www.devmedia.com.br
[versão para impressão]
Link original: http://www.devmedia.com.br/articles/viewcomp.asp?comp=33579

JavaScript QUnit: Conheça o Framework de testes unitários

Este artigo demonstra como trabalhar com o framework QUnit para o desenvolvimento de testes unitários na linguagem de programação JavaScript.

Na maioria das vezes, aplicações baseadas em **JavaScript** tendem a crescer e ficar complexas diante da crescente quantidade de regras de negócio na aplicação. Mesmo com bons programadores trabalhando no código JavaScript para melhoria o software ainda terá defeitos. Os engenheiros de teste fazem o melhor para reduzirem as falhas antes do software ser lançado, mas dificilmente detecta-se todas as falhas manualmente. Então, para melhorar e aumentar a eficiência dos testes na identificação das falhas, os testes automatizados podem ser uma alternativa bastante viável, pois revelam qualquer (se bem planejado) problema presente nas fases iniciais de desenvolvimento, assim reduzindo o custo da correção dos defeitos de forma significativa.

Independentemente das tecnologias utilizadas, as melhores práticas de desenvolvimento incluem a automação dos estágios de testes:

- Testes unitários: Tem como objetivo testar a menor parte (por exemplo, classe, métodos) da aplicação;
- Testes de integração: Tem como objetivo testar os componentes (por exemplo, classes) em grupos combinados;
- Testes de sistemas: Tem como objetivo testar se o sistema funciona como um todo;
- Testes funcionais: Tem como objetivo testar as funcionalidades do sistema, a nível de aplicação.

Atualmente, o mercado oferece várias ferramentas para a automação de testes em JavaScript. Uma das mais populares é o QUnit, tema deste artigo.

Conhecendo o QUnit

O QUnit é um framework de testes unitários JavaScript que nos ajuda a testar o código. Desenvolvido pela equipe do jQuery, este é atualmente o programa padrão de todos os projetos do grupo, incluindo o core do jQuery, jQueryUI, jQueryMobile, entre outros. Além disso, o QUnit pode ser utilizado para testar qualquer código baseado em JavaScript.

Para começar a utilizá-lo vamos primeiro incluir o código da **Listagem 1** no chamado "index.html", onde contém todos os arquivos (html, css e js) necessários para o funcionamento do QUnit.

Listagem 1. Código da index.html para iniciar o QUnit

```
<html>
<head>
  <meta charset="utf-8">
  <title>Artigo sobre o QUnit </title>
  <!--Inclui a folha de estilo do QUnit -->
  <link rel="stylesheet" href="http://code.jquery.com/qunit/qunit-1.19.0.css">
</head>
<body>
  <!-- Interface de usuário do QUnit-->
  <div id="qunit"></div>
  <div id="qunit-fixture"></div>
  <!-- Fim -->
  <!-- Incluir arquivo javascript de funcionamento do QUnit-->
  <script src="http://code.jquery.com/qunit/qunit-1.19.0.js"></script>
  <!-- Incluir arquivo javascript com os testes utilizando o QUnit-->
  <script src="tests.js"></script>
</body>
</html>
```

Em seguida, criaremos o arquivo JavaScript chamado "tests.js" com o código da **Listagem 2,** que contém o nosso primeiro teste com o QUnit.

Listagem 2. Código do arquivo tests.js

```
QUnit.test( "Primeiro teste com QUnit", function( assert ) {
 assert.ok( 1 == "1", "Passou!" );
});
```

Observe que na declaração de testes com QUnit declaramos a classe QUnit para a chamada do método test, incluindo os parâmetros (como o nome do teste e a função com ele). Dentro da function existe um método chamado "assert.ok", que apenas verifica se é verdadeiro ou falsa a comparação dos dois valores. Se verdadeiro, o QUnit apresentará a mensagem "Passou!" na página index.html.

Para executar o teste basta acessar no navegador a página index.html, onde o resultado do nosso primeiro teste, se tudo ocorreu com sucesso, terá como resultado o que mostra a **Figura 1**.

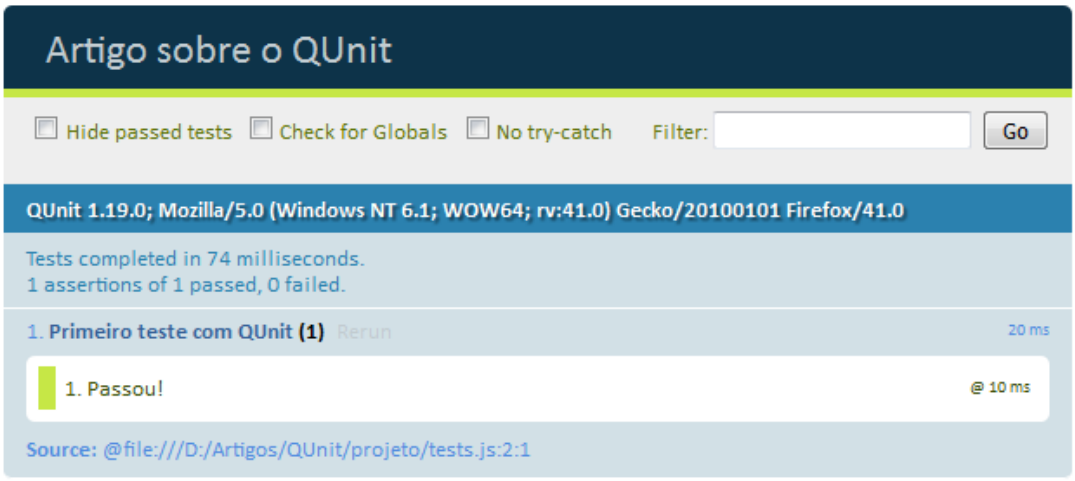

Figura 1. Resultado do primeiro teste com QUnit

Note que o QUnit mostra o cabeçalho da página ("Artigo sobre o QUnit") e abaixo é exibida uma barra verde, que significa que o teste passou. Se esta barra ficar vermelha, isso significa que um ou vários testes não passaram.

Na segunda seção existem três caixas de seleção:

- Hide passed tests: Se marcada, o QUnit vai esconder os testes que passaram;
- Check for Globals: Esta opção, se marcada, permite verificar se uma propriedade foi adicionada ao objeto window, comparando-a antes e depois de cada teste. Caso o teste falhe, serão listadas as diferenças;
- No try-catch: Se marcada, permite verificar se o código lançou alguma exceção quando executou os testes com o QUnit.

Além das caixas de seleção há um campo de busca ao lado do texto "Filter:", que podemos usar para filtrar os testes executados para procurar um em específico.

Na terceira seção podemos observar o valor da propriedade window.navigator.userAgent, que retorna as informações do agente do usuário do navegador que está acessando a página.

A seção inferior mostra o tempo gasto pelo QUnit para executar os testes definidos. Ainda nesta seção podemos ler o número de afirmações definido, o número de testes que passou e o número que falhou.

Testando o código usando afirmações de comparação

As afirmações são o núcleo de teste de software, porque elas nos permitem verificar se o código está funcionando conforme o esperado. Então, o QUnit fornece várias funções para comparação entre objetos ou valores dentro da função "QUnit.test()". Vejamos a visão geral das funções de comparação:

- equal(): Executa uma comparação não rigorosa (tipagem fraca), aproximadamente equivalente ao assertEquals da JUnit;
- notEqual(): Executa uma comparação não rigorosa, verificando se há desigualdade;
- strictEqual(): Executa uma comparação rigorosa (tipagem forte);
- notStrictEqual(): Executa uma comparação rigorosa, verificando se há desigualdade;
- propEqual(): Executa uma comparação rigorosa das propriedades e valores de um objeto;
- notPropEqua(): Executa uma comparação rigorosa das propriedades e valores de um objeto, verificando se há desigualdade;
- deepEqual(): Executa uma comparação rigorosa das propriedades, valores e o protótipo de um objeto;
- notDeepEqual(): Executa uma comparação rigorosa das propriedades, valores e o protótipo de um objeto, verificando se há desigualdade.

Para o melhor entendimento de cada método citado, faremos agora alguns exemplos utilizando-os.

Começaremos com as funções equal() e notEqual() utilizando o arquivo da **Listagem 1** e depois acrescentando o código da **Listagem 3** dentro da tag <head>.

Listagem 3. Função JavaScript que faz a multiplicação e retorna o resultado

```
<script language="javascript">
function multiplicar(a, b) {
                        return a * b;
    }
 </script>
```

Em seguida, utilize o arquivo da **Listagem 2,** substituindo o teste já existente pelo da **Listagem 4**, no qual faremos alguns testes utilizando a função multiplicar(), equal() e notEqual().

Listagem 4. Código do tests.js

```
QUnit.test('Testes utilizando as funções equal(...) e notEqual(...)', function(assert) {
   assert.expect(6);
```

```
    //Com equal()
    assert.equal(multiplicar(2, 2), 4, 'Multiplicação de dois números positivos');
    assert.equal(multiplicar(-2, -2), 4, 'Multiplicação de dois números negativos');
    assert.equal(multiplicar(-2, 2), -4, 'Multiplicação de um número negativo e um número positivo');
    assert.equal(multiplicar(2, 0), 0, 'Multiplicação de um número positivo e um número neutro');

    //Com notEqual()
    assert.notEqual(multiplicar(1, 1), 0, 'Multiplicação de dois número positivo iguais');
    assert.notEqual(multiplicar(2, 3), 0, 'Multiplicação de um número positivo e um número positivo');
});
```

No código apresentado podemos destacar também a importância da utilização da chamada "assert.expect(4)", no qual estamos dizendo ao QUnit que esperamos que os quatro testes sejam executados. É apenas uma prática recomendada para definir o número de afirmações que esperamos para serem executadas. Outro ponto é que a maioria das funções de afirmação do QUnit seguem o mesmo padrão, por exemplo:

```
assert.<funcao-afirmacao>(<valor calculado ou corrente>,<valor esperado>, <descrição opcional da afirma
assert.equal(multiplicar(2, 2), 4, 'Multiplicação de dois números positivos');
assert.notEqual(multiplicar(1, 1), 0, 'Multiplicação de dois números positivos');
```

Então para executamos o nosso teste basta acessar o arquivo index.html no navegador e, se tudo ocorreu como esperado, teremos o resultado semelhante à **Figura 2.**

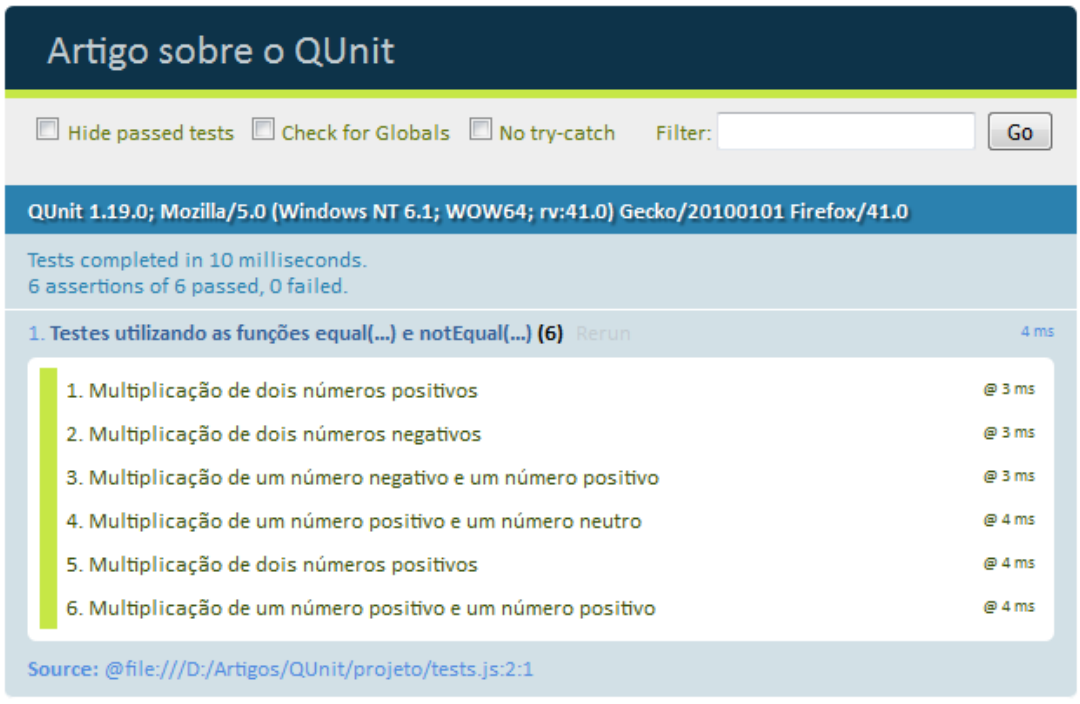

Figura 2. Resultado dos testes utilizando a função equal() e notEqual()

Agora criaremos testes unitários utilizando a função strictEqual() e notStrictEqual do QUnit. Essas duas funções são utilizadas quando requer uma comparação rigorosa (de forte tipagem). Adicione o código da **Listagem 5** no arquivo tests.js.

Listagem 5. Código do tests.js

```
QUnit.test('Testes utilizando as funções strictEqual(...) e notStrictEqual(...)', function(assert) {
    assert.expect(2);
    //Com strictEqual()
```

```
    assert.strictEqual(multiplicar(-2, 2), -4, 'Multiplicação de um negativo e um número positivo é ig
    //Com notStrictEqual()
    assert.notStrictEqual(multiplicar(-2, 2), '-4', 'Multiplicação de um negativo e um número positivc
});
```

Para executamos os testes basta acessar a index.html no navegador e, se tudo ocorreu como esperado, teremos o resultado igual a **Figura 3**.

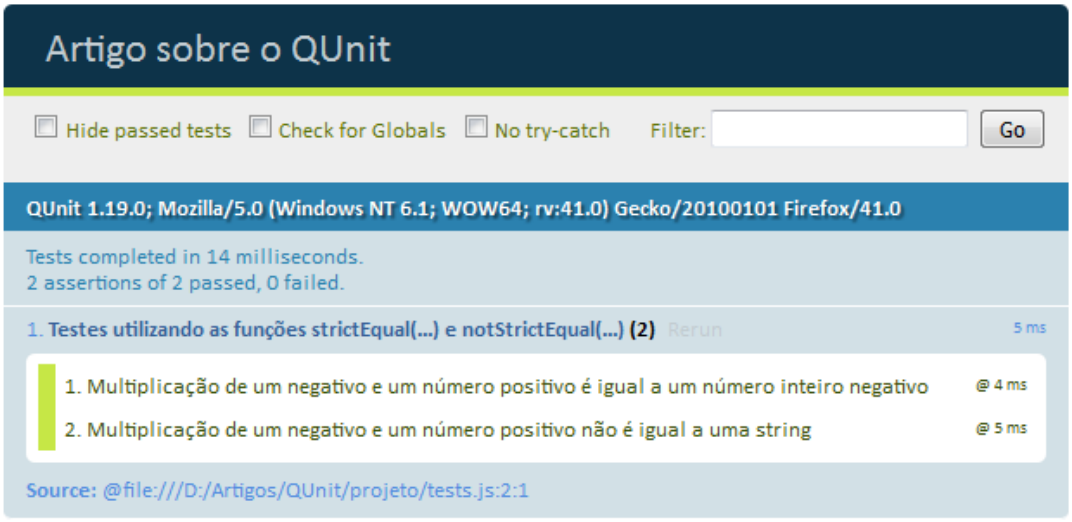

Figura 3. Resultado dos testes utilizando as funções strictEqual() e notStrictEqual()

Note que se substituirmos a função notStrictEqual() por equal() no exemplo anterior, o teste passa, pois o equal() não realiza uma comparação rigorosa (de forte tipagem ' ===’), diferentemente do strictEqual(), que usa.

Faremos agora alguns testes utilizando as funções propEqual() e notPropEqual(). Para isso, utilize o arquivo index.html e depois acrescente o código da **Listagem 6** dentro da tag <head>. Basicamente definimos uma função chamada “Pessoa” contendo duas propriedades e após esta função declaramos um objeto literal chamado “humano” contendo duas propriedades inicializadas com nulo. Essas duas definições utilizaremos nos testes.

Listagem 6. Código do arquivo index.html dentro da tag <head>

```
<script language="javascript">
//Função Pessoa com duas propriedades
    function Pessoa(nome,email) {
                        this.nome = nome
             this.email = email
    }

    //Objeto literal com duas propriedades inicializadas com nulo
    var humano = {
            nome: null,
            email: null
    };
 </script>
```

Em seguida, utilize o arquivo tests.js substituindo o teste já existente pela **Listagem 7**, onde faremos alguns testes utilizando as funções propEqual() e notPropEqual().

Listagem 7. Código do tests.js

```
QUnit.test('Testando utilizando a função propEqual(...)', function(assert) {
```

```
    assert.expect(1);
    // 1° Objeto
    var pessoa = new Pessoa('Brendo Felipe','brendo10x@gmail.com');
    // 2° Objeto
    humano.nome = 'Brendo Felipe';
    humano.email = 'brendo10x@gmail.com';

    assert.propEqual(pessoa, humano, 'Os dois objetos apresentam as mesmas propriedades e valores');

});

QUnit.test('Testando utilizando a função notPropEqual(...)', function(assert) {
    assert.expect(1);
    // 1° Objeto
    var pessoa = new Pessoa('Brendo Felipe','brendo10x@gmail.com');
    // 2° Objeto
    humano.nome = 'Fernando';
    humano.email = 'brendo10x@gmail.com';

    assert.notPropEqual(pessoa, humano, 'Os dois objetos apesar de terem as mesmas propriedades um del
});
```

Basicamente o que queremos testar na listagem apresentada é simplesmente se os dois objetos passam com a função propEqual() e notPropEqual() sem a restrição de protótipos (forma como os objetos JavaScript são criados). Para executamos o nosso teste basta acessar o arquivo index.html no navegador e, se tudo ocorreu como esperado, teremos o resultado semelhante à **Figura 4.**

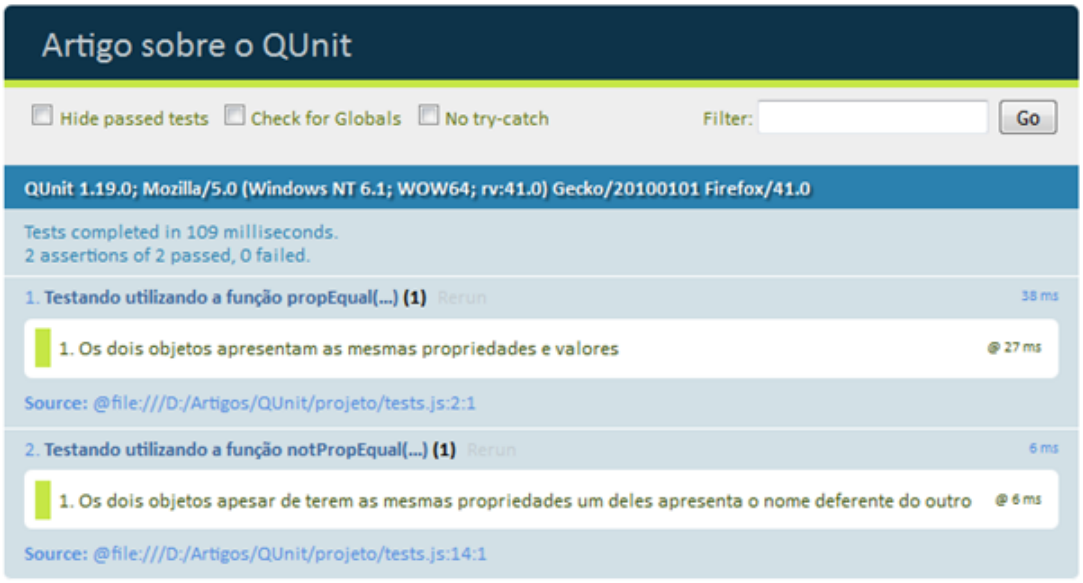

Figura 4. Resultado dos testes utilizando as funções propEqual() e notPropEqual()

Por fim, criaremos testes unitários utilizando as funções deepEqual() e notDeepEqual() do QUnit, que são utilizadas quando se requer uma comparação rigorosa e que leve em consideração o protótipo dos objetos.

Para o novo exemplo adicione o código da **Listagem 8** no arquivo tests.js, lembrando que nesses testes utilizaremos a **Listagem 6** também.

Listagem 8. Código do tests.js

```
QUnit.test('Testando utilizando a função deepEqual(...)', function(assert) {
    assert.expect(1);
```

```
    // 1° Objeto
    var pessoa = new Pessoa('Brendo Felipe','brendo10x@gmail.com');
    // 2° Objeto
    var pessoa2 = new Pessoa('Brendo Felipe','brendo10x@gmail.com');

    assert.deepEqual(pessoa, pessoa2, 'Os dois objetos apresentam as mesmas propriedades, valores e pr

});

QUnit.test('Testando utilizando a função notDeepEqual(...)', function(assert) {
    assert.expect(1);
    // 1° Objeto
    var pessoa = new Pessoa('Brendo Felipe','brendo10x@gmail.com');
    // 2° Objeto
    humano.nome = 'Brendo Felipe';
    humano.email = 'brendo10x@gmail.com';

    assert.notDeepEqual(pessoa, humano, 'Os dois objetos apresentam as mesmas propriedades e valores,

});
```

O que queremos testar é a verificação entre os dois objetos para ver se eles passam com as funções deepEqual() e notDeepEqual() com a restrição de protótipos (forma como os objetos JavaScript são criados). Após executarmos o teste, se tudo ocorreu como esperado, teremos o resultado semelhante à **Figura 5**.

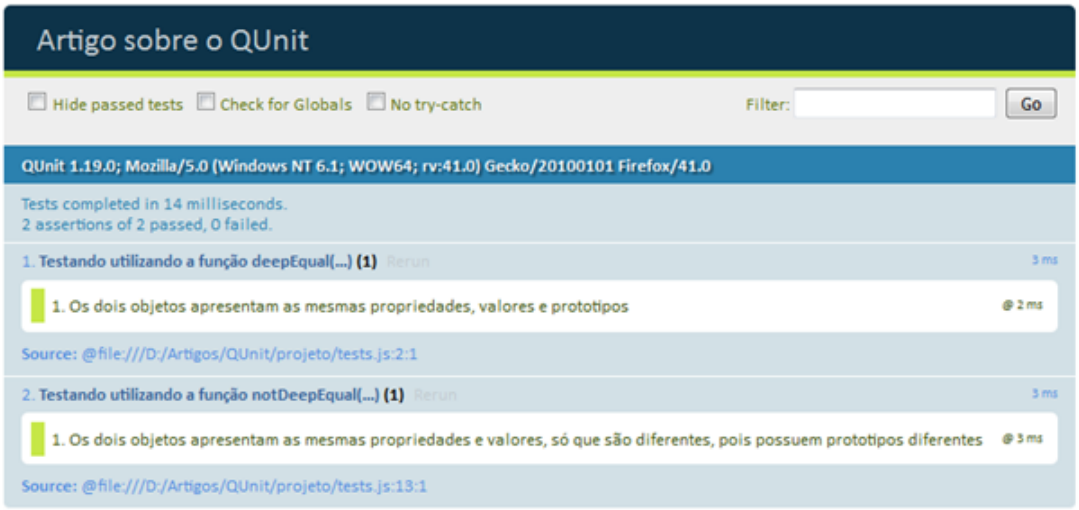

Figura 5. Resultado dos testes utilizando as funções deepEqual() e notDeepEqual()

Note que se substituirmos a função notDeepEqual() por propEqual() no exemplo anterior o teste passa também, pois o propEqual() não leva em consideração a forma como o objeto é criado (protótipo), diferentemente do deepEqual() ou notDeepEqual().

Trabalhando com exceções nos testes com QUnit

Agora que conhecemos as funções de comparação do QUnit, nesta seção iremos conhecer a função **throws(),** que basicamente é utilizada para verificar se durante os testes houve algum lançamento de exceção esperada. Vamos a um exemplo criando um teste onde, durante a sua execução, será lançada uma exceção esperada. Para isso, adicione o código da **Listagem 9** dentro da tag <head> na index.html.

Listagem 9. Código da tag <head> na index.html

```
<script language="javascript">
function verificaSeONumeroEPar(numero) {
            if (typeof numero !== 'number') {
               throw new Error('O argumento (numero) passado por parâmetro não é um número');
            }

            return numero % 2 === 0;
    }
 </script>
```

Basicamente o código contém uma função chamada "verificaSeONumeroEPar", que realiza no início da linha a checagem para ver se o tipo passado por parâmetro "numero" é realmente o número: se não for, lança uma exceção "new Error", mas se for um número, retorna um boolean da verificação se o número informado for par.

Em seguida, utilize o arquivo tests.js substituindo o teste já existente pela **Listagem 10**, onde faremos o uso do throws().

Listagem 10. Código do tests.js

```
QUnit.test('Trabalhando com exceções utilizando o throws(...)', function(assert) {
   assert.expect(1);

   assert.throws(
            function() {
               verificaSeONumeroEPar('teste');
            },
            new Error('O argumento (numero) passado por parâmetro não é um número'), 'Passando uma st
    );

});
```

Observe que na listagem anterior a função throws precisa dos seguintes parâmetros:

```
assert.throws(<A função a ser executada>, <Erro esperado>, <Uma descrição opcional>)
```

Com isso, estamos passando no primeiro parâmetro a função a ser executada chamada "verificaSeONumeroEPar" com o argumento do tipo string, forçando que a exceção ocorra. No próximo argumento declaramos o erro ou exceção esperada durante o teste. Por fim, no último parâmetro definimos uma pequena descrição. Para executamos o teste basta acessar a index.html no navegador. Se tudo ocorreu como esperado, teremos o resultado igual a **Figura 6**.

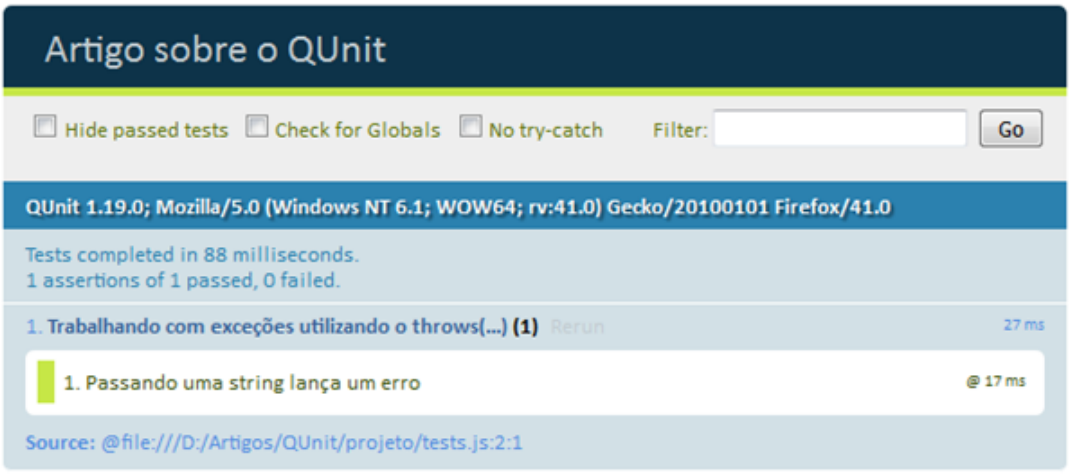

Figura 6. Resultado do teste utilizando a função throws()

Criando testes assíncronos com QUnit

Até agora os nossos testes foram executados de forma síncrona, e nesta seção faremos na forma assíncrona. Em cada projeto escrito em JavaScript tem-se funções assíncronas, que são usadas para executar uma determinada ação após um certo período de tempo, para recuperar dados a partir de um servidor, ou evento para enviar dados para um servidor. Diante disso, o QUnit fornece uma função chamada "async()", com a qual podemos realizar chamadas assíncronas nos testes.

Vamos ao seguinte exemplo: criaremos um teste onde há quatro afirmações (testes simples) e cada uma terá que testar e verificar qual é o maior número dentre os informados e o resultado esperado. Logo, cada afirmação será iniciada a partir de um determinado tempo (em segundos), por exemplo. Uma vez iniciado o teste geral, a segunda afirmação será executada quando passar três segundos. Para isso, acrescente o código da **Listagem 11** dentro da tag <head> na index.html.

Listagem 11. Código da tag <head> na index.html

```
  <script language="javascript">
 function obterNumeroMaior() {
   var numeroMaior = -Infinity;
   for (var i = 0; i < arguments.length; i++) {
            if (arguments[i] > numeroMaior) {
            numeroMaior = arguments[i];
            }
   }

   return numeroMaior;
}
  </script>
```

Observe que definimos uma função chamada "obterNumeroMaior()", que recebe um número indeterminado de valores por parâmetro, e assim a função retorna o maior valor (número) entre os informados. Em seguida, utilize o arquivo tests.js substituindo o teste já existente pela **Listagem 12**, onde faremos o uso do async().

Listagem 12. Código do tests.js

```
QUnit.test('Testes assíncronos com QUnit', function (assert) {
   expect(4);

  var async1 = assert.async();
   window.setTimeout(function() {
            assert.strictEqual(obterNumeroMaior(), -Infinity, 'Nenhum parâmetro'); // 4° teste que se
            async1(); // chamada assíncrona
   }, 4000); // Será executado quando for 4 segundos

            var async2 = assert.async();
   window.setTimeout(function() {
            assert.strictEqual(obterNumeroMaior(1, 1, 2), 2, 'Todos os números positivos');  // 3° te
            async2(); // chamada assíncrona
   }, 3000); // Será executado quando for 3 segundos

   var async3 = assert.async();
   window.setTimeout(function() {
            assert.strictEqual(obterNumeroMaior(-64, 5, 3, 23), 23, 'Números positivos e negativos');
            async3(); // chamada assíncrona
   }, 2000); // Será executado quando for 2 segundos
```

```
    var async4 = assert.async();
    window.setTimeout(function() {
            assert.strictEqual(obterNumeroMaior(-11, -1, -5), -1, 'Todos os números negativos');  //
            async4();
    }, 1000); // Será executado quando for 2 segundos
});
```

Observe que em cada afirmação é incorporada pela função "setTimeout(...)", e com isso estamos dizendo ao QUnit que cada afirmação será executada em um determinado tempo. Outro ponto a destacar é que estamos fazendo uso da função "async()" em cada afirmação, e com isso estamos dizendo ao QUnit que cada teste realiza uma operação assíncrona, e assim o QUnit executa.

Para executar os testes da **Listagem 12**, basta acessar a index.html no navegador. Se tudo ocorreu como esperado, teremos o resultado semelhante à **Figura 7**.

Figura 7. Resultado dos testes fazendo uso da função async()

Organizando os testes utilizando módulos com QUnit

Conforme os testes são criados o código fonte aumenta de tamanho e com isso podemos dividir o código em módulos para aumentar a capacidade de manutenção. O QUnit tem um método simples para agrupar os testes em módulos chamado "QUnit.module()" que tem a seguinte sintaxe:

```
QUnit.module(<Nome do módulo>);
```

Vejamos um exemplo de como utilizá-lo acrescentando o código da **Listagem 13** no arquivo tests.js. Neste teste faremos uso da função "module(...)" para a criação de dois grupos de teste.

Listagem 13. Código do tests.js

```
QUnit.module('1° Grupo de teste');
QUnit.test('Primeiro teste', function(assert) {
    assert.expect(1);
    assert.ok(true);
});
```

```
QUnit.test('Segundo teste', function(assert) {
   assert.expect(1);
   assert.ok(true);
});

QUnit.module('2° Grupo de teste');
QUnit.test('Primeiro teste', function(assert) {
   assert.expect(1);
   assert.ok(true);
});
QUnit.test('Segundo teste', function(assert) {
   assert.expect(1);
   assert.ok(true);
});
QUnit.test('Terceiro teste', function(assert) {
   assert.expect(1);
   assert.ok(true);
});
```

Observe que estamos definindo dois grupos de teste: o primeiro grupo contém dois testes e o segundo grupo contém três testes.

Para executamos o teste da **Listagem 13** basta acessar a index.html e, se tudo ocorreu como esperado, teremos o resultado semelhante à **Figura 8**.

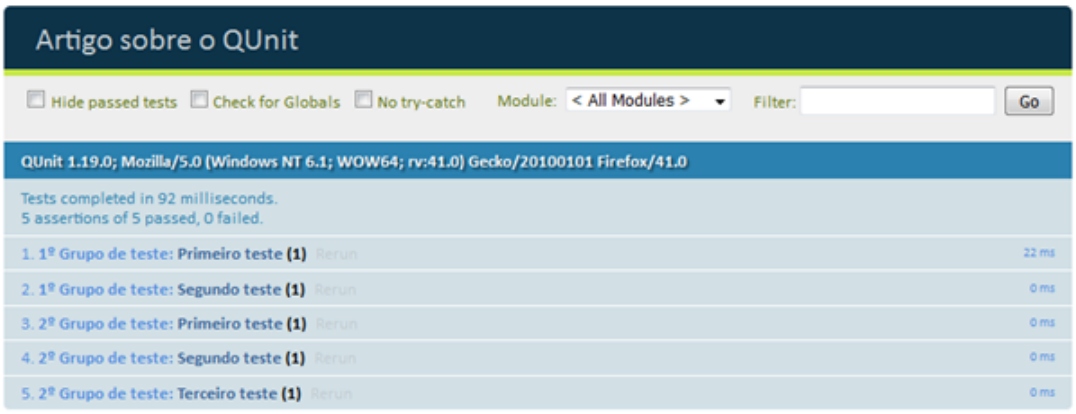

Figura 8. Resultado dos testes fazendo uso da função module(...);

Note que quando fazemos o uso da função "module()" podemos selecionar no drop-down exibido no canto superior direito da página um módulo específico para executar os testes.

Com isso, neste artigo vimos como trabalhar com as principais funcionalidades do framework de testes unitários JavaScript QUnit.

Obrigado e até mais!

Bibliografia

BIBEAULT, Bear; KATZ, Yehuda; ROSA, Aurelio De. **JQuery in Action, Third Edition.** United States: Manning Publications, 2015.

SHEIKO, Dmitry. **Instant Testing with QUnit.** Birmingham: Packt Publishing, 2013.

EQUIPE QUNIT **QUnit:** Js unit testing.
http://qunitjs.com/

Brendo Felipe Rodrigues Arcanjo