Promesas (práctica)

Tiempo estimado: 20min

El objeto de esta práctica es afianzar, reforzar y consolidar los conocimientos teóricos presentados en la lección.

Al finalizarla, el estudiante:

- Habrá trabajado con promesas.
- Habrá creado funciones asíncronas con promesas.
- Habrá encadenado promesas.

Objetivos

El objetivo de la práctica es mostrar cómo trabajar con promesas. Recordemos que las promesas es una característica de JavaScript, no es específica de Node. Se puede usar tanto en Node como en los navegadores webs u otros motores.

En primer lugar, crearemos una función asíncrona mediante usando una promesa. Y a continuación, usaremos esta función para ver cómo registrar controladores asociados a la finalización de la operación asíncrona.

Definición de funciones asíncronas con promesas

Recordemos que las promesas es un medio de comunicación asíncrona. Básicamente tienen un doble objetivo:

- Comunicar que una operación asíncrona ha finalizado.
- Registrar los bloques de código que se deben ejecutar al finalizar la operación asíncrona.

Vamos a definir una función que devuelve una promesa a través de la cual comunicarnos con el usuario. Esta función es una implementación de la función readFile() del módulo fs.

1. Abrir una consola.
2. Crear el directorio de la práctica e ir a él.
3. Crear el archivo arch.txt:

```
$ echo "esto es el contenido" > arch.txt
```

4. Crear el archivo fs-promise.js:

```
//imports
const fs = require("fs");

//api
module.exports = exports = {
  readFile
};

/**
 * Lee el contenido de un archivo.
 *
 * @param path:string   Ruta al archivo.
 * @param opts?:object  Opciones de lectura.
 *
 * @return Promise
 */
function readFile(path, opts = {encoding: "utf8"}) {
  return new Promise(function(resolve, reject) {
    fs.readFile(path, opts, function(err, data) {
```



```
      if (err) reject(err);
      else resolve(data);
    });
  });
}
```

Preste especial atención a la función. Analícela detenidamente y compréndala. Si no lo tiene claro, no sabrá definir sus propias promesas.

Uso de promesas

Ahora, vamos a ver cómo usar funciones asíncronas implementadas mediante promesas. Para ello, usaremos el módulo fs-promise que acabamos de definir.

1. Ir a la consola.
2. Abrir node en modo interactivo.
3. Importar el módulo fs-promise:

```
> fsp = require("./fs-promise")
{ readFile: [Function: readFile] }
>
```

4. Leer el archivo arch.txt mediante la función readFile():

```
> p = fsp.readFile("arch.txt")
Promise { <pending> }
>
```

5. Registrar el controlador de éxito para que muestre el contenido del archivo:

```
> p.then(function(data) { console.log(data); })
Promise { <pending> }
> esto es el contenido
```

6. Registrar otro controlador de realización que muestre el contenido del archivo precedido del mensaje Otra vez:

```
> p.then(function(data) { console.log("Otra vez", data); })
Promise { <pending> }
> Otra vez esto es el contenido
```

A pesar de que la operación asíncrona ya ha finalizado, se puede registrar nuevos controladores. Los cuales se ejecutan asíncronamente si el estado al que se asocian es en el que se encuentra la promesa. Esto significa que las promesas guardan los parámetros que deben pasar a los controladores hasta que el recolector de basura del motor las suprime del sistema.

7. Registrar el controlador de fallo para que muestre el mensaje de error:

```
> p.catch(function(err) { console.error(err.message); })
Promise { <pending> }
>
```

No se ejecuta porque la promesa se encuentra terminada con éxito.

8. Leer el contenido del archivo noexiste.txt, registrando el controlador de error para que muestre el error por la consola:

```
> fsp.readFile("noexiste.txt").catch(function(err) { console.error("Se ha
producido un error:", err.message); })
Promise { <pending> }
> Se ha producido un error: ENOENT: no such file or directory, open 'noexiste.txt'
```

Encadenamiento de promesas

Recordemos que el encadenamiento consiste en enlazar, una detrás de otra, varias promesas relacionadas entre sí.

1. Ir a la consola.
2. Leer el contenido del archivo arch.txt, encadenando dos controladores de éxito:

```
> fsp.readFile("arch.txt").then(function(data) { console.log("PRIMERO:",
data); }).then(function(data) { console.log("SEGUNDO:", data); })
Promise { <pending> }
> PRIMERO: esto es el contenido
```



```
SEGUNDO: undefined
```

¿Por qué el segundo controlador no muestra el contenido? Recordemos que el segundo no se ha asociado a la función readFile(), sino a la promesa devuelta por el método de registro then(), atándose así a esta promesa. Debe ser el controlador de la promesa al que está atado el que indique el parámetro a pasar al segundo controlador. Y esto se hace mediante la sentencia return, o sea, lo que devuelva el primer controlador será lo que se pase al segundo controlador:

```
> fsp.readFile("arch.txt").then(function(data) { console.log("PRIMERO:", data);
return data; }).then(function(data) { console.log("SEGUNDO:", data); })
Promise { <pending> }
> PRIMERO: esto es el contenido

SEGUNDO: esto es el contenido
```

3. Encadenar dos promesas en las que la primera propague un error mediante la sentencia throw:

```
> fsp.readFile("arch.txt").then(function(data) { console.log("PRIMERO:", data);
throw new Error("Esto es el error."); }).then(function(data)
{ console.log("SEGUNDO:", data); }, function(err) { console.error("SEGUNDO
(error):", err.message); })
Promise { <pending> }
> PRIMERO: esto es el contenido

SEGUNDO (error): Esto es el error.
```

Lo anterior se lee mejor como sigue:

```
fsp.readFile("arch.txt").then(function(data) {
  console.log("PRIMERO:", data);
  throw new Error("Esto es el error.");
}).then(
  function(data) { console.log("SEGUNDO:", data); },
  function(err) { console.error("SEGUNDO (error):", err.message); }
)
```

Monitores de promesas

Ahora, vamos a trabajar con monitores de promesas, recordemos, promesas especiales que determinan su estado atendiendo al estado de finalización de otras.

1. Definir un monitor que supervise dos promesas asociadas a la lectura del archivo arch.txt:

```
> p1 = fsp.readFile("arch.txt")
Promise { <pending> }
> p2 = fsp.readFile("arch.txt")
Promise { <pending> }
> mon = Promise.all([p1, p2])
Promise { <pending> }
> mon.then(function(values) { console.dir(values); })
Promise { <pending> }
> [ 'esto es el contenido\n', 'esto es el contenido\n' ]
```

Observe que el controlador del monitor recibe un *array* con los parámetros pasados por el ejecutor de readFile() a resolve().

2. Definir un monitor mediante race():

```
> p1 = fsp.readFile("arch.txt")
Promise { <pending> }
> p2 = fsp.readFile("arch.txt")
Promise { <pending> }
> mon = Promise.race([p1, p2])
Promise { <pending> }
> mon.then(function(value) { console.dir(value); })
Promise { <pending> }
> 'esto es el contenido\n'
```

Ahora, el estado de la promesa monitora depende de la finalización de sólo una promesa de la lista, aquella que termine primero.

