ES6 Ajax
Promise & fetch

Antonio Espín Herranz

Contenidos

- Promesas
- Enlazar promesas:
  - En serie
  - En paralelo
- Fetch:
  - Permite el uso de promesas lo que genera un código mucho más limpio.
  - Reemplaza al antiguo XMLHttpRequest para peticiones Ajax

Promesas

- Las **promesas** son las herramientas de los lenguajes de programación que nos sirven para **gestionar situaciones futuras** en el flujo de ejecución de un programa.

- Es un concepto que se utiliza desde hace poco en JS pero ya viene implementándose desde los años 70 en el mundo de la programación.

- En JavaScript se introducen en el standard **ES6**.

Gestión de Promesas

- En una promesa podrán darse **dos situaciones** posibles:
  - **then**: usado para indicar qué hacer en caso, que la promesa se haya ejecutado con éxito.
  - **catch**: usado para indicar qué hacer en caso, que durante la ejecución de la operación se ha producido un error.
- Ejemplo: *En este caso el método set nos devuelve una promesa.*

```
referenciaFirebase.set(data)
  .then(function(){
    console.log('el dato se ha escrito correctamente');
  })
  .catch(function(err) {
    console.log('hemos detectado un error', err');
  });
```

Gestión de Promesas

- El código se puede encadenar todo seguido (se parte por claridad)

```
referenciaFirebase.set(data).then(function(){
  console.log('el dato se ha escrito correctamente');
}).catch(function(err) {
  console.log('hemos detectado un error', err');
});
```

- Para las funciones de callback (then y catch) se pueden utilizar funciones **arrow**.

Gestión de Promesas

- Las promesas pueden devolver datos. Se recibirán como parámetro en la función callback que estamos adjuntando a then()
- En este caso es datoProcesado.

```
funcionQueDevuelvePromesa()
  .then( function(datoProcesado){
    //hacer algo con el datoProcesado
  })
```

La función llamada que devuelve una Promesa también puede recibir parámetros

No es obligatorio implementar la parte de catch e incluso then. O se puede tener más de un then. **Se pueden encadenar!!**

Implementar Promesas

- Las funciones de JavaScript pueden devolver promesas.
- Por ejemplo, la función **fetch** (para realizar peticiones Ajax) devuelve una promesa.

- El objetivo de una promesa: es hacer algo que puede llevar un tiempo y luego tiene la capacidad de informar si ha ido bien o mal.
  - Por ejemplo, una petición Ajax con fetch.
    - Se realiza una petición a una URL
    - Si va todo bien, nos devuelve los datos: then
    - Si se produce algún error: catch

Implementar Promesas

- La promesa queda identificada en JS por el objeto Promise y necesita por parámetro una función que será la encargada de realizar el procesamiento que puede llevar algo de tiempo de proceso.

- La función recibirá dos parámetros que será dos funciones:
  - Una para procesar los casos de éxito → **resolve**. Cuando finaliza la promesa con éxito
  - Y otra que se ejecutará cuando queramos terminar la promesa informando de un fracaso → **reject**.

Ejemplo

```
function hacerAlgoPromesa() {
  return new Promise( function(resolve, reject){
    console.log('hacer algo que ocupa un tiempo...');
    setTimeout(resolve, 1000);
  })
}
```

- La llamada podría ser:

```
hacerAlgoPromesa()
  .then( function() {
    console.log('la promesa terminó.');
  })
```

Encadenamiento de Promesas

- Las promesas se pueden encadenar.
- Se puede ejecutar la función varias veces.
- hacerAlgoPromesa()
- .then(hacerAlgoPromesa)
- .then(hacerAlgoPromesa)
- .then(hacerAlgoPromesa)

Implementación de Promesas

- **resolve** / **reject**: las funciones que se utilizan en una promesa para indicar si va bien o no.

- La promesa se utiliza para resolver algún tipo de petición asíncrona que tardará un tiempo en resolverse y que se puede dar una situación de éxito (resolve) o de fracaso (reject).

- Se pueden utilizar funciones arrow en la implementación.

Ejemplo: Función que devuelve una promesa

```
function devuelvePromesa(){
  return new Promise( (resolve, reject) => {
    setTimeout(() => {
      let todoCorrecto = true;
      if (todoCorrecto) {
        resolve('Todo ha ido bien');
      } else {
        reject('Algo ha fallado)
      }
    }, 2000)
  })
}
```

La función: **setTimeout**
Recibe una función y un número que representan
Los milisegundos.
La función se ejecuta después de x milisegundos.

La función en este caso se indica como función arrow

Promise recibe otra función (arrow) con dos parámetros
resolve y reject

Ejemplo: Llamada a la función anterior

- devuelvePromesa()
  - .then(respuesta => console.log(respuesta))
  - .catch(error => console.log(error))

- **then** recibe la respuesta indicada en el **resolve**() y **catch**() el error indicado en el **reject**

Ejemplo utilizando fetch

```
function obtenerTexto(url) {
  return new Promise( (resolve, reject) => {
    fetch(url)
      .then(response => {
        if(response.ok) {
          return response.text();
        }
        reject('No se ha podido acceder a ese recurso. Status: ' + response.status);
      })
      .then( texto => resolve(texto) )
      .catch (err => reject(err) );
  });
}
```

Llamada a la función anterior

```
obtenerTexto('test.txt')
  .then( texto => console.log(texto) )
  .catch( err => console.log('ERROR', err) )
```

Promesas en Secuencia

- Las promesas se pueden ejecutar en secuencia una detrás de otra, se van enlazando.
- Normalmente ejecutamos una promesa, obtenemos el resultado y con ese resultado tenemos que ejecutar otra y así sucesivamente.
  - Este proceso en JS se realiza de una forma limpia con las promesas para evitar el anidamiento de callbacks.

- Esta situación es muy típica con la función fetch, donde tendremos que encadenar dos promesas:
  - Con una hacemos una petición al servidor, obtenemos una respuesta
  - Y con la otra nos quedamos con el cuerpo del objeto.

Ejemplo

```
fetch("https://jsonplaceholder.typicode.com/todos/")
  .then( response => response.json() )
  .then( json => console.log(json) )
```

El método **response.json()** devuelve otra promesa que se encadena.

- En la primera promesa obtenemos los datos del servidor y devolvemos el json asociado.
- En la segunda podemos procesar el json.
- El orden que se escriben los then marca el orden de ejecución de las promesas.

Encadenamientos

- Por ejemplo, 2 peticiones a servicios Web y esperar entre las dos peticiones:

```
fetch("https://jsonplaceholder.typicode.com/todos/")
  .then( response => response.json())
  .then( json => console.log(json))
  .then(() => esperar(5000))
  .then( res => console.log(res))
  .then(() => fetch("https://pokeapi.co/api/v2/pokemon/1"))
  .then( response => response.json())
  .then( json => console.log(json));
```

Promesas en Paralelo

- En otras ocasiones las promesas se pueden ejecutar en paralelo porque no depende una de otra y se puede ahorrar tiempo.

```
esperar(5000).then((res) => console.log("Uno"));
esperar(1000).then((res) => console.log("Dos"));
esperar(3000).then((res) => console.log("Tres"));
console.log("Fin!");
```

Promesas en Paralelo

- Hay veces que se pueden ejecutar todas a la vez pero necesitamos que hayan terminado todas antes de ejecutar otro código.

- Disponemos del método **Promise.all()** en JS que recibe un array de promesas y cuando termina la última devuelve un array con todas las respuestas de las promesas.

```
Promise.all([
  esperar(5000),
  esperar(1000),
  esperar(3000)
]).then( respuestas => {
    console.log("Fin!");
});
```

Promesas en Paralelo

- Una vez que hayan terminado todas las promesas del array se pueden procesar las respuestas:

```
Promise.all([
  esperar(5000),
  esperar(1000),
  esperar(3000)
]).then( respuestas => {
    for (let i in respuestas) {
      console.log(respuestas[i]);
    }
  }
);
```

Métodos y propiedades de Fetch

- Tener en cuenta que cuando llamamos a **fetch** con la promesa **then** nos devuelve el objeto (promesa) y se le aplica el método **text**() o **json**() según queramos un formato u otro.

- Y tenemos que **enlazar otra promesa then** para poder acceder al texto o json.

- Ejemplo:

```
fetch(myRequest)
  .then((response) => response.text())  // Obtener el texto y luego a la siguiente ya lo tenemos
  .then((text) => {
        Hacer algo con text. Normalmente modificar el DOM de la página
  });
```

https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch

Objetos

- A fetch se le puede enviar un objeto Request o una url (String) que represente el recurso que estamos solicitando.
- El objeto devuelto por fetch es Response.
- La función fetch acepta un segundo parámetro que será un objeto de tipo Headers. Donde se puede configurar propiedades con el método append.

- Por ejemplo:
  - req = new Request('un recurso');
  - fetch(req).then …

- O:
  - fetch('un recurso').then …

Response

- Disponemos de métodos:
  - blob(): Para recuperar imágenes y ficheros binarios
  - json(): Respuesta en json
  - text(): Respuesta en texto

  - Para obtener un resultado en XML tendremos que obtener el contenido en texto y luego parsearlo a XML.

```
fetch(url)
    .then(response =>response.text())
    .then(datos => new DOMParser().parseFromString(datos, "text/xml"))
    .then(data => {/* Hacer algo con el XML */})
```

Propiedades Response

- Se pueden ver al imprimir:

```
fetch("logo.png")
  .then((response)  => {
     console.log("ok:  "+response.ok);
     console.log("status:  "+response.status);
     console.log("statustext:  "+response.statustext);
     console.log("type:  "+response.type);
     console.log("url:  "+response.url);
     console.log("bodyUsed:  "+response.bodyUsed);
     console.log("body:  "+response.body);

     console.log("Headers:\n");
     response.headers.forEach((v,  k)=> {console.log(k+“:  "+v)});
  })
```

async / await

- Se utilizan para realizar peticiones asíncronas. Se lanza una petición y continuamos con el flujo del programa.
- La palabra clave **async** convierte un método en un método **asincrónico**, lo que permite usar la palabra clave **await** en su cuerpo.
  - Cuando se aplica la palabra clave **await** , se suspende el método de llamada y se cede el control al autor de la llamada hasta que se completa la tarea esperada.
  - Se utilizan siempre que una función pueda tardar un tiempo en ejecutarse.
- https://javascript.info/async-await

Ejemplo

```
function resolveAfter2Seconds() {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve('resolved');
    }, 2000);
  });
}

async function asyncCall() {
  console.log('calling');
  const result = await resolveAfter2Seconds();
  console.log(result);
  // expected output: "resolved"
}

asyncCall();
```

Enlaces

- https://programadorwebvalencia.com/cursos/javascript/ajax/

- Promesas:
  - https://desarrolloweb.com/articulos/introduccion-promesas-es6.html
  - https://desarrolloweb.com/articulos/implementar-promesas-resolve-reject.html

- Para obtener resultados aleatorios: Le indicamos con el parámetro el número de resultados que queremos obtener:
  - https://randomuser.me/api/?results=10