Promesas

Tiempo estimado: 20min

Ya hemos presentado el modelo asíncrono utilizado por Node. JavaScript, de manera intrínseca al lenguaje, define otra manera de hacer lo mismo, conocida formalmente como promesas. El objeto de la presente lección.

La lección comienza con una introducción a las promesas. Sigue con una presentación del ciclo de vida de una promesa y los controladores de estado. A continuación, se muestra cómo crear nuestras propias promesas. Finalmente, se describe cómo se ejecutan los controladores de estado, cómo encadenar promesas y cómo crear monitores de promesas.

Al finalizar la lección, el estudiante sabrá:

- Qué es una promesa.
- Qué es un controlador de estado.
- Cómo usar las promesas.
- Cómo crear promesas.
- Qué es un monitor de promesas.

Introducción

Una promesa (*promise*) es un objeto que representa la ejecución de una función asíncrona. Contiene las operaciones de continuación que el sistema debe ejecutar cuando la operación asíncrona asociada termine.

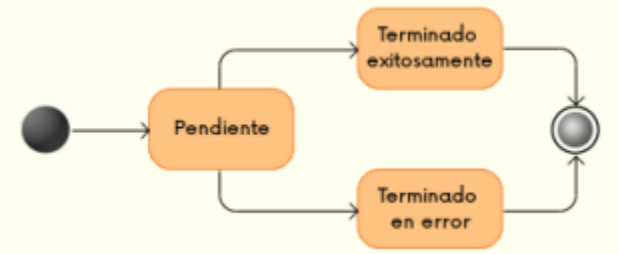
Las promesas son una característica inherente del lenguaje de programación JavaScript. Se han diseñado como medio de respuesta a la finalización de operaciones asíncronas. Se pueden usar tanto en Node como en los navegadores webs.

Recordemos que las funciones de *callback* son una manera de implementar el estilo de paso de continuación. En estas funciones, se define el código del siguiente bloque operativo, el cual debe ejecutarse tras la finalización de la operación de E/S que estamos pendientes de que termine. Generalmente, porque necesitamos los datos que devolverá la E/S. Como ya sabemos, estas funciones se pasan a las operaciones asíncronas, porque son éstas las que acaban invocándolas.

Las promesas son *otra forma* de hacer lo mismo. Con su propia manera de ser y actuar. Las promesas las devuelven las operaciones asíncronas. Ahora, cuando se invoca la operación asíncrona, no se pasa una función *callback*. En su lugar, la función asíncrona devuelve una promesa, en la cual debemos registrar el código del siguiente bloque operativo a ejecutar. Idéntico objetivo, idéntico resultado, manera distinta de trabajar.

Ciclo de vida de una promesa

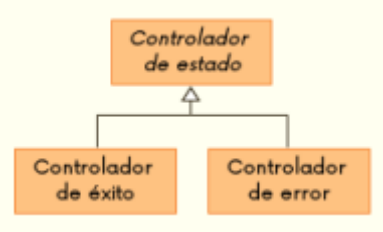
El ciclo de vida de una promesa (*promise life cycle*) representa las distintas fases en las que se puede encontrar una promesa. Cada fase se conoce formalmente como estado (*state*). Una promesa puede encontrarse en uno de tres estados: pendiente, terminado exitosamente o terminado en error.



La promesa se encuentra en estado pendiente (*pending state*), mientras la operación asíncrona está en ejecución, o sea, hasta que termine. Una vez finalizada, pasará a uno de los estados terminados. Si finaliza correctamente, la promesa se pondrá en estado terminado exitosamente (*fulfilled state*). En cambio, si finaliza en fallo o error, pasará al estado terminado en error o fallo (*rejected state*).

Controladores de estado

Así pues, una promesa no es más que un objeto que representa una operación asíncrona y contiene lo que debe ejecutarse cuando finalice. Estos bloques operativos se implementan mediante funciones. Y se conocen formalmente como controladores de estado (*state handlers*) o acciones (*actions*). Atendiendo al estado, se ejecutará un controlador u otro.



El controlador de éxito (*fulfillment handler*) se ejecuta cuando la promesa entra o se encuentra en el estado terminado exitosamente. Mientras que el controlador de error (*rejection handler*), cuando pasa al estado terminado en fallo.

Podemos ver los controladores como funciones *callback*. Pero cada una destinada a contener el bloque de código que debe ejecutarse atendiendo a cómo termine la operación asíncrona. Recordemos que las funciones *callback* tienen que lidiar con ambas situaciones. Los controladores de estado sólo con aquella que les incumbe.

Registro de controladores de estado

El registro de controladores de estado (*state handler register*) es la operación mediante la cual asociamos controladores a la promesa.

Registro de controladores de éxito

Como sabemos, las funciones asíncronas, que se implementan mediante promesas, devuelven la promesa con la que debemos trabajar. A través del método then() de la promesa devuelta, se puede fijar el controlador de éxito:

```
then(handler) : Promise
```

Parámetro	Tipo de datos	Descripción
handler	function	Controlador de éxito.

Al igual que las funciones *callback*, la signatura de la función controladora depende de la operación asíncrona. Pues sus parámetros los pasa la operación asíncrona.

Recordemos la función asíncrona readFile() del módulo fs, a través de la cual obtenemos el contenido de un archivo. Esta función espera el archivo a leer, un objeto de opciones de lectura y finalmente la función *callback* a ejecutar cuando la operación termine. Esta función *callback* tiene dos parámetros: el primero contendrá el objeto error si se ha producido alguno durante la operación de E/S; mientras que el segundo contendrá los datos en un búfer o una cadena de texto. He aquí un ejemplo ilustrativo:

```
fs.readFile("arch.txt", {encoding: "utf8"}, function(err, data) {
  if (error) console.error(err.message);
  else console.log(data);
});
```

Si esta función, en vez de usar una función *callback*, devolviera una promesa, el código anterior sería como sigue:

```
var p = fs.readFile("arch.txt", {encoding: "utf8");

p.then(function(data) {
  console.log(data);
```



```
});
```
Observe que el controlador de éxito no tiene que lidiar con los errores. Para eso está el de error.

Lo anterior se suele escribir más habitualmente como se muestra a continuación:

```
fs.readFile("arch.txt", {encoding: "utf8").then(function(data) {
  console.log(data);
});
```

Registro de controladores de fallo

Si deseamos añadir código específico para procesar el error generado por una operación asíncrona, habrá que añadirlo a su controlador de fallo, el cual se define mediante el método catch():

```
catch(handler) : Promise
```

Parámetro	Tipo de datos	Descripción
handler	function	Controlador de fallo.

La signatura de la función controladora depende también de la operación asíncrona que devolvió la promesa. Pero generalmente consiste en un parámetro a través del cual obtener el error que se ha producido.

Ejemplo:

```
var p = fs.readFile("arch.txt", {encoding: "utf8");

p.then(function(data) {
  console.log(data);
});

p.catch(function(err) {
  console.error(err.message);
});
```

Es posible definir ambos tipos de controladores, mediante el método then(). En este caso, hay que utilizar la siguiente sobrecarga del método:

```
then(fulfillmentHandler, rejectionHandler) : Promise
```

Parámetro	Tipo de datos	Descripción
fulfillmentHandler	function	Controlador de éxito.
rejectionHandler	function	Controlador de fallo.

Creación de promesas

Ya sabemos cómo usar las promesas. Pero todavía no sabemos cómo crear nuestras propias promesas. Para ello, primero hay que introducir el concepto de ejecutor.

El ejecutor (*executor*) es la función que implementa la lógica asíncrona. Y la que, al finalizar su funcionalidad, debe invocar el controlador de estado correspondiente.

Las promesas son instancias de la clase Promise que viene de fábrica con el motor de JavaScript que soporta esta característica. En nuestro caso, node. Su método constructor es como sigue:

```
constructor(executor)
```

Parámetro	Tipo de datos	Descripción
executor	function	Función ejecutora: fn(resolve, reject).

La función ejecutora espera dos argumentos, ambos de tipo función. El primero, resolve, es la función que debe invocar cuando finalice su funcionalidad y lo haga con éxito. Así transitará la promesa del estado pendiente al finalizado con éxito y, entonces, se ejecutará el controlador de éxito. Los argumentos que se pasen en la invocación de resolve() serán los que se pasen a la función controladora.

En cambio, si el ejecutor encuentra un fallo y decide que debe terminar e informar de ello, entonces invocará el parámetro reject. Poniendo así la promesa en estado terminado en fallo. Y solicitando a la

promesa la ejecución del controlador de fallo. Lo que se pase a este parámetro, se pasará al controlador de error.

Veamos un ejemplo. Vamos a implementar la función fs.readFile() mediante una promesa:

```
function readFilePromise(file, opts) {
  return new Promise(function executor(resolve, reject) {
    fs.readFile(file, opts, function(err, data) {
      if (err) reject(err);
      else resolve(data);
    });
  });
}
```

La función no hace más que esperar dos argumentos: el archivo a leer y las opciones de lectura. A diferencia de readFile() que no devuelve nada, nuestra implementación sí: la promesa a través de la cual indicar qué debe ejecutarse cuando se haya leído el archivo.

Esta promesa se implementa mediante una función ejecutora que no hace más que leer el archivo asíncronamente, mediante readFile(). Por lo que le pasa una función *callback*. La cual finalmente, invocará el parámetro resolve() o reject() según finalice la operación de E/S.

Cuando desarrolle un paquete, puede utilizar el medio de continuación con el que se sienta más a gusto: *callback* o promesas. Algunos paquetes implementan ambos, dotándolo de mayor flexibilidad y adaptándose mejor a las preferencias de sus usuarios.

Gestión de errores

No se recomienda que una función asíncrona propague un error mediante la sentencia throw. En su lugar, se recomienda capturar el error e invocar el parámetro reject() pasando el error. Cuando se propaga un error mediante throw, se habla de error no controlado (*unhandled error*).

En el momento de escribir estas líneas, noviembre de 2017, node informa que, en un futuro no muy lejano, los errores no controlados producirán su terminación con un código de salida distinto de cero. Por lo tanto, hay que tener mucho cuidado cuando desarrollemos la función ejecutora y las controladoras de estado para que no propaguen errores no controlados.

Si lo deseamos, podemos registrar dos controladores de evento en el proceso node: rejectionHandled y unhandledRejection. El primer evento, rejectionHandled, se genera cuando se propaga un error y la promesa dispone de controlador de error. Si es así, invocará el controlador de fallo de la promesa y generará el evento. En cambio, el evento unhandledRejection se genera cuando no se propaga un error y la promesa no dispone de controlador de fallo registrado.

Los controladores para estos eventos se pueden registrar mediante la función process.on():

```
function on(eventName, handler)
```

Parámetro	Tipo de datos	Descripción
eventName	string	Nombre del evento.
handler	function	Controlador del evento.

La signatura de la función controladora depende del evento:

- Evento rejectionHandled: fn(promise).
- Evento unhandledRejection: fn(reason, promise).

Funciones Promise.resolve() y Promise.reject()

En algunas ocasiones, cuando se devuelve la promesa, ya se sabe de antemano su resultado: finalizado exitosamente o en fallo. Si lo deseamos, podemos obtener una promesa ya finalizada. Para ello, se usa las funciones o métodos estáticos resolve() y reject() de la clase Promise:

```
static resolve(value) : Promise
static reject(value): Promise
```

Parámetro	Tipo de datos	Descripción

value	object	Argumento a pasar al controlador.

Veamos un ejemplo:

```
function readFilePromise(file, opts) {
  return Promise.resolve(fs.readFileSync(file, opts));
}
```

Lo anterior es lo mismo que lo siguiente:

```
function readFilePromise(file, opts) {
  return new Promise(function executor(resolve, reject) {
    resolve(fs.readFileSync(file, opts));
  });
}
```

Se utilizan básicamente cuando el acceso es síncrono y se desea implementar como si fuera asíncrono.

Ejecución de los controladores de estado

En este punto, vamos a prestar un poco de atención al registro y ejecución de los controladores de estado. Recordemos que cuando ejecutamos algo asíncronamente, básicamente lo que estamos haciendo es realizar una E/S. Node solicita la E/S, pero no se detiene, sino que sigue con su trabajo.

Debemos tener claro, pues, que cuando ejecutemos la operación asíncrona de turno, lo que se hará es eso. Solicitarla, sin esperar a que finalice. Por lo que cuando invoquemos una función asíncrona que devuelve una promesa, en la mayoría de los casos ésta no habrá terminado.

Por otra parte, tal como vimos en el ejemplo anterior, sus controladores se registran mediante los métodos then() y/o catch(). Y cuando se hace, generalmente la promesa se encuentra en estado pendiente.

Resumiendo, los controladores se registrarán antes de finalizar la operación de E/S y se ejecutarán cuando la promesa transite a uno de los dos estados de finalización. Hasta aquí todo normal. Más o menos como las funciones *callback*.

Pero hay que decir que se puede registrar un controlador en cualquier momento. Tanto si la promesa se encuentra en estado pendiente como si está en uno de finalización. Cuando lo registremos con la operación asíncrona finalizada, el controlador se ejecutará también. Es importante tenerlo claro. Los controladores se ejecutan una vez la operación de E/S ha finalizado. Y lo harán asíncronamente como no podía ser de otra manera.

Cadenas de promesas

Una cadena de promesas (*promise chain*) es una serie de promesas relacionadas entre sí.

Básicamente, las promesas pueden tener dos tipos de creadores, las funciones asíncronas y otras promesas. Cuando se crea una promesa, mediante la clase Promise, el que la crea es su origen (*source*). Define el código ejecutor y este código es el responsable de notificar cuando la operación ha finalizado mediante los parámetros resolve() y reject(). A esta promesa se le conoce como promesa principal (*primary promise*).

Por otra parte, volvamos atrás y recordemos los métodos de registro de los controladores de estado: then() y catch(). Estos métodos devuelven a su vez promesas. Las cuales se atan o asocian a la terminación de los controladores, no al origen de toda la cadena. Cuando registramos nuevos controladores a promesas de promesas, estamos formando una cadena. Cada una de ellas, representa un eslabón de la cadena. Y se van ejecutando a medida que su origen finaliza. A estas promesas, se les conoce como promesas secundarias (*secondary promises*).

Tal como vimos, cuando finaliza la operación asíncrona, se ejecutará los controladores asociados a su promesa. Cuando los controladores registrados finalizan, si la promesa tiene asociada otra promesa, entonces invocará los controladores de su promesa. Si el controlador origen finaliza en éxito, invocará el controlador de realización; en otro caso, el de error. Así de simple.

Parámetros de los controladores secundarios

Recordemos que los parámetros de los controladores de la promesa principal se pasan a través de los parámetros resolve() y reject() de la función ejecutora. En cambio, los controladores de estado no disponen de estos parámetros. Y hay que decir que los parámetros pasados a la promesa principal no son los que recibirán las secundarias.

Así pues, ¿cómo se pasan los parámetros a lo largo y ancho de la cadena de promesas? Sencillo: lo que devuelva un controlador de estado, mediante su sentencia return, será lo que se pase como parámetro al siguiente de la cadena.

Monitores de promesas

Un monitor de promesas (*promise monitor*) es un tipo especial de promesa que supervisa la ejecución de una o más promesas. Como es una promesa, puede registrar controladores de estado. Estos controladores se ejecutan cuando el monitor ha finalizado su trabajo e invocará un controlador u otro atendiendo al resultado de la monitorización. Si todo va bien, ejecutará el controlador de éxito. En otro caso, el de error.

Para crear monitores de promesas, hay que utilizar las funciones o métodos estáticos all() y race() de la clase Promise:

```
static all(promises) : Promise
static race(promises) : Promise
```

Parámetro	Tipo de datos	Descripción
promises	Iterable	Lista de promesas a supervisar.

El método estático Promise.all() espera a que todas las promesas indicadas finalicen. Ejecutando su controlador de estado correspondiente atendiendo al estado global de las promesas supervisadas. Ahora bien, Promise.race() finaliza cuando lo hace una de las promesas dadas. No espera a que finalicen todas, sólo espera a que lo haga una. El estado de la primera terminada será el estado de la promesa monitor.

La signatura del controlador de estado de éxito del método all() es fn(values), donde el parámetro es un *array* de los valores devueltos por el origen de la promesa. En cambio, el de race() es fn(value), el valor devuelto por la primera promesa finalizada.

Ejemplo:

```
Promise.all([p1, p2, p3]).then(
  function(values) { … },
  function(err) { … }
)
```

