**Callback**

*// CALLBACK: function which receives another function (call) as a parameter*

*// Function which will be called*

function sum(num1, num2){

return num1 + num2

}

*// function which will do the callback*

function call\_func(num1, num2, callback){

return callback(num1, num2)

}

console.log(call\_func(6,4,sum))

**Answer:**

10

function date\_printed(){

console.log(new Date)

setTimeout(function internal\_fun(){

let date = new Date

function printDate (dateNow) {

console.log (dateNow)

}

printDate(date);

}, 3000)

}

date\_printed()

Equivalent

function print\_date (dateNow){

console.log (dateNow)

}

function date(callback){

console.log(new Date)

setTimeout(function internal\_fun(){

let date = new Date

callback(date)

}, 3000)

}

date(print\_date)

**Answer:**

2020-02-17T04:18:55.798Z

2020-02-17T04:18:58.798Z

**Example**

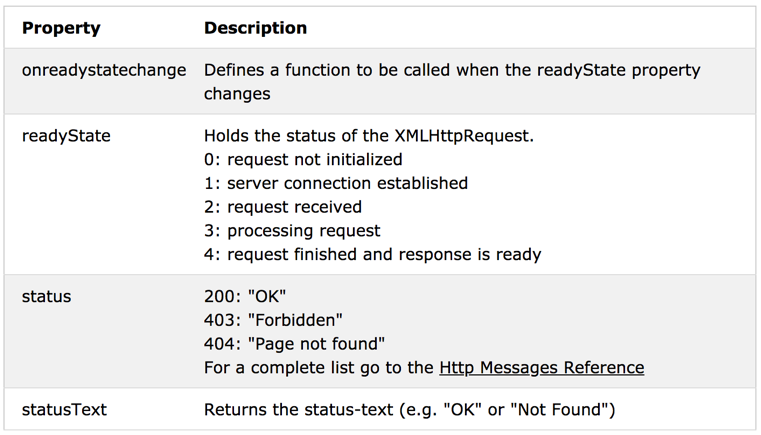
Do a request to an URL thorugh callbacks

The onreadystatechange event is triggered five times (0-4), one for each change in the readyState.

It is necessaty to be listening to onreadystatechange and ask if there is any change.

* The readyState property holds the status of the XMLHttpRequest.
* The onreadystatechange property defines a function to be executed when the readyState changes.
* The status property and the statusText property holds the status of the XMLHttpRequest object.

Values in: <https://www.w3schools.com/xml/ajax_xmlhttprequest_response.asp>



*//****xmlhttprequest*** *is inside the javaScript*

*//which will allow us to do request to any service*

*//1. Install dependency: terminal>> npm install xmlhttprequest --save*

*// Instantiate the library which was installed*

let XMLHttpRequest = require('xmlhttprequest').XMLHttpRequest

*// URL to which we are going to make the requests*

const API = 'https://rickandmortyapi.com/api/character/'

*// "fetch\_data" brings the information from the API*

function fetch\_data(url\_api, callback){

// Step 1. Element XMLHttpRequest

const xhttp = new XMLHttpRequest()

// Step 2. listen to onreadystatechange and ask if there is any change

xhttp.onreadystatechange = function (){

if (xhttp.readyState === 4){

if(xhttp.status === 200 ){

*// the callback will have two parameters: 1.error, 2.the answer of the call to the app*

*// it is necessary to transform to JSON format, because if not I will receive a string (the same if I dont have postman)*

callback (null, JSON.parse(xhttp.responseText))

} else {

const error = new Error('Error-->' + url\_api)

return callback(error, null)

}

}

}

*//Step 3. call an URL and do a request (GET)*

xhttp.open('GET', url\_api, true)

*//Step 4. finalize the request*

xhttp.send()

return callback(error, null) *//null because i don't have any answer from the request*

xhttp.open('GET', url\_api, true) *//true to activate the asynchronism. we are opening a conception*

**Implementation:**

Structure of this Callback = (error, string from the URL)

*//My first request is the information in the page 1*

fetch\_data(API, function(error1, data1){

if (error1) return console.log (error1)

console.log(data1)

})

**Answer:**

It will print the data from API which corresponds to the data in the page 1

*//My first request is the information in the page 3*

const data\_1 = fetch\_data(API+'?page=3', function(error1, data1){

if (error1) return console.log (error1)

console.log(data1)

})

**Answer:**

it will print the data from *API + '?page=3'*, it corresponds to the data in the page 3

*//data2: second request gets the information of the character 3*

*// id\_character and new\_API get the same data*

fetch\_data(API,function(error1, data1){

if (error1) return console.log(error1)

const id\_character = data1.results[2].id

console.log(id\_character)

fetch\_data(API+id\_character, function(error2, data2){

if (error2) return console.log(error2)

console.log (data2)

})

const new\_API = data1.results[2].url

fetch\_data(new\_API, function(error2, data2){

if (error2) return console.log(error2)

console.log (data2)

})

})

**Answer:**

Request 1 (fetch\_data(API,function(error1, data1){}) will return the id number of the character = 3

Request 2.1 and 2.2. are the same, both will get the infromation of the character 3.

Request 2.1 (fetch\_data(API+id\_character, function(error2, data2){})

Request 2.2 (fetch\_data(new\_API, function(error2, data2){})

fetch\_data(API, function(error1, data1){

if (error1) return console.log(error1)

fetch\_data (API + data1.results[0].id, function (error2, data2){

if (error2) return console.log(error2)

fetch\_data (data2.origin.url, function (error3, data3){

if (error3) return console.log(error3)

console.log(data1.info.count)

console.log(data2.name)

console.log(data3.dimension)

})

})

})

**Answer:**

493

Rick Sanchez

Dimension C-137

**Promise**

**Asybchronism:** Asynchronous programming is a form of parallel programming that allows a unit of work to run separately from the primary application thread.

JavaScript engine can only process one statement at a time in a single thread.

**Example of asynchronism:** setTimeout runs separately from the main program.

let age = 18

console.log ('A. Checking age.')

setTimeout(()=>{

age++

console.log('B. Happy birthday.')

},2000)

console.log (`C. Now your age is ${age}`)

**Answer:**

A. Checking age.

C. Now your age is 18

B. Happy birthday.

A promise is an object which can be returned synchronously from an asynchronous function. This function asynchronously generates grades by giving a record of grades and two callback functions; One called if the course is successfully approved (resolve), and the other called if it is failed (reject).

**Example of promise:**

let age = 18

function fun\_promise (resolve, reject){

setTimeout(()=>{

age++

console.log('2. Happy birthday.')

resolve(age)

},2000)

}

const my\_promise = new Promise (fun\_promise)

console.log ('1. Checking age.')

*// I promise when Im done with my\_promise (with what is inside)*

*// "then" I will do what is inside the "positive\_answer"*

my\_promise

.then(positive\_answer => console.log (`3. Now your age is ${age}`))

**Answer:**

1. Checking age.

2. Happy birthday.

3. Now your age is 19

A callback is a function which has another function as a parameter. It is possible to transfor a callback into a promise:

**Example 1** Transforming callbacks into promises:

**Callback**

**Promise**

function sum\_promises(num1, num2){

return new Promise((resolve, reject)=>{

if (num1 > num2){

resolve(num1 + num2)

}else{

reject(num2-num1)

}

})

}

Equivalent

function sum(num1, num2){

if (num1 > num2){

return(num1 + num2)

}else{

return(num2-num1)

}

}

function call\_func(num1, num2, callback){

return callback(num1, num2)

}

console.log(call\_func(6,4,sum))

console.log(call\_func(4,6,sum))

**Answer:**

10

2

The arguments to "then" are optional, and "catch(failureCallback)"" is short for "then(null, failureCallback)"

sum\_promises(6,4)

.then ((returnFromResolve)=> {

console.log(returnFromResolve)

return (sum\_promises(4,6))

})

.then ((returnFromResolve)=> console.log(returnFromResolve))

.catch((returnFromReject) => console.log(returnFromReject))

**Answer:**

10

2

Promises usually are defined as a let or const.

When we create a promise we are telling it what we are going to do if we succeed "resolve (num1 + num2)" or what we are going to do if we fail "reject (num2 - num1)"

**Promise (function)**

**Promise (let or const)**

let sum\_promises\_let = (num1, num2) =>(

new Promise((resolve, reject)=>{

if (num1 > num2){

resolve(num1 + num2)

}else{

reject(num2-num1)

}

})

)

function sum\_promises(num1, num2){

return new Promise((resolve, reject)=>{

if (num1 > num2){

resolve(num1 + num2)

}else{

reject(num2-num1)

}

})

}

So when we use the promise we are saying: do "then" if we succeed or do "catch" if we fail

sum\_promises\_let(6,4)

.then ((returnFromResolve)=> {

console.log(returnFromResolve)

return (sum\_promises(4,6))

})

.then ((returnFromResolve)=> console.log(returnFromResolve))

.catch((returnFromReject) => console.log(returnFromReject))

**Answer:**

10

2

**Example 2** Transforming callbacks into promises:

**Callback**

**Promise**

const IWantDate = true

function date(){

return new Promise((resolve,reject)=>{

if (IWantDate){

console.log(new Date)

setTimeout(()=>{

let date = new Date

resolve (date)

}, 3000)

}else{

const error=new Error('Dammit!')

reject (error)

}

})

}

Equivalent

function print\_date (dateNow){

console.log (dateNow)

}

function date\_callback(callback){

console.log(new Date)

setTimeout(function internal\_fun(){

let date = new Date

callback(date)

}, 3000)

}

date\_callback(print\_date)

date()

.then((dateFromResolve) =>{

console.log (dateFromResolve)

})

.catch((messageFromError) => {

console.log (messageFromError)

})

**Answer:**

2020-02-23T13:20:30.929Z

2020-02-23T13:20:33.930Z

Promises will be in one of 3 possible states:

Fulfilled: onFulfilled() will be called (e.g., resolve() was called)

Rejected: onRejected() will be called (e.g., reject() was called)

Pending: not yet fulfilled or rejected

**Promise (function)**

**Promise (let or const)**

let dateLet=new Promise((resolve,reject)=>{

if (IWantDate){

console.log(new Date)

setTimeout(()=>{

let date = new Date

resolve (date)

}, 3000)

}else{

const error=new Error('Dammit!')

reject (error)

}

})

const IWantDate = true

function date(){

return new Promise((resolve,reject)=>{

if (IWantDate){

console.log(new Date)

setTimeout(()=>{

let date = new Date

resolve (date)

}, 3000)

}else{

const error=new Error('Dammit!')

reject (error)

}

})

}

dateLet

.then((dateFromResolve) => console.log (dateFromResolve))

.catch((messageFromError) => console.log (messageFromError))

**Answer:**

2020-02-23T13:20:30.929Z

2020-02-23T13:20:33.930Z

**Concatenate promises with Promise.all( )**

The **Promise.all()** method returns a single Promise that fulfills when all of the promises passed as an iterable have been fulfilled

**Promise1**

**Promise2**

promise2()

.then(response => console.log(response))

.then(response => console.log('Second response'))

.catch(err => console.error(err))

promise1()

.then(response => console.log(response))

.catch(err => console.error(err))

const promise2 = () =>{

return new Promise((resolve, reject)=>{

if (true){

setTimeout(()=>{

resolve('Solved promise2')

},2000)

}else{

const error = new Error('Dammit! promise2')

reject (error)

}

})

}

const promise1 = () => {

return new Promise((resolve, reject) =>{

if (true){

resolve('Solved promise1')

}else{

reject('Dammit! promise1')

}

})

}

**Answer:**

Solved promise1

Solved promise2

Second response

Promise.all([promise1(), promise2()])

.then(response => {

console.log('Array of results:', response)

})

.then(response => console.log('Second response'))

.catch(err => console.error(err))

**Answer:**

Array of results: [ 'Solved promise1', 'Solved promise2' ]

Second response

**Example**

Do a request to an URL thorugh promises

*//****xmlhttprequest*** *is inside the javaScript*

*//which will allow us to do request to any service*

*//1. Install dependency: terminal>> npm install xmlhttprequest --save*

*// Instantiate the library which was installed*

let XMLHttpRequest = require('xmlhttprequest').XMLHttpRequest

*// URL to which we are going to make the requests*

const API = 'https://rickandmortyapi.com/api/character/'

const fetch\_data = (url\_api) => {

return new Promise((resolve, reject)=>{

const xhttp = new XMLHttpRequest()

xhttp.onreadystatechange = (() => {

if (xhttp.readyState === 4){

if(xhttp.status === 200){

*// it is necessary to transform to JSON format, because if not I will receive a string (the same if I dont have postman)*

resolve (JSON.parse(xhttp.responseText))

}else{

reject (new Error ('Error ', url\_api))

}

}

})

xhttp.open('GET', url\_api, true)

xhttp.send()

})

}

**Implementation:**

Structure of this promise = promise(API).then(return data1front the API)

fetch\_data(API) *//equal to the parse of the first page*

.then ((returnFromResolve)=> console.log(returnFromResolve))

.catch((returnFromReject) => console.log(returnFromReject))

**Answer:**

It will print the data from API which corresponds to the data in the page 1

fetch\_data(API+'?page=3')

.then ((returnFromResolve)=> console.log(returnFromResolve))

.catch((returnFromReject) => console.error(returnFromReject))

*//no difference with .log and .error for a var New Error()*

**Answer:**

it will print the data from *API + '?page=3'*, it corresponds to the data in the page 3

fetch\_data(API)

.then((data1)=> {

*// result of the first promise (page1)*

console.log('PROMISE 1:', data1)

console.log(data1.info.count)

return fetch\_data (API + data1.results[2].id)

})

.then((data2)=> {

*// result of the second promise (character3)*

console.log('PROMISE 2:', data2)

console.log(data2.name)

return fetch\_data (data2.origin.url)

})

.then((data3)=> {

*// result of the third promise (origen)*

console.log('PROMISE 3:', data3)

console.log(data3.dimension)

})

.catch((returnFromReject) => console.log(returnFromReject))

.then1 will print the full data1 and the count of charates. Also, it will call the following promise

.then2 will print the full data2 and the name of the id 2. Also, it will call the following promise

.then3 will print the full data3 and the dimension of the character with id 2.

**Answer:**

PROMISE 1:

…

493

PROMISE 2:

…

Summer Smith

PROMISE 3:

…

Replacement Dimension

**Async/await**

We use an async function and await for resolving the promise before a event happens.

As we know the promises can create asynchronism in our code

**Promise:**

let age = 10

const increaseAge = () =>{

return new Promise ((resolve, reject)=>{

(true)

? setTimeout(()=>{

message=`A. Increasing age to ${age+=1}`

resolve (console.log(message))

},2000)

: reject (new Error('Dammit!'))

})

}

Async/await is a tool to tranform a ansynchronous function into a sinchronous function:

1. You have to have a function function DoSomethingAsync () to
2. Use async before function DoSomethingAsync () to tell JavaScript that function is asynchronous. So, JavaScript will know how to handle the doce inside.
3. Use await before call the promise increaseAge() to tell JavaScript to wait until that code is executed and continue with the next thing.
4. To get the errors or reject from the promise we can use try{} and catch (error){}.

Having an asynchronous code as the following implementation of a promise, we can try to make synchronous an asynchronoush code.

console.log ('Before')

constDoSomethingAsync()

console.log ('After')

**Asynchronous**

**Synchronous**

console.log ('Before')

constDoSomethingAsync()

console.log ('After')

async function DoSomethingSync () {

const something = await increaseAge()

console.log ('B. Confirming the age')

console.log(`C. Happy ${age}th birthday.`)

}

function DoSomethingAsync () {

const something = increaseAge()

console.log('B. Confirming the age')

console.log(`C. Happy ${age}th birthday`)

}

**Answer for Asyncronous:**

Before

B. Confirming the age

C. Happy 10th birthday.

After

A. increasing age to 11

**Answer for Synchronous:**

Before

After

A. increasing age to 11

B. Confirming the age

C. Happy 11th birthday.

One common use of Async/Await is to use it to chain multiple asynchronous calls consecutively.

If you instead want the steps to happen in parallel, you can simply use **Promise.all** to wait for all the promises to have fulfilled. Promise.all returns an array with the resolved values once all the passed-in promises have resolved.

const chainPromises = async ()=>{

try{

console.log ('1.Checking age.')

const something = await increaseAge ()

const something2 = await increaseAge ()

const something3 = await increaseAge ()

const something4 = await increaseAge ()

console.log('B. Happy fourth birthday.')

console.log(`C. Happy ${age}th birthday.`)

}catch (error){

console.log(error)

}

}

console.log ('Before')

chainPromises()

console.log ('After')

**Answer:**

Before

1.Checking age.

After

A. Increasing age to 11

A. Increasing age to 12

A. Increasing age to 13

A. Increasing age to 14

B. Happy fourth birthday.

C. Happy 14th birthday.

**Example**

Do a request to an URL thorugh promises

*//****xmlhttprequest*** *is inside the javaScript*

*//which will allow us to do request to any service*

*//1. Install dependency: terminal>> npm install xmlhttprequest --save*

*// Instantiate the library which was installed*

let XMLHttpRequest = require('xmlhttprequest').XMLHttpRequest

*// URL to which we are going to make the requests*

const API = 'https://rickandmortyapi.com/api/character/'

const fetch\_data = (url\_api) => {

return new Promise((resolve, reject)=>{

const xhttp = new XMLHttpRequest()

xhttp.onreadystatechange = (() => {

if (xhttp.readyState === 4){

if(xhttp.status === 200){

*// it is necessary to transform to JSON format, because if not I will receive a string (the same if I dont have postman)*

resolve (JSON.parse(xhttp.responseText))

}else{

reject (new Error ('Error ', url\_api))

}

}

})

xhttp.open('GET', url\_api, true)

xhttp.send()

})

}

**Implementation:**

The power of async functions becomes more evident when there are multiple steps involved.

async function doSomethingAsync (url\_API){

try{

const data1 = await fetchData (url\_API)

const data2 = await fetchData (url\_API + data1.results[2].id)

const data3 = await fetchData (data2.origin.url)

console.log(data1.info.count)

console.log(data2.name)

console.log(data3.dimension)

}catch (error){

console.log(error)

}

}

.await1 will get the count of charates.

.await2 will get the name of the id 2.

.await3 will get the dimension of the character with id 2.

console.log ('Before')

doSomethingAsync(API)

console.log ('After')

**Answer:**

Before

After

493

Summer Smith

Replacement Dimension

*/\**

*To solve asynchronous problems we can use callback, promises and async/await*

*\*callback:*

*it is a function which has another function as a parameter*

*they are universal, mostly all browser can run callbacks*

*it is easy to get confuse and lose because it grows and grows*

*its structure is basically: if error, if error, ...*

*?promise:*

*easy to chain. An then can be chained to the next one by a return inside the then*

*it doesn't work with exceptions.*

*At the bottom has a catch where all errors are collected*

*It needs a qualify to run in all browsers.*

*!async :*

*they are easy to read*

*we wait until something happens to continue with the next line*

*we have to wait for each call*

*If we want to do more calls, we need to wait for them more and more*

*It needs a qualify to run in all browsers.*

*\*/*