TRYBE
Modulo II – Front-end

Bloco 9 – Jest & Javascript assíncrono

Definição
Jest é um framework de testes unitários para JavaScript desenvolvido pelo Facebook.

Concorrentes
Mocha e Jasmine

Vantagens
- capaz de encontrar e executar automaticamente todos os testes da aplicação;
- reportar informações sobre cobertura de testes;
- fornecer feedback sobre quais testes falharam e por quê;
- assert insuficiente no mercado;
- rápido e integrável com React.

Instalar

Ref: https://medium.com/jaguaribetech/testando-seu-javascript-com-jest-de2a4674f4ad
→ Arquivo package.json

```
{
  "scripts": {
    "test": "jest"
  }
}
```

→ terminal *npm install --save-dev jest*

Escrevendo testes

Organização
→ Arquivo do código
module.exports = functionname exporta a função para que possa ser utilizada em outros módulos.
→ Arquivo separado .test.js
const nameconst= require('./pathtotestedcode')
Jest procura por arquivos com as extensões .js, .jsx, .ts e .tsx dentro de uma pasta com o nome __tests__, ou arquivos com o sufixo .test ou .spec.

test
Função a ser utilizada para testar. A função *it* é um alias para test.
Três argumentos: *nome do teste escolhido, expectation, timeout.*

.skip
- Pular testes com cada test.skip
- Pular vários testes:

```
describe.skip('Suíte de Testes', () => {
testes a serem pulados
}
```

Expect e matchers

Função expect é utilizada para dar acesso a um conjunto de **métodos chamados matchers** para **testar valores.**
Documentação completa Jest de matchers: https://jestjs.io/docs/en/expect

matcher toBe
Testa igualdade estrita.
Exemplo de equivalente Jest comparado com assert:

```
test("it's defined", () => {
  expect(typeof convertToRoman).toBe('function');
})

assert.equal(typeof convertToRoman, 'function');
```

matcher toEqual
Para testar igualdade de objetos e arrays: acessa cada elemento do objeto ou array.

Matchers para Valores booleanos
Para testar igPara fazer distinção entre valores falsy (null, undefined e false).

- `toBeNull` matches only `null`
- `toBeUndefined` matches only `undefined`
- `toBeDefined` is the opposite of `toBeUndefined`
- `toBeTruthy` matches anything that an `if` statement treats as true
- `toBeFalsy` matches anything that an `if` statement treats as false

Números
Há matchers para as principais formas de comparar números:

```
test('dois mais dois', () => {
  const value = 2 + 2;
  expect(value).toBeGreaterThan(3);
  expect(value).toBeGreaterThanOrEqual(3.5);
  expect(value).toBeLessThan(5);
  expect(value).toBeLessThanOrEqual(4.5);
```

Strings
.toMatch(regexpOrString)

Arrays
Para verificar se um array contém um item em particular:
.toContain
Para estrutura mais complexa:
.toContainEqual
.toHaveLength

Objetos
.toMatchObject(object)
Para testar se um objeto possui uma propriedade específica:
.toHaveProperty

Excepções

.toThrow testa que uma função lança um erro quando é executada e é preciso envolver o código em uma função (normalmente uma arrow function).

not

Permite testar o oposto de algo.

```
test('Sunday is not a workday', () => {
  expect(workdays).not.toContain('Sunday');
});
```

O bloco describe

A função describe cria um **bloco para agrupar vários testes**.
Vantagens: organizar, pode ser arbitrariamente, definir bloco.

Dicas diversas

charAt https://www.w3schools.com/JSREF/jsref_charat.asp
–
Bom resumo dos matchers: https://github.com/sapegin/jest-cheat-sheet
–
Doc oficial: https://jestjs.io/ .
–
Npm é iniciador de pacotes do Node. Reaproveita code jà pronto. A idéia aqui é que importamos Jest dentro de nosso code.
–
Verificar tipos de dados

```
const convertToRoman = integer => {
  if(typeof integer !== 'number') {
    throw new Error('Argument must a number');
  }
```

```
describe('convertToInteger', () => {
  test("it's defined", () => {
    expect(typeof convertToInteger).toBe('function');
  });

  test('raises an error if argument is not a string', ()
  => {
    expect(() => { convertToInteger(1234) })
      .toThrow(/Argument must be a string/);
  });

  test('raises an error if argument contain invalid
  characters', () => {
    expect(() => { convertToInteger('XXIT') })
      .toThrow(/Argument can't contain invalid roman
      symbols/);

    expect(() => { convertToInteger('ZVLD') })
      .toThrow(/Argument can't contain invalid roman
      symbols/);
  });
```

–
watch para basicamente visualizar ao vivo o test

```
  "scripts": {
    "test": "jest --watch"
  },
  "devDependencies": {
    "jest": "^26.1.0"
  }
}
```

---

2) JavaScript Assíncrono e Callbacks

Javascript permite programação assíncrona, ou seja onde A B C não precisam serem executadas na mesma ordem e esperando umas às outras.
Uso: garantir que as functions sejam executadas na ordem mais correta.

Operações assíncronas

setTimeout

```
setTimeout(function() {
  console.log("Water plant")
},3000);
```

//3000 ms = 3 segundos

```
Math.random() * 2000);
```

//Para passar tempo aleatório

Sintax
setTimeout(element, time);

Callbacks

Definição
Callback é uma *função passada como parâmetro para outra função*. (Lembrando que *funções são super objetos* e que por isso podem: ser tratadas como valor, ter outras functions dentro, retornar functions a serem chamadas mais tarde). Assim que callback nem sempre é assincrono! //esse exemplo é sincrono →

```
function greeting(name) {
  alert('Hello ' + name);
}

function processUserInput(callback) {
  var name = prompt('Please enter your name.');
  callback(name);
}

processUserInput(greeting);
```

*"The function you're passing as an argument is called a **callback** function and the function you're passing the callback function to is called a **higher order function**."*

Sintax setInterval

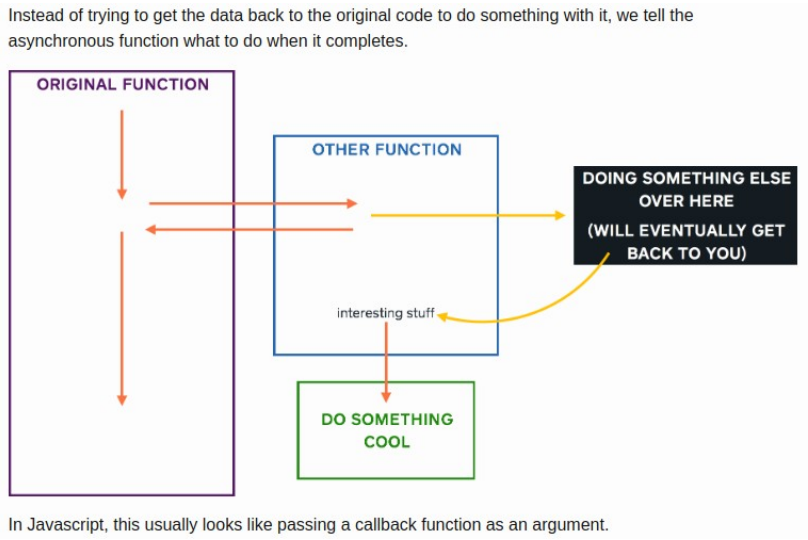
```
setInterval(function() {
  console.log('hello!');
}, 1000);
```

2 parâmetros: *(função callback, tempo que o interpretador irá esperar até executar essa função).*
setInterval(callback, time);

Uso do callback
- Function a ser chamada depois de concluir a function superior.

```
const fun1 = (callback) => {
  setTimeout(() => {
    console.log('Função 1')
    callback();
  }, Math.random() * 2000);
```

- Pode ter outro nome: callback é apenas a forma como estruturamos, definimos a sequência desejada de um code assincrono.



Método getJSON do jQuery
Para aumentar o prazo de execução de uma função até termos os dados que precisamos:
Dois parâmetros: *(URL ou seja jeito de acessar a API, callback).*

```
// updateUI and showError are irrelevant.
// Pretend they do what they sound like.

const id = 'tylermcginnis'

$.getJSON({
  url: `https://api.github.com/users/${id}`,
  success: updateUI,
  error: showError,
})
```

Lidando com erros nas operações assíncronas

Possíveis causas externas de erro
- falha de conexão
- requisição de API não funciona
Resposta adequada → *mensagem de erro.*

Resumindo o dia:

```
## JavaScript Síncrono

  - Forma linear, tradicional em que o código é executado, linha-a-linha. Execução sequencial.

## JavaScript Assíncrono

  - É a forma não linear em que o código funciona independentemente da ordem em que ele é escrito.

## Callbacks

  - É a forma que usamos para definirmos uma sequencia para um código assíncrono.
```

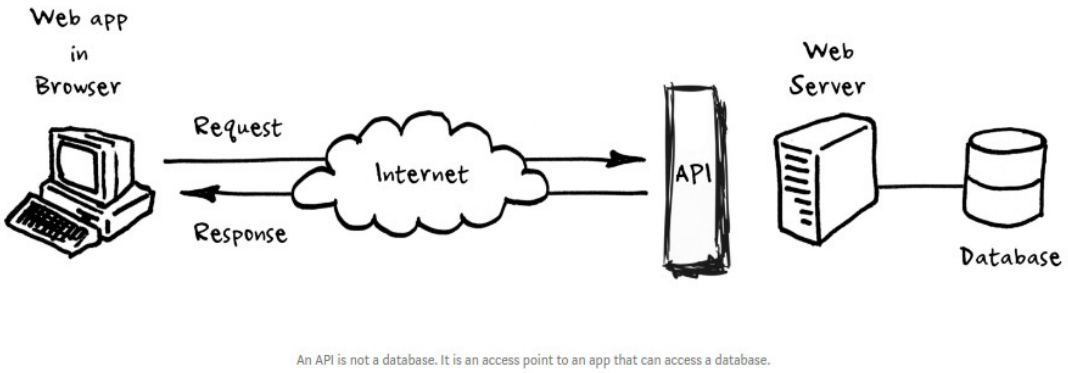
---

3) JavaScript Promises

Criadas para facilitar a leitura de códigos assíncronos e tornar sua lógica mais previsível e legível. Principal forma de você se comunicar com APIs.

API

Application Programming Interface (API)
APIS permitem a comunicação entre aplicações. APIS da web retornam dados em resposta a uma solicitação feita pelo cliente. Permitem **pegar dados de fontes externas**.



An API is not a database. It is an access point to an app that can access a database.

- Quem faz: geralmente grandes empresas e forças púbiacas, mas pode ser qualquer pessoa;
- APIS têm rotas na internet do mesmo jeito que sites têm rotas.
- **JSON** (JavaScript Object Notation) é um jeito de representar dados, parecido com objetos Js.

Promises

Porque precisamos das promises: problemas do callback
- *pouco legível* (todo aninhado, o humano prefere pensar sequencialmente)
- *difícil manutenção*,
- *controle transferido* para terceiros da chamada da function:

```
function criticalFunction () {
  // It's critical that this function
  // gets called and with the correct
  // arguments.
}

thirdPartyLib(criticalFunction)
```

Três estados do promise
pending, fulfilled ou rejected, que representam o estado de uma solicitação asincrona.

Sintax
```
const promise = new Promise((resolve, reject) => {
  //asynchronous code goes here
});
```

Tem dois callbacks como parâmetros, ***resolve e reject.***
Resolve é uma function que muda o estatuto da promise para fulfilled; *reject* para rejected.
Ou seja são dois planos de ação caso a requisição der certo ou der errado.

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve() // Change status to 'fulfilled'
  }, 2000)
})
```

.then & .catch
Para ouvir a mudança de estatuto do Promise.

```
getRandomJoke("spider")
  .then(() => getRandomJoke("ghost"))
  .then(() => getRandomJoke("pizza"))
  .then(() => getRandomJoke("wolf"))
  .then(() => getRandomJoke("ant"))
```

Métodos .then() e .catch() sempre retornam uma promise.

```
function onSuccess () {
  console.log('Success!')
}

function onError () {
  console.log('💩')
}

const promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    reject()
  }, 2000)
})

promise.then(onSuccess)
promise.catch(onError)
```

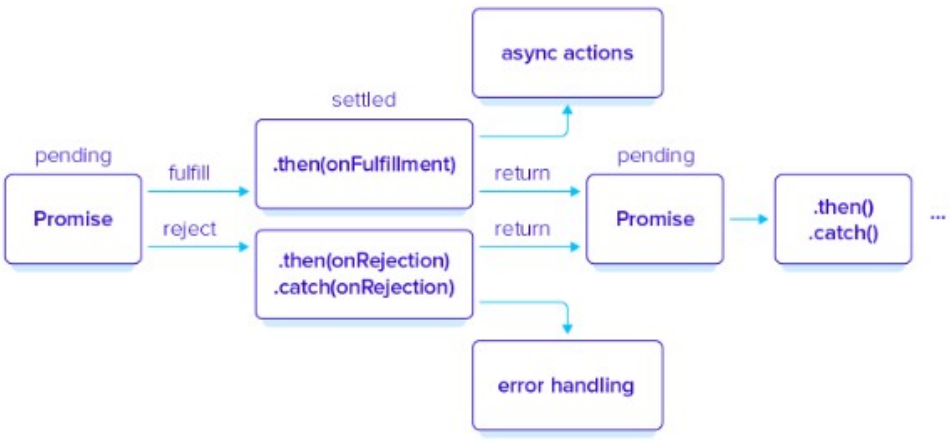
.then organiza a ordem de execução desejada em caso de estatuto fulfilled da promise.
.then() pode aceita dois callbacks: o primeiro invocado quando o promise é resolvido, o segundo quando é rejeitado. Porém, a melhor prática é usar .catch em caso de estatuto rejected da promise.

.catch libera o cenário previsto em caso de rejected.
Seria um equivalente de .then(undefined, onRejected).

```
let promiseDoHomework = new Promise(function(resolve, reject) {
    //code to do homework...
    let homeWorkDone = true;
    if (homeWorkDone)
        resolve('ALL DONE!');
    else
        reject('NO CAN DO!');
});


promiseDoHomework.then(function(resolveReturn) {
    console.log("Homework done! Return: " + resolveReturn);
 }).catch(function(rejectReturn){
     console.log("Homework not done! Return: " + rejectReturn);
 });
```

<u>Resumindo o fluxo da Promise:</u>



Fetch

Fetch é uma function base poderosa para fazer **requisição de HTTP** no Javacript, utilizada para fazer chamadas às URL's das APIs. Trata-se de uma função assíncrona, baseada em uma promise.

Sintax

fetch(API_URL, myObject)

Tem dois parâmetros: *(URL do serviço alvo da API, objeto com informações sobre requisição de API contendo chaves). É encadenado com then.*

```
fetch(API_URL, myObject)
  .then(response => response.json())
  .then(data => console.log(data));
```

*//Método `.json()` na resposta da *API* converte o conteúdo do `body` da *Response* e retorna uma outra *Promise*, que, quando bem-sucedida, retorna um *JSON* contendo as informações do API.*

```
function findLyrics(artist, song) {
    return fetch(`https://api.lyrics.ovh/v1/${artist}/${song}`);
}
```

//Outro exemplo de uso com param usados na URL

```
const getPosts = () => {
  return fetch(`http://jsonplaceholder.typicode.com/posts`)
    .then(res => res.json())
    .then(posts => console.log(posts))
}
```

//Fluxo: function promise que navega até aquela API, retorna o json, visualiza o objeto no console. APENAS uma requisição, não tem modificação.

async await function

Outro modo de escrever Promise. Permite usar function asincrona como se fosse sincrona.

```
async function getJokes () {
  try {
    await getRandomJoke("ghost")
    await getRandomJoke("pizza")
    await getRandomJoke("wolf")
    await getRandomJoke("ant")
  } catch (error) {
    console.log(error)
  }
}

getJokes();
```

Async é uma palavra-chave do Javascript que indica que dentro daquela function vão acontecer invocações assincronas de functions. *Implicitamente, async function sempre retorna uma promise.*

Await é outra palavra-chave que fala basicamente "ok, essa function aqui é assincrona e retorna uma promise, assim que por favor em vez de continuar como sempre faz, espera o valor da promise e retorna ele antes de continuar". Não pode ser usada fora de uma função *async*.

A function também inclui dois blocos ***try & catch*** para retornar funcionamento ou erro.

Dicas diversas

–

Ternary com undefined para “n fazer nada” em caso de condição não satisfeita.

```
callback ? callback() : undefined
```

–

“Metódos fetch, getJson, setTimeout/interval e chamadas a API são assincronos.
Callbacks nem necessariamente. Código síncrono pode ser atrelado a código assíncrono.”

–

No terminal:
curl -H "Accept: text/html" "https://icanhazdadjoke.com/" retorna html
curl -H "Accept: text/plain" "https://icanhazdadjoke.com/" retorna uma piada
curl -H "Accept: application/json" "https://icanhazdadjoke.com/" retorna um objeto JSON

PROJETO FINAL: Shopping cart

// com fetch, é preciso visualizar no console (do site correspondendo com o html).

// Diferença then / await: then espera o que é antes, await espera o que é escrito depois.

// parseFloat(string)
https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/parseFloat

–

Para fazer aparecer na página, function onload do browser

```
window.onload = () => functionname();
```