UT09.2: Promesas

Contenido

1. Introducción

JavaScript tiene muchas funciones que permiten ejecutar acciones de manera *asíncrona*, es decir, acciones que sabemos cuándo comienzan, pero no sabemos cuándo terminarán. Un ejemplo podría ser `setTimeout`.

Veamos un ejemplo de gestión de peticiones asíncronas. Tenemos que realizar 3 tareas de manera secuencial en la que se garantice el orden.

```
function doTask(task) {
  console.log(`Tarea: ${task}.`);
}
doTask('task1');
doTask('task2');
doTask('task3');
```

Ahora vemos que la tarea número 1 es asíncrona, se va a empezar a ejecutar, pero no sabemos exactamente cuando terminará. En este caso las tareas 2 y 3 terminan antes que la tarea número 1. En este caso vemos con la tarea 1 se retrasa sobre las demás.

```
setTimeout(() => doTask('task1'), 2000);
doTask('task2');
doTask('task3');
```

Vamos a garantizar que se ejecuten en el orden correcto las 3 tareas. A la función le vamos a añadir un nuevo parámetro con una función de callback que se invoque cuando haya terminado de realizarse la tarea.

```
function doTask(task, response) {
  console.log(`Tarea: ${task}.`);
  if (response) response();
}
```

Para garantizar el orden, invocamos a cada tarea como la función de callback.

```
doTask('task1', () => {
  doTask('task2', () => {
    doTask('task3');
  });
});
```

Aunque insertemos el un *timeout*, el orden se respeta al realizar las invocaciones desde la función de callback.

```
setTimeout(() => {
  setTimeout(doTask('task1', () => {
    setTimeout(doTask('task2', () => {
      doTask('task3');
    }), 1500);
  }), 1000);
}, 2000);
```

Podemos complicar aún más el escenario si realizamos la gestión de errores. Añadimos un parámetro a la función para que invoque una función de callback en caso de detectar algún error.

```
function doTask(task, response, error) {
  console.log(`Tarea: ${task}.`);
  if (error) {
    error(task);
  } else if (response) response();
}
```

Creamos una función para la gestión del error.

```
function error(task) {
  console.error(`Error en la tarea: ${task}`);
}
```

Y realizamos la invocación pasando algún error.

```
setTimeout(() => {
  setTimeout(doTask('task1', () => {
    setTimeout(doTask('task2', () => {
      doTask('task3');
    }, error), 1500);
  }), 1000);
}, 2000);
```

Podemos gestionar varias tareas asíncronas anidando llamadas de callback, el problema es que a partir de la tercera llamada el código se hace poco legible. Este tipo de escenario es conocido como **pirámide infernal** por la forma en la que va creciendo el código hacía la derecha debido a la anidación.

Las promesas intentan resolver este escenario.

2. Objeto `Promise`

Una promesa en una petición asíncrona representa un valor que pude estar **disponible ahora**, o que podría retrasarse para obtenerse en un **futuro**, o que **nunca** podría obtenerse. Las promesas funcionan como un *Proxy* para obtener un valor, y nos van a permitir que nuestra codificación funcione de forma síncrona, aunque realmente sea asíncrona, mejorando la legibilidad del código.

Para crear una promesa le pasamos como argumento una función denominada *ejecutor*, esta función es la encarga de realizar el trabajo asíncrono y, por tanto, no sabemos cuándo puede terminar, se trata de una carga de datos por la red, o la apertura de una conexión a una base de datos. El *ejecutor* comienza su ejecución de manera automática.

El *ejecutor* tiene dos argumentos. El primero es una función que *resolverá* la promesa, es decir, esta función necesita los datos generados por el *ejecutor* para realizar su misión. El segundo argumento es una función que *rechazará* la promesa.

```
let promise = new Promise(function(resolve, reject) {
  // Ejecutor: Realiza el trabajo asíncrono.
});
```

Los estados en los que podemos encontrar la promesa son los siguientes:

- **Pendiente**: La promesa no está ni cumplida ni rechazada, se encuentra en el estado inicial.
- **Cumplida**: La promesa se ha completado satisfactoriamente.
- **Rechazada**: La promesa ha fallado.

En la programación asíncrona, las promesas ofrecen ciertas garantías de obtención de un resultado, que con los objetos tradicionales no tenemos. Vamos a obtener garantías de que las funciones de callback se invoquen una vez finalizado la cadena de eventos que han implicado la petición asíncrona, pudiendo encadenar llamadas varias veces en el orden que estamos esperando, y no en el orden que son recibidas las respuestas.

3. Resolviendo una promesa

Vamos a estudiar un ejemplo de cómo podemos consumir una promesa. Vamos a crear una función que simule la apertura de una conexión contra una base de datos.

```
function openConecction(BD) {
  if (BD) return { connection: true };
  throw new Error('No se ha podido abrir la conexión.');
}
```

Creamos una promesa que simule la apertura de la conexión dentro del ejecutor. Si en la invocación de la apertura nos e produce ningún error, debemos llamar a la función que resuelve la promesa pasando como resultado el objeto obtenido. En caso de detectar algún error, invocaremos a la función para rechazar la promesa pasando el error.

```
const promise = new Promise((resolve, reject) => {
  let connection;
  setTimeout(() => {
    try {
      connection = openConecction('BD');
      if (connection) resolve(connection);
    } catch (error) {
      reject(error);
    }
  }, 2000);
});
```

3.1. Método then()

El método `then()` consume la promesa, sus argumentos son las funciones que resuelve o rechazan la promesa, por lo que serán las funciones de callback que invocará el ejecutor.

```
promise.then(
  (result) => {
    console.log(result.connection);
  },
  (error) => {
    console.log(error);
  },
);
```

Podemos jugar a formar el error modificando el argumento en la función `openConnection()`.

3.2. Método catch()

Si se utiliza actuaría como la función de rechazo de una promesa. El siguiente código es similar al del ejemplo anterior implementado con el `then()`.

```
promise.then(
  (result) => {
    console.log(result.connection);
  },
).catch((error) => {
  console.log(error);
});
```

3.3. Método finally()

Funciona igual que con el uso de las excepciones. Se invocará independientemente de que se haya producido un error o no. El método se utiliza para liberar los recursos que se hayan podido utilizar durante la ejecución de la promesa.

```
promise.then(
  (result) => {
    console.log(result.connection);
  },
).catch((error) => {
  console.log(error);
}).finally(() => {
  console.log('Finalizamos la ejecución');
});
```

4. Encadenamiento

Las promesas las podemos encadenar como hacemos con los métodos de una clase, lo que nos permite gestionar el orden de ejecución de tareas asíncronas, y resolver el problema que teníamos en la introducción.

4.1. Encadenando manejadores

El método `then()` devuelve una promesa, ya sea desde el manejador que resuelve la promesa o desde el que lo rechaza, por lo que si devolvemos un valor o lanzamos una excepción se creará una promesa que se resolverá o rechazará de forma automática al pasarse al siguiente método `then()`.

Simulamos una petición asíncrona en la que obtenemos un dato, mediante un `setTimeout()`, la petición tiene una duración de 1 segundo. En el ejecutor invocamos el manejador de resolución pasando la respuesta obtenida como resultado, en este caso un entero, que será atendido en el manejador del método `then()`, donde realizamos la primera tarea.

El manejador devuelve un nuevo entero, este es empaquetado como resultado de una nueva promesa, con lo que la atendemos en una encadenación con otro método `then()` para realizar la segunda tarea. Nuevamente devolvemos un entero para repetir el proceso y realizar una tercera tarea en la siguiente encadenación.

```
new Promise((resolve, reject) => {
  setTimeout(() => resolve(1), 1000);
}).then((result) => {
  doTask('Tarea1');
  console.log(result); // 1
  return result * 2;
}).then((result) => {
  doTask('Tarea2');
  console.log(result); // 2
  return result * 2;
}).then((result) => {
  doTask('Tarea3');
  console.log(result); // 4
  return result * 2;
});
```

Un error clásico es que en lugar de encadenar los `then()` los invocamos sobre la promesa. En este caso estamos añadiendo varios manejadores sobre la misma promesa, no encadenando el resultado de uno con el siguiente, por lo que todos obtendrán el mismo resultado de la promesa inicial.

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => resolve(1), 1000);
});

promise.then((result) => {
  doTask('Tarea1');
  console.log(result); // 1
  return result * 2;
});
```

```
promise.then((result) => {
  doTask('Tarea2');
  console.log(result); // 2
  return result * 2;
});

promise.then((result) => {
  doTask('Tarea3');
  console.log(result); // 4
  return result * 2;
});
```

4.2. Retornando promesas

Las tareas realizadas en el método `then()` podrían ser asíncronas también, por lo que dentro del manejador deberíamos crear una nueva promesa. El método `then()` puede devolver una promesa, satisfecha o fallida según el caso, lo que nos permite encadenarlas.

En el siguiente ejemplo es similar al anterior, pero cada método `then()` devuelve una nueva promesa, obteniendo el resultado una vez que la promesa ha sido satisfecha. Podemos observar que el resultado es obtenido un segundo después de la invocación.

```
new Promise((resolve, reject) => {
  setTimeout(() => resolve(1), 1000);
}).then((result) => {
  doTask('Tarea1');
  console.log(result); // 1
  return new Promise((resolve, reject) => {
    setTimeout(() => resolve(result * 2), 1000);
  });
}).then((result) => {
  doTask('Tarea2');
  console.log(result); // 2
  return new Promise((resolve, reject) => {
    setTimeout(() => resolve(result * 2), 1000);
  });
}).then((result) => {
  doTask('Tarea3');
  console.log(result); // 4
  return result * 2;
});
```

4.3. Objetos thenables

Los manejadores pueden devolver un objeto denominado **thenable**, es decir, un objeto que implementa un método `then()`. Este objeto es compatible con las promesas, por lo que podemos extender la funcionalidad sin tener que heredar de un objeto `Promise`.

```
class Thenable {
  constructor(num) {
    this.num = num;
  }

  then(resolve, reject) {
    console.log(resolve);
    setTimeout(() => resolve(this.num * 2), 1000);
  }
}

new Promise((resolve) => resolve(1))
  .then((result) => new Thenable(result), // (*)
  )
  .then(console.log);
```

5. Manejo de errores

Cuando una promesa encadenada es rechazada, el control pasa al manejador de rechazo más cercado, o en último caso en el método `catch()`. Tanto en los manejadores como en los ejecutores, las promesas tienen embebido un control try-catch implícito, por lo que la excepción será atrapada por él y se gestionará como un rechazo.

En el siguiente código forzamos un error en la gestión de la tarea 1, el cual será gestionado en el método `catch()` recibiendo el objeto con el error.

```
new Promise((resolve, reject) => {
  setTimeout(() => resolve(1), 1000);
}).then((result) => {
  doTask('Tarea1');
  console.log(result); // 1
  throw (new Error('Error en tarea 1'));
  return result * 2;
}).then((result) => {
  doTask('Tarea2');
  console.log(result); // 2
  return result * 2;
}).then((result) => {
  doTask('Tarea3');
  console.log(result); // 4
  return result * 2;
})
  .catch((error) => {
    console.log(error);
  });
```

En caso de tener un manejador de rechazo previo, éste atendería la excepción. Tenemos que tener en cuenta que la cadena de promesas sigue, ya que el manejador de rechazo devuelve una promesa que será gestionado por el próximo `then()`. En el manejador de rechazo podemos recuperar el error, lo que implicaría devolver un nuevo resultado, o bien relanzar la excepción.

```
new Promise((resolve, reject) => {
  setTimeout(() => resolve(1), 1000);
}).then((result) => {
  doTask('Tarea1');
  console.log(result); // 1
  throw (new Error('Error en tarea 1'));
  return result * 2;
}).then((result) => {
  doTask('Tarea2');
  console.log(result); // 2
  return result * 2;
}, (error) => {
  console.log('Error capturado en tarea 2');
  console.log(error);
  if (error instanceof URIError) {
    return 1;
  }
  console.log('No puedo recuperar el error');
  throw error;
}).then((result) => {
  doTask('Tarea3');
  console.log(result); // 4
  return result * 2;
}).catch((error) => {
    console.log(error);
});
```

6. API del objeto `Promise`

El objeto `Promise` dispone de 6 métodos estáticos.

6.1. `Promise.all()`

Se trata de un método que nos permite ejecutar en paralelo una serie de promesas y obtener el resultado cuando todas estén resueltas. El método recibe un objeto iterable con las promesas.

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

El siguiente ejemplo recibe un array de promesas la cuales serán atendidas por el manejador recibiendo como argumento un array con el resultado de cada una de ellas en el orden que las hayamos definido, no en el orden que son recuperados los resultados.

```
Promise.all([
  new Promise((resolve) => setTimeout(() => resolve(1), 3000)), // 1
  new Promise((resolve) => setTimeout(() => resolve(2), 2000)), // 2
  new Promise((resolve) => setTimeout(() => resolve(3), 1000)), // 3
]).then(console.log); // [1,2,3]
```

Si una de las promesas es rechazada, el resto se ignoran.

```
Promise.all([
  new Promise((resolve, reject) => setTimeout(() => resolve(1), 1000)),
  new Promise((resolve, reject) => setTimeout(() => reject(new
Error('Algo ha ido mal')), 2000)),
  new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000)),
]).catch(console.log);
```

6.2. Promise.allSettled ()

Funciona igual que el anterior, espera que se resuelvan todas las promesas para ser procesadas, pero en caso de que una promesa sea rechazada se empaqueta igualmente en el array. En este caso el array no tiene los resultados directamente, sino que utiliza una objeto literal para diferenciar el cómo se ha resuelto cada promesa.

- `{status:"fulfilled", value:result}` para respuestas exitosas,
- `{status:"rejected", reason:error}` para errores.

```
Promise.allSettled([
  new Promise((resolve, reject) => setTimeout(() => resolve(1), 1000)),
  new Promise((resolve, reject) => setTimeout(() => reject(new
Error('Algo ha ido mal')), 2000)),
  new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000)),
]).then((results) => {
  results.forEach((result) => {
    if (result.status == 'fulfilled') {
      console.log(`Valor: ${result.value}`);
    }
    if (result.status == 'rejected') {
      console.log(`Razón: ${result.reason}`);
    }
  });
});
```

6.3. Promise.race()

Devuelve la primera promesa que haya sido resultado de manera exitosa o mediante un error.

```
Promise.race([
  new Promise((resolve) => setTimeout(() => resolve(1), 3000)),
  new Promise((resolve, reject) => setTimeout(() => reject(new
Error('Algo ha ido mal')), 2000)),
  new Promise((resolve) => setTimeout(() => resolve(3), 1000)),
]).then(console.log); // 3
```

6.4. `Promise.any()`

Es igual que el anterior, pero devuelve la primera promesa exitosa ignorando si se ha producido alguna de rechazo.

```
Promise.any([
  new Promise((resolve) => setTimeout(() => resolve(1), 2000)),
  new Promise((resolve, reject) => setTimeout(() => reject(new
Error('Algo ha ido mal')), 1000)),
  new Promise((resolve) => setTimeout(() => resolve(3), 3000)),
]).then(console.log); // 1
```

6.5. `Promise.resolve()`

`Promise.resolve(value)` crea una promesa resuelta con el resultado value.

6.6. `Promise.all()`

`Promise.reject(error)` crea una promesa rechazada con error.

7. Async/await

Permite especificar una sintaxis más cómoda para trabajar con promesas.

7.1. Funciones async

Declarar una función como `async` indica que la función será asíncrona por lo que devolverá una promesa. El siguiente código devuelve una promesa resuelta con resultado de un entero.

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

El resultado de la función queda devuelto dentro de una promesa.

Podemos definir métodos de clase como `async`.

7.2. Await

`await` hace que JavaScript espere hasta que la promesa responda y devuelve su resultado.

```
async function f() {
  const promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve('¡Hecho!'), 1000);
  });
  const result = await promise;
  console.log(result);
}
f();
```

Es una forma de obtener el resultado de un `then()`, lo que hace que el código sea más legible.

Lógicamente `await` solo se puede ejecutar dentro del contexto de una función `async`.

7.3. Gestión de errores

La promesa que estamos esperando puede ser rechazada, por lo que generará el error. En este contexto tenemos que tener un bloque `try-catch` que envuelva a `await`, en caso de producirse el rechazo, el bloque catch capturará la excepción.

```
async function f() {
  const promise = new Promise((resolve, reject) => {
    setTimeout(() => reject(new Error('Error en la promesa')), 1000);
  });
  try {
    const result = await promise;
    console.log(result);
  } catch (error) {
    console.log(error);
  }
}
f();
```