**Javascript**

É uma linguagem de programação que permite a você criar conteúdo que se atualiza dinamicamente, controlar multimídias, imagens animadas, e tudo o mais que há de interessante. Ok, não tudo, mas é maravilhoso o que você pode efetuar com algumas linhas de código JavaScript. Para criar um arquivo basta adicionar .js no final.

•Variáveis

Variáveis em javascript não necessitam de informar o seu tipo porem existem 4 tipos eles são String, Number, Boolean e Undefined.

Para informar uma variável usamos o comando let ou var:

let name = "Maria Jones";

var age = 21;

let hasDriverLicense = true;

var empty = undefined;

Para exibir no terminal:

console.log (name)

Uma variável string normalmente tem letras números e símbolos, variável do tipo number pode ser inteiro, decima, fracionário, boolean que é true or false e undefined que não tem um tipo.

Para informar o tipo da variável basta adicionar o comando typeof:

console.log(name + “=” + typeof name)

•Objetos

Objetos em javascript são uma coleção de dados ou funcionalidades que estão relacionadas. Para mostrar um objeto em javascript fazemos:

let pessoa = {}

Se você der um console.log(pessoa) o terminal deve te mostrar um resultado assim [object Object].

Agora vamos colocar mais alguns dados dentro do objeto:

let pessoa = {

nome: ‘João das Neves’,

idade: 23,

classe: “Guerreiro”,

força: 25

}

Para mostrar todos os dados incluindo o nome dos objetos usamos a seguinte função:

console.log(JSON.stringify(pessoa))

Dentro do terminal ele irá mostrar os dados assim:

{“nome”: João das Neves,” idade”: 23, “classe”: Guerreiro, “força”: 25}

Agora caso você queira informar um dado especifico dentro do seu objeto usamos da seguinte forma [nome da variável que armazena o objeto].[propriedade dentro do objeto]

console.log(pessoa.nome)

No terminal isso irá te mostrar só o valor do objeto pessoa.nome no caso:

João das Neves

Tá, mas e se você quiser mostrar todos os dados sem informar a propriedade que armazena o valor. Simples usamos a seguinte expressão Object.values([nome do objeto]).

console.log(Object.values(pessoa))

Isso mostrará apenas os valores das propriedades dentro do objeto no terminal ficará assim:

João das Neves,23, Guerreiro,25

Em objetos também podemos inserir objetos dentro deles um exemplo:

let pessoa = {

nome: ‘João das Neves’,

idade: 23,

classe: “Guerreiro”,

status:{

força: 25

agilidade: 20

inteligência: 15

}

}

Usando o operador delete você pode delatar uma propriedade de um objeto:

delete pessoa.classe

Remove a propriedade classe do objeto pessoa do exemplo acima.

•Objetos Avançados

•Privacidade

Acessar e atualizar propriedades é fundamental para o trabalho com objetos. No entanto, há casos em que não queremos que outro código simplesmente acesse e atualize as propriedades de um objeto. Em vez disso, os desenvolvedores de Javascript seguem convenções de nomenclatura que sinalizam para outros desenvolvedores como interagir com uma propriedade. Uma convenção comum é colocar um sublinhado \_ antes do nome de uma propriedade para significar que a propriedade não deve ser alterada.

const bankAccount = { \_amount: 1000 }

•Getters

Getters são métodos que obtêm e retornam as propriedades internas de um objeto. Mas eles podem fazer mais do que apenas recuperar o valor de uma propriedade.

const robot = {

\_model: '1E78V2',

\_energyLevel: 100,

get energyLevel(){

if(typeof this.\_energyLevel === 'number'){

return `My current energy level is ${this.\_energyLevel}`

}else{

return `System malfunction: cannot retrieve energy level`

}

}

};

console.log(robot.energyLevel)

Agora que analisamos a sintaxe, vamos discutir algumas vantagens notáveis ​​do uso de métodos getter:

•Os getters podem executar uma ação nos dados ao obter uma propriedade.

•Getters podem retornar valores diferentes usando condicionais.

•Em um getter, podemos acessar as propriedades do objeto de chamada usando this.

•A funcionalidade do nosso código é mais fácil para outros desenvolvedores entenderem.

Outra coisa a ter em mente ao usar os métodos getter (e setter) é que as propriedades não podem compartilhar o mesmo nome que a função getter / setter. Se fizermos isso, chamar o método resultará em um erro infinito na pilha de chamadas. Uma solução alternativa é adicionar um sublinhado antes do nome da propriedade, como fizemos no exemplo acima.

•Setters

Juntamente com os métodos getter, também podemos criar métodos setter que reatribuem valores de propriedades existentes dentro de um objeto.

const person = {

\_age: 37,

set age(newAge){

if (typeof newAge === 'number'){

this.\_age = newAge;

} else {

console.log('You must assign a number to age');

}

}

};

person.age = 40;

console.log(person.\_age); // Logs: 40

person.age = '40'; // Logs: You must assign a number to age

•Function Factory

Até o momento, criamos objetos individualmente, mas há momentos em que queremos criar muitas instâncias de um objeto rapidamente. É aqui que entram as funções da fábrica. Uma fábrica do mundo real fabrica várias cópias de um item rapidamente e em grande escala. Uma função de fábrica é uma função que retorna um objeto e pode ser reutilizada para criar várias instâncias de objeto. As funções de fábrica também podem ter parâmetros que permitem personalizar o objeto que é retornado.

const monsterFactory = (name, age, energySource, catchPhrase) => {

return {

name: name,

age: age,

energySource: energySource,

scare() {

console.log(catchPhrase);

}

}

};

Na monsterFactory função acima, tem quatro parâmetros e devolve um objeto que tem as propriedades: name, age, energySource, e scare(). Para criar um objeto que representa um monstro específico como um fantasma, podemos chamar monsterFactory com os argumentos necessários e atribuir o valor de retorno a uma variável:

const ghost = monsterFactory('Ghouly', 251, 'ectoplasm', 'BOO!');

ghost.scare(); // 'BOO!

•Arrays

Arrays são geralmente descritas como "lista de objetos"; elas são basicamente objetos que contem múltiplos valores armazenados em uma lista. Um exemplo de array:

let nome = [“Gabriel”,”Rafael”,”Miguel”]

Um array geralmente é salvado por um tipo de dado, mas você pode colocar vários tipos dentro dele, porém por boa pratica é bom manter um array com um tipo de dado. Exemplo:

let nomes = [“Rafael”,” Juliano”,” Daniel”] Tipo: String

let idade = [21,23,29] Tipo: Number

Dentro de array também podemos colocar objetos e dados do tipo undefined.

Para encontrar o tamanho de um array caso um array seja grande usamos a função:

console.log(nomes.lenght)

Um dado dentro do array sempre começa da posição 0 . Exemplo neste array com 3 dados dentro o primeiro dado está na posição 0, o segundo na posição 1 e o terceiro na posição 2:

let nome = [“Gabriel”,”Rafael”,”Miguel”]

Agora se usar este comando console.log(nome[1]) ele irá informar o dado que está na posição 1 que no caso seria “Rafael”.

Para mostrar ou informar os dados dentro do array por completo em forma de loops usamos a seguinte forma que mais pra frente iremos abordar melhor esses são os três tipos mais comum de loops para arrays

for (let n of nome) {

console.log(n)

}

for (let i=0; I < nome.lenght; i++) {

console.log(nome[i])

}

nome. forEach (function (n, posicao) {

console.log (posicao + “-” + n)

})

•Operadores Aritméticos

Em javascript usamos cinco operadores aritméticos eles são:

+ soma let add = 2 + 2 4

- subtração let sub = 3 – 2 1

\* multiplicação let mult = 3 \* 4 12

/ divisão let div = 10 / 2 5

% resto let rest = 10 % 4 2

•Funções

Funções são blocos de construção fundamentais em JavaScript. Uma função é um procedimento de JavaScript - um conjunto de instruções que executa uma tarefa ou calcula um valor.

A definição da função (também chamada de declaração de função) consiste no uso da palavra-chave function, seguida por:

- Nome da Função.

- Lista de argumentos para a função, entre parênteses e separados por vírgulas.

- Declarações JavaScript que definem a função, entre chaves { }.

function soma (num1,num2){

return num1 + num2

}

console.log(soma(10,20))

•Laços

Existem várias formas diferentes de laços, mas eles essencialmente fazem a mesma coisa: repetir uma ação múltiplas vezes (inclusive você poderá repetir 0 vezes). Os vários mecanismos diferentes de laços oferecem diferentes formas de determinar quando este irá começar ou terminar. Há várias situações em que é mais fácil resolver um problema utilizando um determinado tipo de laço do que outros.

Os possíveis laços de repetição em JavaScript:

•for\_statement

Um laço for é repetido até que a condição especificada seja falsa. O laço for no JavaScript é similar ao Java e C. Uma declaração for é feita da seguinte maneira:

for ([expressaoInicial]; [condicao]; [incremento]){

declaracao

}

Exemplo:

for(let i=0; i<10;i++){

console.log(i)

}

No terminal irá mostrar os números de 0 a 9.

•do...while\_statement

A instrução do...while repetirá até que a condição especificada seja falsa.

do{

declaracao

}while (condicao);

Exemplo:

do{

let i=0;

i++

console.log(i)

}while(i<10)

•while\_statement

Uma declaração while executa suas instruções, desde que uma condição especificada seja avaliada como verdadeira. Segue uma declaração while:

while (condicao){

declaracao

}

Exemplo:

while(i<10){

i++

let i=0;

console.log(i)

}

•break\_statement

Use break para terminar laços, switch, ou um conjunto que utiliza label.

while (condicao){

declaracao

if(condicao){

break;

}

}

Exemplo:

while(i<10){

i++

let i=0;

if(i== 3){

break;

}

console.log(i)

}

•continue\_statement

A declaração continue pode ser usada para reiniciar uma instrução while, do-while, for, ou label.

while (condicao){

declaracao

if(condicao){

continue;

}

}

Exemplo:

let i=0;

while(i<10){

i++

if(i== 3){

continue;

}

console.log(i)

}

•for...in\_statement

A declaração for...in executa iterações a partir de uma variável específica, percorrendo todas as propriedades de um objeto.

Para cada propriedade distinta, o JavaScript executará uma iteração. Segue a sintaxe:

for (variavel in objeto) {

declaracoes

}

Exemplo:

let notas = {a:1,b:2,c:3}

for (let i in notas) {

console.log( i +":"+ notas[i])

}

•for...of\_statement

A declaração for...of cria uma laço com objetos interativos ((incluindo, Array, Map, Set, assim por conseguinte ), executando uma iteração para o valor de cada propriedade distinta.

for (variavel of objeto) {

declaracoes

}

Exemplo:

let n = [3,5,7]

for (let I of n) {

console.log(i)

}

•If

A condicional if é uma estrutura condicional que executa a afirmação, dentro do bloco, se determinada condição for verdadeira. Se for falsa, executa as afirmações dentro de else.

A sintaxe é assim:

if (condição) {

afirmação1

} else {

afirmação2

}

Múltiplas condicionais if ... else podem ser aninhados quando necessário.

if (condição1) {

instrução1

} else if (condição2) {

instrução2

} else if (condição3) {

instrução3

} else {

instruçãoN

}

•Switch

A condicional switch avalia uma expressão, combinando o valor da expressão para uma cláusula case, e executa as instruções associadas ao case. Normalmente um switch é usado para não ficar usando a condição if em grandes quantidades.

A sintaxe do switch em javascript é assim:

switch (expressão) {

case valor1:

//Instruções executadas quando o resultado da expressão for igual á valor1

break;

case valor2:

//Instruções executadas quando o resultado da expressão for igual á valor2

break;

case valueN:

//Instruções executadas quando o resultado da expressão for igual á valorN

break;

default:

//Instruções executadas quando o valor da expressão é diferente de todos os cases

[break;]

}

•Operadores Lógicos e de Comparação

Operadores lógicos são tipicamente usados com valores Booleanos (lógicos). Quando eles o são, retornam um valor booleano. Porém, os operadores && e || de fato retornam o valor de um dos operandos específicos, então se esses operadores são usados com valores não booleanos, eles podem retornar um valor não booleano.

Para adicionar uma condição reversa no caso falsa, usamos ! na frente da condição.

Sintaxe de operadores lógicos:

if (condição1 == condição2 && condição3 == condição4) {

executa alguma função

}

if (condição1 == condição2 || condição3 == condição4) {

executa alguma função

}

if ( !(condição1 == condição2) && !(condição3 == condição4)) {

executa alguma função

}

•Comparações de igualdade e uniformidade

JavaScript fornece três diferentes operações de comparação de valores:

•igualdade estrita (ou "três iguais" ou "identidade") usando ===,

•igualdade ampla ("dois iguais") usando ==,

•e Object.is (novo no ECMAScript 6).

Quando tentamos comparar dois valores um do tipo string e um do tipo number por exemplo com o type coercion isso se tornaria verdade se fosse “2” == 2 porém se usarmos o “2” === 2 se torna falso pois os tipos dos valores são diferentes.

•Aspas duplas, aspas simples e ponto e virgula

Em javascript as aspas simples e duplas não tem diferença e também ponto e vírgula no final não é obrigatório.

•Map|Filter|Reduce

Esses métodos map, filter e reduce são usados para arrays eles são bem comuns em framewroks de javascript.

O método. map() mapeia um array:

let number = [1,4 ,9]

let raiz = number.map(Math\*sqrt)

console.log(raiz)

Usando uma função de call-back

let number = [1,2,3,4,5].map( function(n){

return n\*2

})

console.log(number)

O metodo .filter() é usado para para procurar algo dentro do array :

function(maiorque){

return maiorque>=5

}

let number = [1,2,3,4,5,6,7,8,9,10].filter(maiorque)

O método .reduce() é usado para reduzir numero ou objeto dentro de um array:

let total = [0, 1, 2, 3].reduce(function(acumulador, valorAtual) {

return acumulador + valorAtual;

}

console.log(total)

Esse método soma os valores dentro do array transformando em um único objeto/elemento no array

let reduzido = [[0, 1], [2, 3], [4, 5]].reduce(

function(acumulador, valorAtual) {

return acumulador.concat(valorAtual)

})

console.log(reduzido)

Esse método reduz os arrays dentro do array para formar um único array .

•Callback

Uma função callback é uma função passada a outra função como argumento, que é então invocado dentro da função externa para completar algum tipo de rotina ou ação.

function greeting(name) {

alert('Hello ' + name);

}

function processUserInput(callback) {

var name = prompt('Please enter your name.');

callback(name);

}

processUserInput(greeting);

•Named Exports/Imports

O export é utilizado quando criamos módulos JavaScript para permitir que suas funções, objetos ou valores primitivos sejam utilizados por outros scripts através de import

Podem ser exportados funções classes, variáveis, objetos para criar um modelo. Exemplo:

Crie um arquivo onde você está fazendo seus códigos “Math.js“ e dentro crie uma função:

export let add = function (n1,n2) {

return n1+n2

}

Dentro do local onde você está fazendo seus códigos coloque acima do corpo um import as \* [o nome do arquivo] from [o local do arquivo] isso irá importar todo o modelo que você criou.

import \* as Math from './Math'

Depois para chamar suas funções ou função:

console.log (Math.add(5,2))

No terminal irá mostrar o resultado da soma de 5+2

Agora caso você quer importar só algumas funções ou objetos invés do modelo inteiro usamos:

import {add} from './Math'

Agora dentro do código você não precisa chamar o modelo basta so chamar função:

console.log (add(5,2))

No terminal irá mostrar o resultado da soma de 5+2

•Default Exprts/Imports

Permite importar/exportar apenas um valor de um modelo e os modelos mais comuns de dafalut exports são de classes e objetos.

Crie um arquivo dentro do local do projeto com o nome ‘Animal.js’ e dentro dele faça:

export default class Animal{

constructor(){

console.log(‘Im a na animal’)

}

getClasstype(){

return ‘Animal’

}

}

Depois vá para seu arquivo origem e importe o seu modelo assim:

import Animal from ‘./Animal.js’

Depois pra chamar a classe crie uma variável para armazenar:

let animal = new Animal()

console.log(‘Animal.getClasstype()’)

•Template Literals

Aprendemos antes que para mostrar os valores das variáveis ou resultados colocamos ‘+’ para mostrar o valor exemplo:

const name = 'Anna'

const country = 'Canada'

const age = 33

console.log('Name: ' +name+ 'Country: ' +country+ 'Age: '+age)

Com o template literals podemos fazer de uma forma mais simples e que também permite que chamemos algumas funções de javascript dentro do console.log, para usar o template literals usamos as aspas invertidas dentro do log e para chamar a variável usamos o símbolo $ + {}

console.log(`Name: ${name} Country ${country} Age ${age}`)

•Arrays e Spread Operators

Os spread operators servem para juntar ou concatenar arrays, objetos ou strings para uma chamada de função ou um novo array/sring. Para juntar os nomes dentro de um novo array criamos um novo array para armazenar e chamamos a varialvel com ´...´

const arrayOne = ['Maria',' Anna','Alex']

const arrayTwo = ['Said','Ismael','Aisha']

const concatArray = [...arrayOne,...arrayTwo];

Para mostrar os nomes criamos um laço de repetição:

concatArray.forEach((name)=>{

log(name)

})

Colocando o valor de uma variável com o spread operator:

const name = 'Marina'

const nametoArray = [...name]

Para mostrar as letras do nome inserido no array:

nametoArray.forEach((letter)=>{

log(letter)

})

Também podemos usar para números:

const addNumbers = function (n1,n2,n3) {

return n1+n2+n3

}

Essa função retorna a soma de 3 valores para isso criamos um array com 3 valores:

const numbers = [1,5,9]

Agora para mostrar a soma usamos o spread operator chamando a função:

const addition = addNumbers(...numbers)

console.log(addition)

No terminal irá informar a soma do array numbers.

•Objetos e Spread Operators

Os objetos também podem abusar do spread operator porém ele mais comum no framework React para usar o spread operators com objetos fazemos da mesma forma usando os ‘...’:

const address = {

city:'LA',

country: 'USA',

postCode: 'LA44'

};

const name = {

firstName: 'Andy',

lastName: 'Jones'

}

Criamos mais uma vaiável constante para armazenar os valores dos objetos de uma forma mais complicada faríamos assim :

const person = {

city: address.city,

country : address.country,

postCode: address.postCode,

firstName: name.firstName,

lastName: name.lastName

}

console.log(person)

Agora com o uso do spread operator olha a simplicidade que fica o código:

const person = {...address,...name}

console.log(JSON.stringify(person,null,2))

•Arrow Functions

Uma expressão arrow function possui uma sintaxe mais curta quando comparada a uma expressão de função ( function(name){return name} ), arrow functions são expressadas removendo o nome function e adicionando => após os parênteses.

(name) = > {return name}

Se a arrow function tiver apenas 1 parametro podemos remover os parentes da mesma também.

const milestoKm = miles => miles \* 1.60934

•This Lexico

This comporta-se um pouco diferente em Javascript se comparado com outras linguagens. Também possui algumas diferenças entre o modo estrito e o modo não estrito.

Em muitos casos, o valor this é determinado pela forma como a função é chamada. Ele não pode ser assinado durante a execução, e isso pode ser diferente a cada vez que a função é chamada. ES5 introduziu o método bind para estabelecer o valor this da função, independentemente de como ela seja chamada, e ECMAScript 2015 introduziu o arrow functions, cujo this é lexicalmente delimitado (o valor this é estabelecido segundo o escopo de execução no qual está inserido).

const person = {

name: 'Alex',

cars:['Ferrari','Lamborguini'],

toString: function(){

console.log(`${this.name} has ${this.cars}`)

this.cars.forEach( (car) => {

console.log(`${this.name} has ${car}`)

})

}

}

person.toString()

•Classes

Classes em JavaScript provêm uma maneira mais simples e clara de criar objetos e lidar com herança.

Uma maneira de definir uma classe é usando uma declaração de classe. Para declarar uma classe, você deve usar a palavra-chave class seguida pelo nome da classe (aqui "Retângulo").

class Animal {

constructor(name, age) {

console.log(`${name} is an animal and was created`);

this.name = name;

this.age = age;

}

eat() {

console.log(`${this.name} is eating`);

}

sleep() {

console.log(`${this.name} is sleeping`);

}

logAge() {

console.log(`${this.name} is ${this.age} year old`);

}

}

const booby = new Animal(‘bobby’,2)

bobby.eat()

bobby.sleep()

bobby.logAge()

O método constructor é um tipo especial de método para criar e iniciar um objeto criado pela classe. Só pode existir um método especial com o nome "constructor" dentro da classe.

•Herança

Em orientação a objetos a herança server para usar uma classe pai como base para uma nova classe. Exemplo:

class Dog extends Animal {

constructor(name, age, breed) {

super(name, age)

this.breed = breed;

}

logBreed() {

console.log(`${this.name} is a ${this.breed}`);

}

logDogAge(){

super.logAge()

}

logSleepDog(){  
 super.sleep()

}

}

No exemplo criamos uma classe para cachorro e como cachorro é um animal podemos usar a classe animal como herança para cachorro. Para usar a classe usamos extends ‘Nome da classe Pai’ e usamos o super para chamar o constructor da classe pai.

•Promise

Promise é um objeto usado para processamento assíncrono. Uma Promise representa um valor que pode estar disponível agora, no futuro ou nunca.

Uma Promise está em um destes estados:

* pending (pendente): Estado inicial, que não foi realizada nem rejeitada.
* fulfilled (realizada): sucesso na operação.
* rejected (rejeitado): falha na operação.
* settled (estabelecida): Que foi realizada ou rejeitada.

Para criar uma promise usaremos um exemplo com o setTimeout() para entender como funciona:

const promise = new Promise (function(resolve, reject){

setTimeout ( ( ) =>{

resolve( ‘data received from the server.’ )

}, 3000)

setTimeout ( ( )=>{

reject( ‘no data back from the server, there was an error’ )

},5000 )

} )

promise.then( (response)=>{

console.log(response)

} ).catch((error)=>{

console.log(error)

})

O valor a ser mostrado será o primeiro do setTimout no caso o resolve pois após 3s a mensagem será imprimida.

•Promise.all

O método Promise.all( ) retorna uma única Promise que resolve quando todas as promises no argumento iterável forem resolvidas ou quando o iterável passado como argumento não contém promises. É rejeitado com o motivo da primeira promise que foi rejeitada.

const namePromise = new Promise((resolve, reject) => {

setTimeout(() => {

resolve(['Anna' , 'Jones' , 'Ali' , 'Jake']);

}, 3000);

setTimeout(() => {

reject("no data back from the server, there was an error");

}, 5000);

});

const surnamePromise = new Promise((resolve, reject) => {

setTimeout(() => {

resolve(['Willians' , 'Bravo' , 'Mohammad' , 'Smith']);

}, 3000);

setTimeout(() => {

reject("no data back from the server, there was an error");

}, 5000);

});

Promise.all([namePromise,surnamePromise]).then( (data)=>{

const [names,surnames] = data

for (let i = 0; i < names.length; i++) {

const name = names[i]

const surname = surnames[i]

console.log(`${name} ${surname}`)

}

}).catch((erro)=>{

console.log(erro)

})

•Fetch Api

O Fetch fornece uma definição genérica de objetos de Request e Response (e outras coisas envolvidas com solicitações de rede). Isso permitirá que eles sejam usados onde quer que sejam necessários no futuro, seja para service workers, Cache API e outras coisas similares que manipulam ou modifiquem pedidos e respostas ou qualquer tipo de caso de uso que possa exigir que você gere suas próprias responses programaticamente.

Ele também fornece uma definição para conceitos relacionados como CORS e a semântica de cabeçalho de origem HTTP, suplantando suas definições separadas em outro lugar.

<https://developer.mozilla.org/pt-BR/docs/Web/API/Fetch_API/Using_Fetch>

• Promises e Fetch API

const getRandomUsers = (n)=>{

const fetchRandomUsers = fetch(`https://randomuser.me/api/?results=${n}`)

fetchRandomUsers.then((data)=>{

data.json().then((randomUsers) => {

log(JSON.stringify(randomUsers.results.length))

randomUsers.results.forEach(user => {

const {gender, email} = user

log(`${gender} - ${email}`)

});

})

})

}

getRandomUsers(100)

•Generators

Generator é uma função que pode ser pausada, e isso nos permite escrever nosso código de uma forma assíncrona.

Para escrever um generator a sintaxe é bem simples:

function\* name(){

yield ‘Anna’

}

Escreva uma função e coloque um asterisco na frente e para uma função ser generator temos que usar o yield para os valores das variáveis.

const getValues = function\* (){

yield 1,

yield 'hello',

yield true,

yield {name:'Alex'}

return ‘I am done’

}

const valuesGen = getValues()

console.log(valuesGen.next().value)

console.log(valuesGen.next().value)

console.log(valuesGen.next().value)

console.log(JSON.stringify(valuesGen.next().value))

console.log(valuesGen.next().value)

•Coroutines

Para usar coroutines com promises precisamos instalar 2 programas, caso você esteja no front-end e queira usar coroutines+promises instale o bluebird web caso esteja queira usar no back-end instale o co+node a sintaxe do coroutine é assim:

coroutine(function\* numbersGen(){

const response = yield promise

})

•Generators e Promises

const getRandomUsers = co(function\*(n){

const fetchRandomUsers = yield fetch(`https://randomuser.me/api/?results=${n}`)

const data = yield fetchRandomUsers.json()

return data

})

getRandomUsers(10).then(randomUsers =>{

randomUsers.results.forEach(user => {

const {gender, email} = user;

log(`${gender} - ${email}`);

});

}).catch(error =>{

console.log(error)

})

•Async Await

Quando uma função assíncrona é chamada, ela retorna uma Promise. Quando a função assíncrona retorna um valor, a Promise será resolvida com o valor retornado. Quando a função assíncrona lança uma exceção ou algum valor, a Promise será rejeitada com o valor lançado.

Uma função assíncrona pode conter uma expressão await, que pausa a execução da função assíncrona e espera pela resolução da Promise passada, e depois retoma a execução da função assíncrona e retorna o valor resolvido.

<https://alligator.io/js/async-functions/>