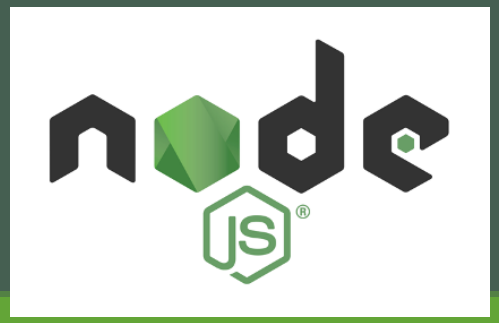


Async|Await

Contenido

- Promesas
- Async|Await
- Inherit

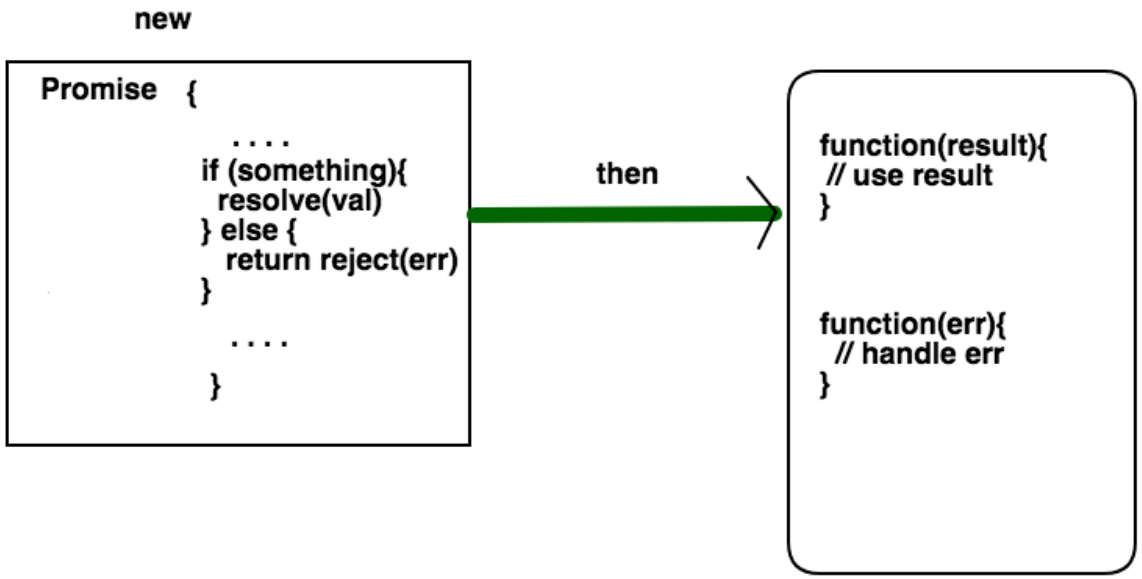
Promesas

- Capa de captura (proxy) de valores que no es necesario conocer en el momento de crear dicha promesa.
- Permite asociar gestores a acciones asíncronas que retornan un valor de éxito o de error
- Se crean utilizando el operador **new** sobre el objecto **Promise** pasando como único parametro una **función ejecutora**. Esta función a su vez recibe dos parametros: un callback que será ejecutado cuando la promesa se ha cumplido (**resolve**) y otro que será ejecutado cuando no (**reject**).

```
const promise = new Promise(function executor(resolve, reject)
{
  // Some async operations here...
});
```

Promesas

- Una vez se ha generado la promesa se puede tratar con valores asíncronos utilizando la función **then** que recibe dos parámetros:
  - onSuccess: El callback que será ejecutado cuando la promesa se resuelva.
  - onError: El callback que será ejecutado si la promesa devuelve un error.
  - El error también puede tratarse directamente con la función **catch()**



Promesas: Consideraciones

- Una promesa puede ser creada en el código o puede ser devuelta por un package node externo
- Cualquier promesa que lleve a cabo operaciones asíncronas debe llamar a uno de estos dos métodos: **resolve** o **reject.**
- La lógica de programación debe controlar cuándo y dónde llamar a esas funciones
- Si la operación tiene éxito se pasan los datos al código que usa la promesa si no, se pasa el error
- El código que usa la promesa invoca a la función **then** que lleva dos funciones anónimas como parámetros que se ejecutarán según se haya resuelto o rechazado la promesa
- Si se trata de acceder al valor de la promesa antes de que se resuelva o rechaza la promesa estará en estado '**pending**'

Promesa: Encadenamiento de promesas

- La función **then** devuelve a su vez una promesa permitiendo que sean encadenadas facilmente. Además el valor que se devuelve en then se pasa como parámetro al siguiente.

Ejemplo Promesa (I)

- https://scotch.io/tutorials/javascript-promises-for-dummies
- Imagina que eres un niño y tu madre te promete que te regalará un móvil nuevo si te portas bien.
  - Creamos la promesa:

```
const isMomHappy = true;

const willIGetNewPhone = new Promise(
    (resolve, reject) => { // fat arrow
        if (isMomHappy) {
            const phone = {
                brand: 'Samsung',
                color: 'black'
            };
        resolve(phone);
        } else {
            const reason = new Error('mom is not happy');
            reject(reason);
        }

     }
);
```

Ejemplo Promesa (II)

- Llamamos a la promesa:

```
const askMom = function() {
    willIGetNewPhone
    .then(console.log)
    .catch( error =>console.log(error.message));
}

askMom();
```

Ejemplo Promesa (III)

- Ahora prometes que si tienes un móvil nuevo se lo enseñarás a tus amigos. Añades otra promesa

```
const showOff = function (phone) {
    const message = 'Hey friend, I have a new ' +
          phone.color + ' ' + phone.brand + ' phone';
    return Promise.resolve(message);
};
// call our promise
const askMom = function () {
    willIGetNewPhone
        .then(showOff)
        .then(fulfilled => console.log(fulfilled)) // fat arrow
        .catch(error => console.log(error.message)); // fat arrow
        };

askMom();
```

Ejercicio 11

- Crear una operación que devuelva el resultado de una suma asíncrona en una promesa. Después elevar al cuadrado el resultado.
  - Si algún sumando es 0 devolverá error.
  - Si el resultado de la potencia es mayor que 100 devolverá error.

Promesas: Métodos estáticos

- Promise.**resolve**

```
let promise = Promise.resolve(valor);
```

  - Crea una promesa de cumplimiento
- Promise.**reject**

```
let promise = Promise.reject(error);
```

  - Crea una promesa de rechazo
- Promise.**all**

```
let promise = Promise.all(iterador);
```

  - Ejecuta muchas promesas en paralelo y espera hasta que todas ellas han terminado.
  - El iterador normalmente es un array que de vuelve una promesa

Ej.:

```
Promise.all([
new Promise((resolve, reject) => setTimeout(() => resolve(1),
3000)), // 1
new Promise((resolve, reject) => setTimeout(() => resolve(2),
2000)), // 2
new Promise((resolve, reject) => setTimeout(() => resolve(3),
1000)) // 3
]).then(console.log); // 1,2,3 when promises are ready: each
promise contributes an array member
```

Promesas: Métodos estáticos

- Promise.**all**
  - Una estrategia utilizada es mapear un array de datos a un array de promesas y después envolverlo con Promise.all. Ej:

```
'use strict';
let urls = [
'https://api.github.com/users/iliakan',
'https://api.github.com/users/remy',
'https://api.github.com/users/jeresig'
];
// map every url to the promise fetch(github url)
let requests = urls.map(url => fetch(url));
// Promise.all waits until all jobs are resolved
Promise.all(requests)
.then(responses => responses.forEach(
response => alert(`${response.url}:
${response.status}`)
));
```

Ejercicio 12

- Recorrer un directorio manejando la respuesta de si es un fichero o un directorio con promesas.
  - Si es un fichero concatenar el nombre del fichero al directorio
  - Si es un directorio volver a invocar a la función.
  - Obtener el resultado por consola.

async|await

- Desde ES7 y nodejs > 6 se introduce la expresión async/await se utiliza para manejar la asincronía:
  - **async** delante de una función significa que esa función devolverá una promesa.

```
async function f(){return 1;}
f().then(console.log);
```

//es igual a async function f() {return Promise.resolve(1)}

  - **await** trabaja sólo con funciones async y lo que hace es parar la ejecución del programa hasta que se completa la función async.

```
async function f() {

let promise = new Promise((resolve, reject) => {
setTimeout(() => resolve("done!"), 1000)
});
let result = await promise; // wait till the promise
resolves (*)
console.log(result); // "done!"
}
f();
```

async|await

- Manejar los errores con try-catch
  - **async** delante de una función significa que esa función devolverá una promesa.

```
async function f() {

try {
let response = await fetch('/no-user-here');
let user = await response.json();
} catch(err) {
// catches errors both in fetch and response.json
alert(err);
}
}
f();
```

Ejercicio 13

- Rehacer el ejemplo del "móvil nuevo" con async/await