Asynchronous

someAsyncOperation(dataObject, function(error, success) {

if (error) {  
 // handle error  
 } else {  
 // do something with success  
 }  
})

//función síncrona  
**function addToArraySync(dato, lista) {**

**if ( !Array.isArray(lista) ) {**

**console.log( "la variable no es un array" )**

**}**

**else {**

**lista.push(data);**

**}**

**}**

//función Asíncrona  
**function addToArrayAsync(dato, lista, funcionCallback) {**

**if ( !Array.isArray(lista) ) {**

//new Error() Is optional, but it requieres 2 parameters  **funcionCallback(new Error( "No existe el array" ), null); }**

**else {**

**lista.push(dato);  
 callback(null, lista);  
 }  
}**

Both functions do the same, but if the Array list comes from a socket/API or is subject to any externalized modification, it is necessary to control the execution time 🡪Asynchronous function

Callback Pros:

* Handy for single async operations. Allow easy data and error control.
* Should work on every node version and almost all packages of node. As callbacks are functions, they don’t need any transpilers.

Callback Cons:

* For multiple nested async operations this creates a callback hell
* Error handling has to be done for each operation (no global exception handler)

A promise represents an operation that still it has not been completed but it is expected in the future. Promises are objects which have 3 main states — pending, resolved and rejected. Depending on the response of an async action a promise is either resolved or rejected. Multiple promises can be chained one below the other. A single catch handler at the bottom is sufficient for an error in any promise.

**document.write( "Arranque de los dos coches" );  
document.write( "Porsche 911 vs Ferrari Testarrsa<br/>" );**

**let miPromesa = new Promise( function(resolve, reject) {**

**setTimeout(**

**function() {**

**if (Math.random() > 0.5) {**

**resolve( alert( "El Porsche 911 es el más rápido" ) );**

**}**

**else {**

// uncaught exception  
 **reject( "Ferrari Testarrosa es el más rápido" );**

**}**

**},**

**5000**

**);**

**});**

**document.write( "Control del radar tras 5 segundos" );**

Promises Pros:

* Allows for easy chaining of async operations. Whatever is returned in the .then function, can be chained in the next .then function.
* One catch handler at the bottom will catch an error if either of the chained promises throws an exception.

Promises Cons:

* Most libraries may require a promisify wrapper around it like bluebird, unless they support promises out of the box.
* The scope of a chained function is isolated to that function itself. So some data resolved in the second chain cannot be used in the 4th chain unless a global let variable is declared.

**let hamburguesa = [];**

# Solución de Asincronía con Callback

**function addToArrayCallback ( dato, lista, funcionCallback ) {**

**if ( !Array.isArray(lista) ) {**

//new Error() Is optional, but it requires 2 parameters  **funcionCallback(new Error( "No existe el array" ), null);   
 }  
 else {**

**lista.push(dato);  
 callback(null, lista);  
 }  
}**

**function callbackHamburguesas( mensajeError, lista ) {**

**if (mensajeError !== null) {return console.log( mensajeError );} else { console.log( lista ); }  
}**

It is the callback function who manage the execution.

**addToArrayCallback( "pan de abajo", hamburguesa, callbackHamburguesas);  
addToArrayCallback( "carne", hamburguesa, callbackHamburguesas );  
addToArrayCallback( "lechuga", hamburguesa, callbackHamburguesas );**

# Solución de Asincronía con Promesa

**function exito(dato, lista) {**

**lista.push(dato);** //Acciones a procesar **console.log(lista);  
}**

**function fracaso() {console.log( "Error, la hamburguesa está llena" );}**

**function addToArrayPromise( dato, lista ) {**

**let promesa = new Promise( function(exito, fracaso) {**

**if( !Array.isArray(lista) ) { fracaso(); }  
 else { exito(dato, lista); }**

**});**

**return promesa;  
}**

**function pintarHamburguesa(lista) { console.log(lista); }**

**function decirError(lista) {**

**console.log( "Error, no se ha podido aumentar la hamburguesa" );  
}**

**addToArrayPromise( "cebolla", hamburguesa )**

**.then( pintarHamburguesa )  
.catch( decirError );**

Try() delimits a block of instructions that has to be checked, and the catch() block, contains a return in case an error happens in the try() block. The try() block, always present by definition, can have many instructions. For the exception processing, there may be several consecutive catch, or even none. It is also possible to foresee, lastly after the catch() series, a last block called finally() and in which systematically executed sentences will be integrated before the execution of the following instructions of the program.

**function comprobarDia( numeroDia ) {**

**numeroDia = numeroDia - 1;**

**let tablaDias = new Array(**

**"Lunes",  
"Martes",   
"Miércoles",   
"Jueves",   
"Viernes",   
"Sábado",   
"Domingo"**

**);**

**if ( tablaDias[numeroDia] != null ) {return tablaDias[numeroDia]; }**

**else { return "Error en el número del día"; }**

**}**

**const dayNumber = 8;**

**try {**

**alert( comprobarDia(dayNumber) );  
 alert( verificacionDia() );**

**}  
catch(error) { alert( "No existe la función verificaicionDia" ); }(**

Async / await at the end of the day is still a promise. It’s just a way of writing asynchronous code in a sort of synchronous manner. Each async function has to be prefixed with async. Every asynchronous action in it has to be prefixed with the word await . Also, every async function returns a promise which can be resolved further.

Of1c10365

Consultoria y desarrollo

IBM, bizaggui, CoiProcess y Appian

**Proyecto Altamira** 🡪 excedentes mobiliarios del Santander

23 días laboral anuales + 4 días Atmira + 1 fisio