Javascript: Promesas

Promesas

- Objeto que representa la finalización o fracaso de una operación asíncrona
- Ejemplo:
  - Imaginemos una función que genera un archivo de sonido de forma asíncrona a partir de una configuración
  - Dos funciones de retorno:
    - Éxito: casos en que se crea exitosamente
    - Fallo: casos de error

Promesas

- El código se puede ver como:

```
function exitoCallback(resultado) {
  console.log("Archivo de audio disponible en la URL " + resultado);
}

function falloCallback(error) {
  console.log("Error generando archivo de audio " + error);
}

crearArchivoAudioAsync(audioConfig, exitoCallback, falloCallback);
```

Promesas

- Al usar promesas:
  - Se puede adjuntar funciones de retorno
  - Si `crearArchivoAudioAsync` se escribe para que retorne una promesa:

```
crearArchivoAudioAsync(audioConfig).then(exitoCallback, falloCallback);
```

  - Es una versión corta de:

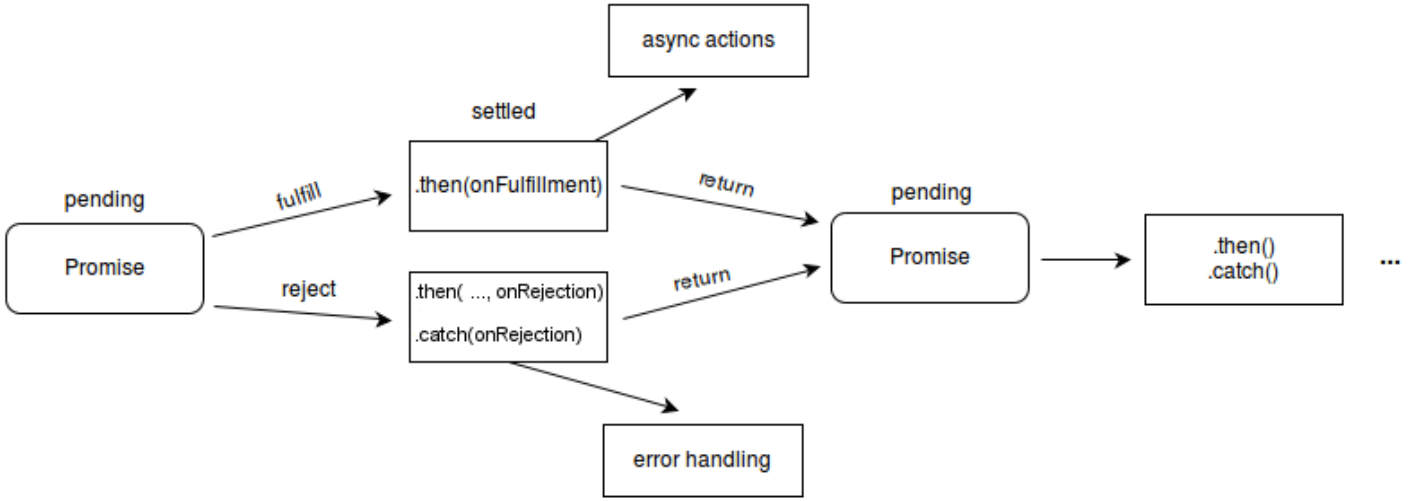
```
const promesa = crearArchivoAudioAsync(audioConfig);
promesa.then(exitoCallback, falloCallback);
```

Promesas

- Ventajas de usar promesas:
  - Las funciones de retorno nunca serán llamadas antes de la terminación de la ejecución actual
  - Se pueden encadenar llamadas de funciones de retorno

Promesas

- Una promesa está en alguno de estos estados:
  - Pendiente: estado inicial, ni cumplida ni rechazada
  - Cumplida: la operación se completó con éxito
  - Rechazada: la operación falló



Promesas

- Encadenamiento: es común la ejecución de 2 o más operaciones asíncronas seguidas
  - Cada operación posterior se inicia cuando la previa tiene éxito con el resultado del paso previo
  - Se pueden ir creando Promesas encadenadas

```
const myPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('foo');
  }, 300);
});

myPromise
  .then(handleResolvedA, handleRejectedA)
  .then(handleResolvedB, handleRejectedB)
  .then(handleResolvedC, handleRejectedC);
```

Promesas

- Encadenamiento
  - Se puede omitir el manejo de errores hasta una sentencia catch

```
myPromise
   .then(handleResolvedA)
   .then(handleResolvedB)
   .then(handleResolvedC)
   .catch(handleRejectedAny);
```

Promesas

- Ejemplo simple:

```
let myFirstPromise = new Promise((resolve, reject) => {
  // Llamamos a resolve(...) cuando lo que estábamos haciendo de forma
  // asíncrona fue exitoso, y reject(...) cuando falló.
  // En este ejemplo, usamos setTimeout(...) para simular código asíncrono.
  // En realidad, probablemente usará algo como XHR o una API HTML5.
  setTimeout( function() {
    resolve("Éxito!")  // ¡Hurra! ¡salió bien!
  }, 1000)
})

myFirstPromise.then((successMessage) => {
  // successMessage es lo que hayamos pasado en la función resolve(...) anterior.
  // No tiene que ser una cadena, pero si es solo un mensaje de éxito,
  // probablemente lo sea.
  console.log("¡Hurra! " + successMessage)
});
```

Promesas

- Ejemplo saludo con llamadas de retorno:

```
<!DOCTYPE html>
<html>
<body>
<h1>JavaScript Functions</h1>
<h2>setInterval() with a Callback</h2>

<p>Wait 3 seconds (3000 milliseconds) for this page to change.</p>

<h1 id="demo"></h1>

<script>
setTimeout(function() { myFunction("Hello World !!!"); }, 3000);

function myFunction(value) {
  document.getElementById("demo").innerHTML = value;
}
</script>

</body>
</html>
```

Promesas

- Ejemplo saludo con promesas:

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Promise</h2>

<p>Wait 3 seconds (3000 milliseconds) for this page to change.</p>
<h1 id="demo"></h1>

<script>
const myPromise = new Promise(function(myResolve, myReject) {
  setTimeout(function(){ myResolve("Hello world !!"); }, 3000);
});

myPromise.then(function(value) {
  document.getElementById("demo").innerHTML = value;
});
</script>

</body>
</html>
```

Promesas

- Llamadas asíncronas a un servidor: `fetch()`
  - Similar a XHR pero retorna Promesas
    - API más simple y limpia
  - Parámetros
    - Input: recurso que se quiere solicitar (URL)
    - Init (opcional): objeto de opciones que contiene configuraciones para la solicitud
      - Método HTTP, cabeceras, datos, credenciales, modo de caché, etcétera
    - Retorna una `Promise` que resuelve a un objeto `Response`.

Promesas: ejemplo XHR

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Promise</h2>
<p id="demo"></p>
<script>
function myDisplayer(some) {
  document.getElementById("demo").innerHTML = some;
}

let myPromise = new Promise(function(myResolve, myReject) {
  let req = new XMLHttpRequest();
  req.open('GET', "mycar.html");
  req.onload = function() {
    if (req.status == 200) {
      myResolve(req.response);
    } else {
      myReject("File not Found");
    }
  };
  req.send();
});

myPromise.then(
  function(value) {myDisplayer(value);},
  function(error) {myDisplayer(error);}
);
</script>
</body>
</html>
```

Revisar en: https://www.w3schools.com/js/tryit.asp?filename=tryjs_promise1

Promesas: fetch

- Solicitudes con XHR necesitan 2 listeners para éxito y error.

```
function reqListener() {
    var data = JSON.parse(this.responseText);
    console.log(data);
}

function reqError(err) {
    console.log('Fetch Error :-S', err);
}

var oReq = new XMLHttpRequest();
oReq.onload = reqListener;
oReq.onerror = reqError;
oReq.open('get', './api/some.json', true);
oReq.send();
```

Promesas: fetch

```
fetch('./api/some.json')
    .then(
    function(response) {
        if (response.status !== 200) {
        console.log('Looks like there was a problem. Status Code: ' +
            response.status);
        return;
        }

        // Examine the text in the response
        response.json().then(function(data) {
        console.log(data);
        });
    }
    )
    .catch(function(err) {
    console.log('Fetch Error :-S', err);
    });
```

Promesas: fetch

- La respuesta a peticiones fetch tiene un tipo asociado:
  - Basic: peticiones realizadas al mismo origen, no tiene restricciones de lo que podemos ver en la respuesta
  - Cors: restringe las cabeceras que se pueden ver:
    - Cache-Control, Content-Language, Content-Type, Expires, Last-Modified, Pragma
  - Opaque: peticiones a un recurso en distinto origen y no tiene cabecera CORS.
    - No se puede acceder a los datos retornados ni el estatus de la solicitud

Promesas: fetch

- Se pueden definir peticiones con ciertos modos:
  - Same-origin: exitoso solo para peticiones al mismo origen, resto fallará
  - Cors: exitoso para peticiones al mismo origen y a otros que tengan la cabecera CORs
  - Cors-with-forced-preflight: siempre realizará un petición verificada previamente
  - No-cors: peticiones a otro origen que no tiene cabecera CORs y resulta en respuesta “opaque”