvolvedor está o tempo todo alterando código que já funciona; e sabemos que é difícil lembrar de todos os possíveis cenários. É para isso que o teste está lá. Para nos dar segurança em caso de mudanças!

Repare que escrever um teste automatizado não é nem tão difícil. Apenas invocamos um método e verificamos a sua saída. E fazemos isso para vários cenários diferentes. Escrever testes é um caminho sem volta!