Javascript S8 | Manejando la asincronía | Peticiones

Después de esta lección podrás:

1. Manejo de la asincronía en Js.
2. Diferenciar entre síncrono y asíncrono.
3. Trabajar con Callbacks.
4. Entender y realizar Promesas.
5. Uso de Fetch → API

La evolución de Javascript nos permite manejar llamadas asíncroncas y manejarla a tavés de algunas funciones. Normalmente se suele utilizar en operaciones de entrada o salida de datos como escritura de un JSON o la lectura de disco pero donde realmente se ve su uso son con las peticones AJAX.

Vamos a ver cómo se han gestionado hasta 2015 y con la aparición de ES6 el estándar actual, eso no quita que os encontréis situaciones o código que usen la forma más tradicional. Veamos cuales son.

Callbacks

Es la primera y la forma más común de controlar la asincronía en JavaScript hasta 2015, y como siempre la mejor forma de comprender algo es con un ejemplo.

```
const callbackExample = (list, message) => {
  console.log(`El listado final es: ${list.join(', ')} -`, message);
}

const addItemAndTriggerCallback = (item, list, callback) => {
  if (!list) {
    setTimeout(() => {
      throw new Error('No existe el array');
    }, 2000);
  } else {
    setTimeout(() => {
      list.push(item);
      callback(list, '[OK]');
    }, 2000);
  }
}
```

Vamos a ver qué ha pasado en el ejemplo anterior. Nuestra **función que recibe como parámetros** un **item de entrada**: `item`, un **array con datos** `list` y una **función de callback**: `callbackExample`.

Con estos tres parámetros hacemos lo siguiente, al listado `list` **se le añade** `item` que **viene por parámetro** y cuando **termine**, **llama a la función de** `callback` que recibe por parámetro, en ese caso la llama con la `list` modificada.

BONUS: Hemos añadido un pequeño bloque para comprobar si la lista existe y si no lanzar un error.

Ahora vamos a ejecutar nuestra función para ver cómo tratar el ***callback*** y veamos qué está pasando:

```
addItemAndTriggerCallback('Mario', ['Alberto', 'Jose'], callbackExample);
// addItemAndTriggerCallback('Mario', null, callbackExample);
```

¿Qué ha pasado? Cuando se ha termiando de ejecutar `addItemAndTriggerCallback` se ejecuta el `callback` y nuestro array `list` tiene un nuevo dato.

Frente a eso me diréis que podemos añadir el dato al array y después hacer un ***console.log***, pero qué hubiese sucedido si queremos añadir un dato a un array que aún no tenemos? en esto consiste la asíncrona, como son las peticiones vía AJAX.

¿En el siguiente fragmento, cómo haríamos para garantizar que el listado se imprime con todos los elementos? Pues por esta razón, necesitamos los callbacks:

```
const list = ['Jose', 'Alberto'];

setTimeout(() => list.push('Mario'), 2000);
list.push('Pedro');

console.log(list); // ['Jose', 'Alberto', 'Pedro'] 🤔 y Mario?
```

> Cuando imprimimos el array aún no se ha añadido el nuevo item, por lo tanto el comportamiento que sucede no es el buscado. De esta forma los callbacks nos ayudan a que esto no suceda. En resumen nos ayuda a manejar la asincronía.

Pero cuidado con los `callbacks`, porque si empezamos a enlazarlos unos con otros... puede ocurrir lo siguiente:

```
let list = ['Raising Arizona','Fargo','Barton Fink'];

addItemAndTriggerCallback('The Big Lewoski', list, function (err) {
  if (err) ...
  addItemAndTriggerCallback('O Brother, Where Art Thou?', list, function (err) {
    if (err) ...
    addItemAndTriggerCallback('The Man Who Wasnt There', list, function (err) {
      if (err) ...
      addItemAndTriggerCallback('The Ladykillers', list, function () {
        // Y podemos seguir...
      })
    })
  })
});
```

A esto se le conoce como **Callback Hell** o **Pyramid of Doom.**

Promesas

Una `Promise` (promesa en castellano) es un objeto que representa la terminación o el fallo de una operación asíncrona. Surgen en ES6 para mejorar el proceso de callbacks.

Por lo general en nuestro proyectos NO vamos a **crear** promesas, vamos a consumirlas (`then`/`catch`), pero veamos la base de las promesas:

```
const addItem = (item, list) => {
  const promise = new Promise((resolve, reject) => {
    if (!list) {
      reject('No existe el array');
    }

    setTimeout(function () {
      list.push(item);
      resolve(list);
    }, 2000);
  });

  return promise;
};

const list = ['Rojo', 'Azul', 'Verde'];

addItem('Amarillo', list)
  .then((list) => {
    console.log(`El listado final es: ${list.join(', ')}`);
  })
  .catch((err) => {
    throw new Error(err);
  });
```

Ahora la función `addItem` crea un objeto `Promise` que recibe como parámetros una función con las funciones `resolve` y `reject`. Llamaremos a `resolve` cuando nuestra ejecución finalice correctamente, y a `reject` para indicar que ha habido un rechazo (error) en la ejecución.

De esta manera, podemos escribir código de manera más elegante, y el *Callback Hell* anterior puede ser resuelto así:

```
const list = ['Raising Arizona','Fargo','Barton Fink'];

addItem('The big Lewoski', list)
  .then(() => addItem('O Brother, Where Art Thou?', list))
```

```
  .then(() => addItem('The Man Who Wasnt There', list))
  .then(() => addItem('The Ladykillers', list))
  .then(() => {
    console.log(list);
  });

// (4 seg. de delay) -> ['Raising Arizona','Fargo','Barton Fink', ...];
```

Esto es conocido como ***anidación promesas***.

La forma de **tratar errores en una promesa**, es por medio de la **función** `catch` que **recoge** lo que **enviamos** en la **función** `reject` dentro de la Promesa. Y esta función solo hay que invocarla una vez, no necesitamos comprobar en cada llamada si existe error o no. Lo cual reduce mucho la cantidad de código:

```
const filmography = '';
addToCoenBrothers('The big Lewoski', filmography)
  .then(...)
  .then(...)
  .then(...)
  .catch(err => console.log(err.message));

// No existe el array -> es un string - salta error
```

¿Repasamos?

Lo más habitual es que **consumamos** promesas ya creadas, pero vamos a repasar cómo se crearía una promesa.

```
// ES5
let promise = new Promise(function (resolve, reject) {
  // El ejecutor se ejecuta automáticamente cuando se// construye la promesaconsole.log("EXECUTED EXECUTER");

  // Pasado 1 seg estamos resolviendo la promesa con el // valor "done"
  setTimeout(function() {
    resolve("done")
  }, 1000);
});

// ES6
let promise = new Promise((resolve, reject) => {
  // El ejecutor se ejecuta automáticamente cuando se// construye la promesaconsole.log("EXECUTED EXECUTER");

  // Pasado 1 seg estamos resolviendo la promesa con el // valor "done"
  setTimeout(() => resolve("done"), 1000);
});
```

Hemos visto cómo crear una promesa, pero ya decíamos lo más habitual es consumirlas:

```
promise.then(
  (result) => {
    // Manejamos el resultado
    console.log(result);
  },
  (err) => {
    // Manejamos el reject concreto
    console.error(err);
  }
);

// Error general
promise.catch(
  (err) => {
    // Manejamos el error
    console.error(err);
  }
);
```

Si nos fijamos, `then(function, function)` puede recibir dos parámetros: 2 funciones o callbacks.

- La primera functión se ejecutará en caso de éxito y nos permitirá manejar la respuesta en caso de OK.
- La segunda función se ejecutará en caso de error y nos permitirá manejar el KO.

Es decir, podemos controlar también el fallo de una promesa concreta, dependiendo de la necesidad usaremos el control específico o el `catch`.

Try / Catch

Para terminar vamos a introducir una mínima pauta en el manejo de errores, ya que por ahora no nos hemos preocupado de posibles fallos en la ejecución de Javascript.

Siempre que queramos **controlar** un error, basta con que envolvamos nuestro código en un `try` / `catch`

```
// error NO controlado
const error = null.nombre;

try {
  // error controlado
  const error = null.nombre;
} catch (error) {
  console.error('error', error);
}
```

Async/Await

En vista de la complejidad de trabajar con promesas, los nuevos estándares de javascript incluyeron un par de sentencias especiales en el set del lenguaje: `async` / `await`

Nos permitirán trabajar con las promesas de una manera más cómoda, sin tener que declarar los `then` / `catch`.

Digamos que "marcaremos" aquellos fragmento de código dónde tenemos que "esperar" a que la promesa se resuelva (para que nuestra ejecución NO siga tirando para adelante como loca 🥴).

Es sorprendentemente fácil de entender y usar. La sintaxis para una función que utilice ***async/await*** es la siguiente:

```
const myFunction = async () => {
  var result = await functionAsincrona();
};
```

La función irá precedida por la palabra reservada `async`. Dentro del `try` llamaremos a la función asíncrona con la palabra reservada `await` delante, con esto hacemos que la función *espere* a que se ejecute y el resultado de la misma está disponible en este caso en la variable `result`.

Combinando `async` / `await` con una función basada en `Promesas`, podemos hacer lo siguiente con el ejemplo que estábamos viendo:

```
const addItem = (item, list) => {
  const promise = new Promise((resolve, reject) => {
    if (!list) {
      reject('No existe el array');
    }

    setTimeout(function () {
      list.push(item);
      resolve(list);
    }, 2000);
  })

  return promise;
};

const processFilm = async (film, filmography) => {
  try {
    const result = await addItem(film, filmography);
    console.log(result);
  } catch (error) {
    console.error('error', error);
  }
}

const filmography = ['Raising Arizona','Fargo','Barton Fink'];
processFilm('The big Lewoski', filmography);
processFilm('O Brother, Where Art Thou?', filmography);
processFilm('The Ladykillers', filmography);
```

De esta manera estamos escribiendo código de manera secuencial pero JavaScript está (por debajo) ejecutando código asincrónico.

Promise all

Por último ¿qué pasa si tengo que resolver 2 promesas "a la vez"? Esto ocurre cuando tengo que consultar 2 orígenes de datos asíncronos.

Veremos como sería resolver varias promesas gracias a `Promise.all`. Encadenaremos varias promesas y haremos que se resuelvan todas a la vez, cuando estén resultas nos devolverá nuestra "agrupadora de promesas" resuelta. Veamos un ejemplo:

```
const theBigLewoski = Promise.resolve('The big Lewoski');
const trueGrit = new Promise((resolve, reject) => {
  setTimeout(resolve, 2000, 'True Grit');
});
```

```
Promise.all([theBigLewoski, trueGrit]).then(films => {
  console.log(films);
  // ['The big Lewoski', 'True Grit']
});
```

Promise.all espera a que todo se cumpla (o bien al primer rechazo). Rechaza si uno de los elementos ha sido rechazado y ***Promise.all*** falla rápido: Si tienes cuatro promesas que se resuelven después de un timeout y una de ellas falla inmediatamente, entonces ***Promise.all*** se rechaza inmediatamente.

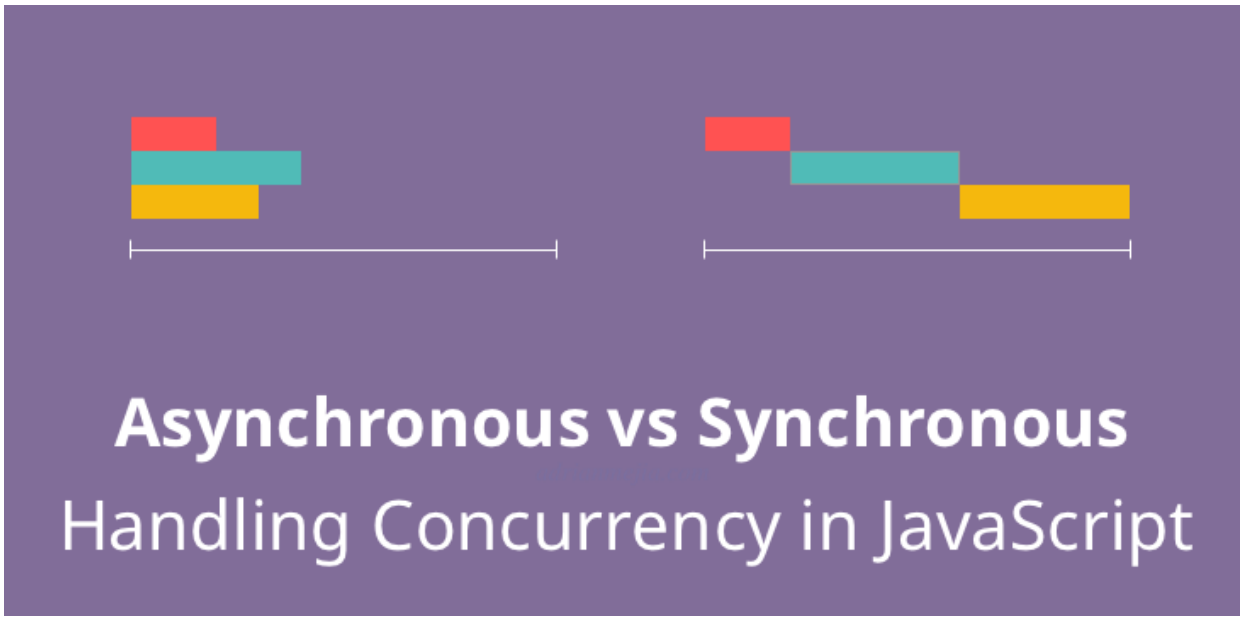
Veamos un ejemplo de ***Promise.all*** si alguna de las promesas es rechazada:

```
const theBigLewoski = new Promise((resolve, reject) => {
  setTimeout(resolve, 1000, 'The big Lewoski');
});
const theLadyKillers = new Promise((resolve, reject) => {
  setTimeout(resolve, 2000, 'The Lady Killers');
});
const joJoRabbit = new Promise((resolve, reject) => {
  reject('jo jo rabbit - error Taika Waititi');
});

/* Example: .then with 2 functions */
Promise.all([theBigLewoski, theLadyKillers, joJoRabbit]).then(
  (films) => {
    console.log(films);
  }, (err) => {
    console.error(err);
});

/* Example: .catch */
// Promise.all([theBigLewoski, theLadyKillers, joJoRabbit])
//   .then((films) => {
//     console.log(films);
//   }).catch((err) => {
//     console.error(err);
//   });
```

¿Javascript es síncrono o asíncrono?

El motor de JavaScript siempre es síncrono y todas las peticiones que consideremos asíncronas se hacen a través del **pool de Thread de WebAPI** que es algo externo.

Javascript por defecto corre en un único *hilo*, es decir, las instrucciones se van ejecutando una detrás de otra y una instrucción no puede ejecutarse hasta que la anterior ha terminado.

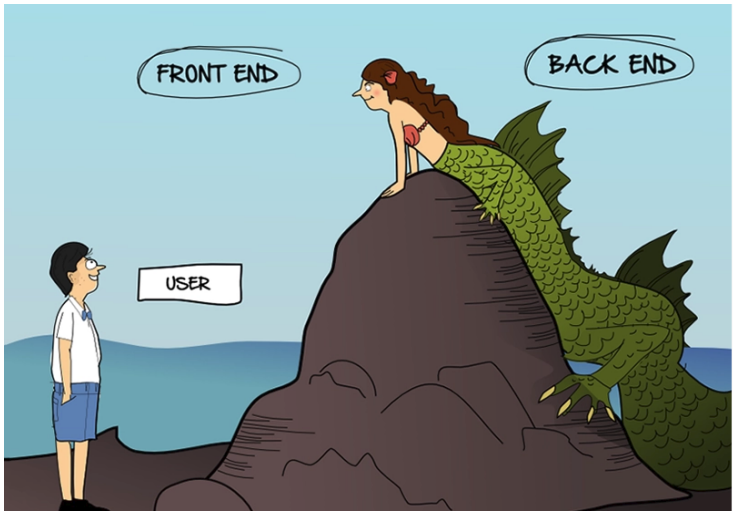


Un código asíncrono no es más que un código que empieza *ahora* y es capaz de abrir *hilos* de ejecución en paralelo de tal forma que hay varias partes del código ejecutándose a la vez. Es posible (o no) que pasado el tiempo vuelvan a *juntarse*.

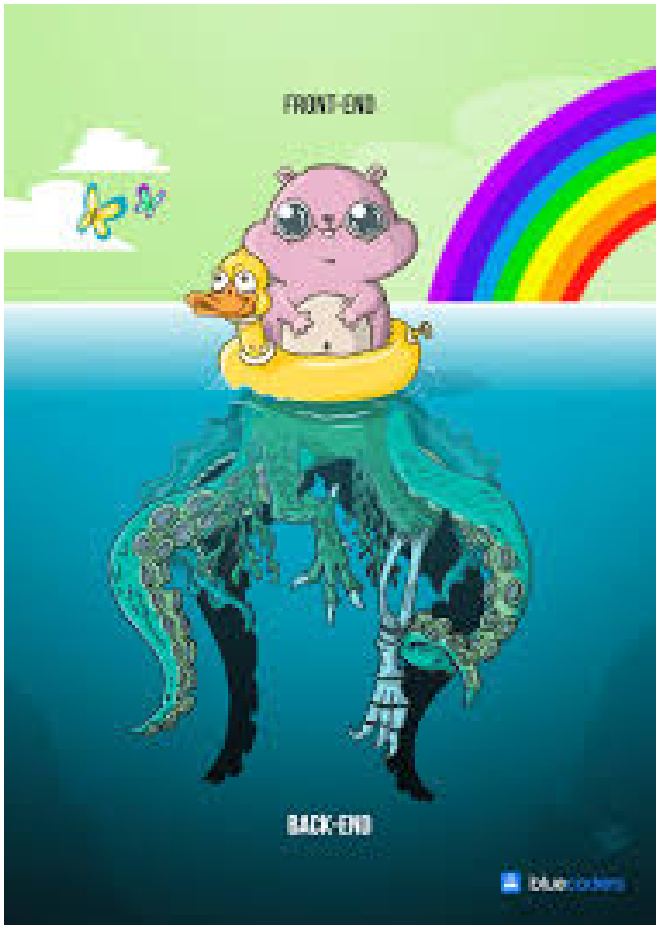
Peticiones a una API

Por fin ha llegado el momento que todos esperábamos, no es otro que aprender a utilizar **AJAX** que es un **enlace entre cliente**, por ejemplo Chrome, **y el servidor**. Esto nos ayudará a enlazar el frontend y el backend de nuestra aplicación. Las **peticiones AJAX** nos **permiten acceder y manipular datos** en el servidor **desde el frontend**.

Antes de nada estaría bien que entendamos la **diferencia** entre **frontend y backend**, por aquello de situar dónde está el API y entender por qué es necesaria.

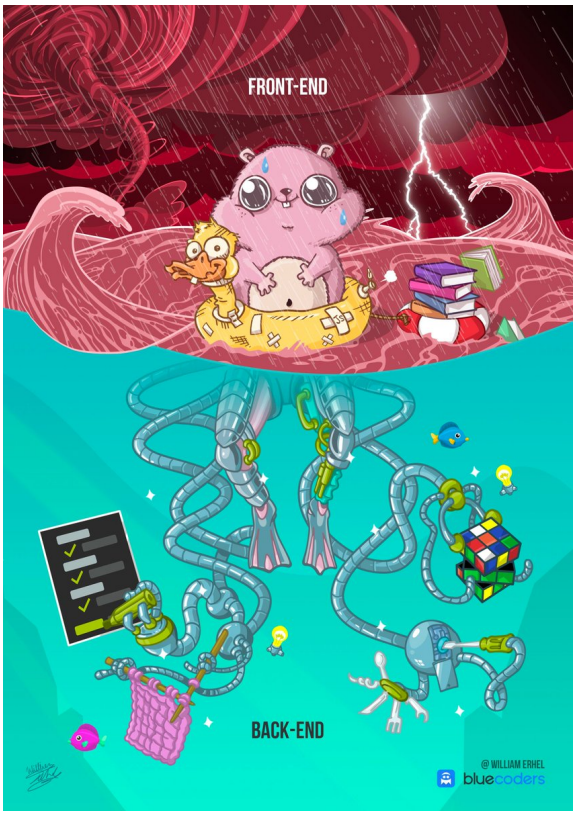


`Frontend` es la parte de nuestra aplicación a la que un usuario puede acceder. Esto hace referencia a todas las tecnologías de desarrollo web o lenguje de marcado que corren en el navegador o cliente. Los principales lenguajes son HTML, CSS y Javascript, cabe destacar que a partir de aquí se abre un gran abanico de frameworks y librerías que nos ayudan a realizar nuestras interfaces. Angular, React, PostCss, SASS son algunos de ellos.



Desde el punto de vista de un Backend, el Front es todo luz y color... y lo complejo está en las tripas de la aplicación.

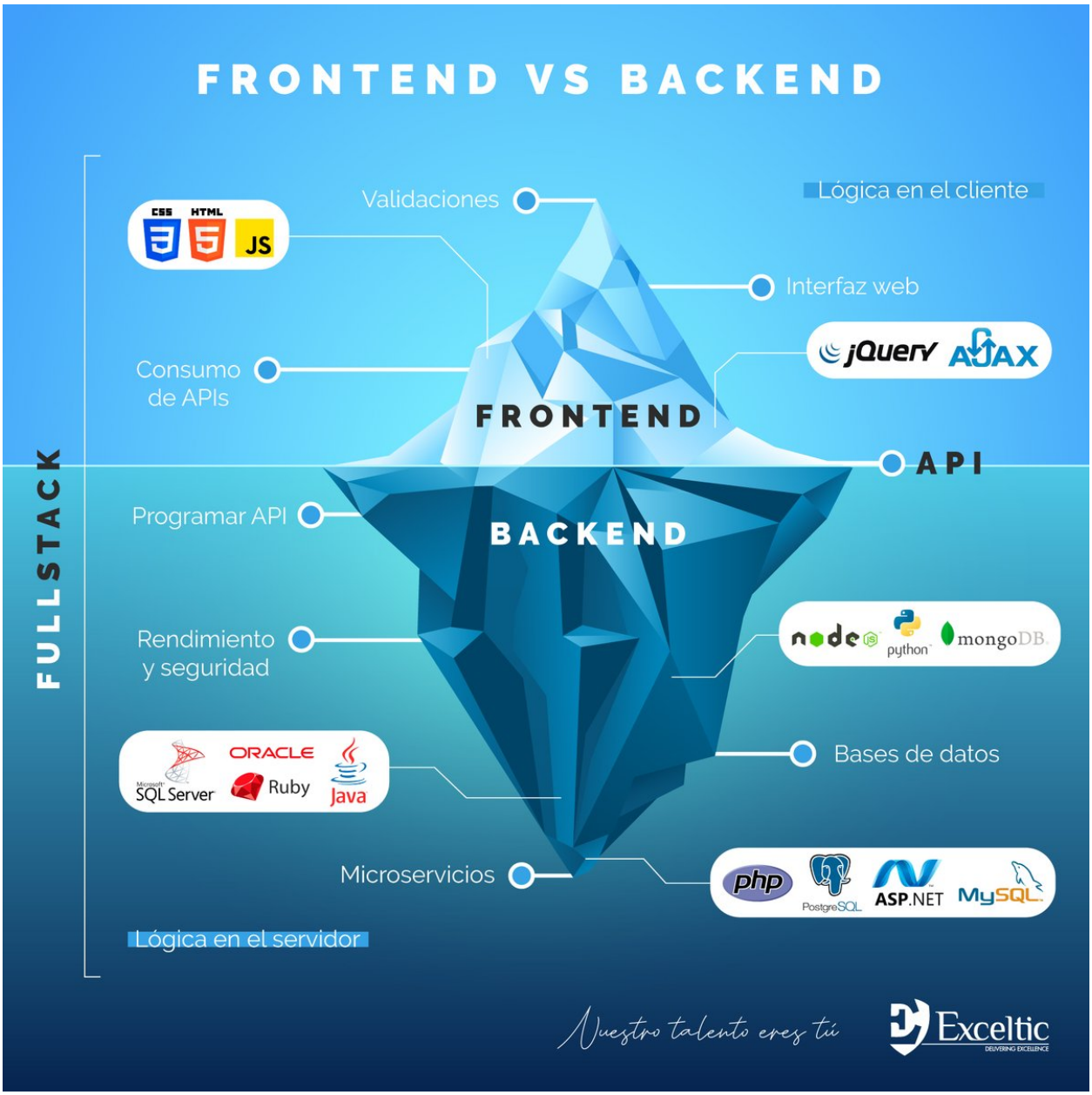
`Backend` dentro de nuestra aplicación es lo que se conoce como servidor, una zona donde el usuario no tiene acceso directo. Desde el backend enviaremos la información solicitada, utilizando Javascript en el backend encontramos Node.js y en caso de utilizar el tipado con typescript Deno.js.



Desde el punto de vista de un Frontend, el Back es un entramado de cables y flujos... y lo complejo está en la capa visual.

En resumen, la comunicación entre Backend y Frontend se realizará vía API, mediante contratos de intercambio. Digamos que Back y Front acuerdan cómo van a comunicarse, mediante qué métodos (endpoints) y que objectos (JSON) se tienen que intercambiar. Para que la información de la base de datos llegue hasta el frontal y viceversa.

Aquí os dejamos un resumen de ambos mundos:



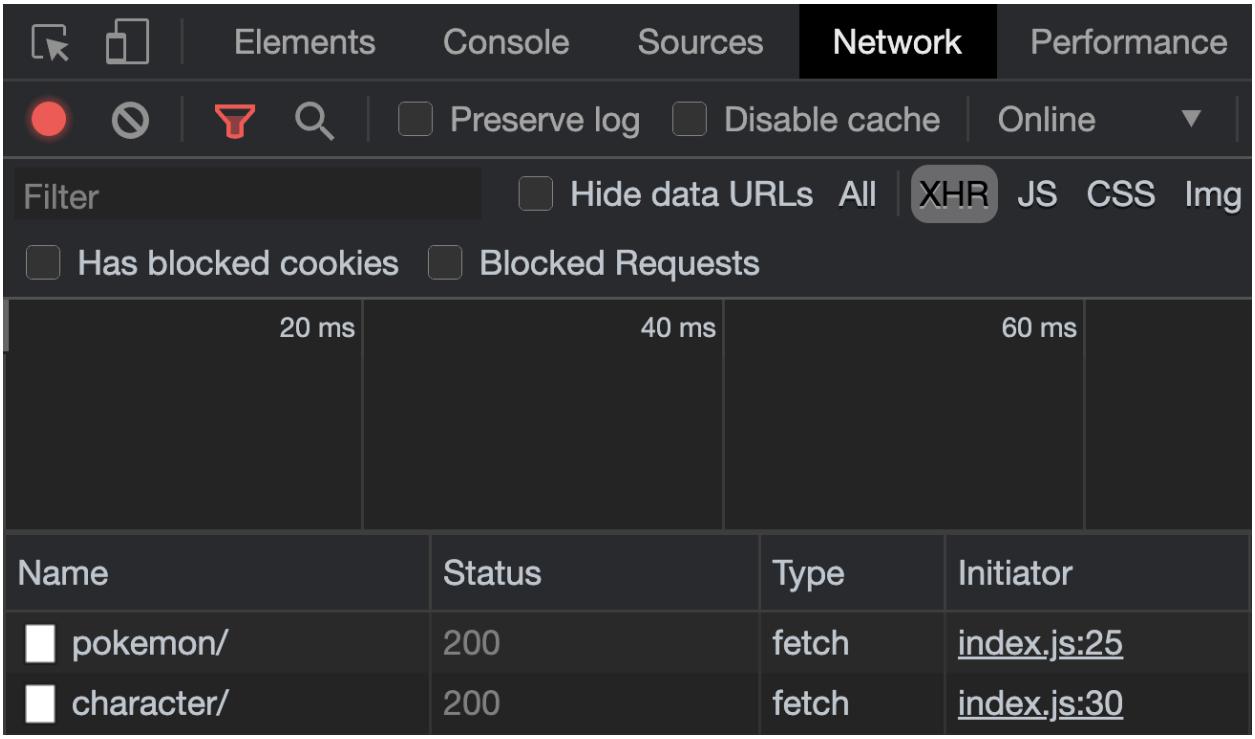
Peticiones XHR

En su momento, teníamos que valernos de librerías externas para atacar API's y hacer llamadas a endpoints. Una de las librerías más usadas en el front fue Axios.

Con la salida de ES6 apareció un nuevo API de JS, `Fetch`. Nos permitirá hacer peticiones AJAX de una manera más cómoda haciendo uso de promesas.

```
fetch('https://pokeapi.co/api/v2/pokemon/')
  .then((response) => {
    return response.json();
  })
  .then((myJson) => {
    console.log(myJson);
  });

fetch('https://rickandmortyapi.com/api/character/')
  .then((response) => {
    return response.json();
  })
  .then((myJson) => {
    console.log(myJson);
  });
```

Con ello, vamos a empezar a usar la pestaña Network de las dev-tools de Chrome:



Aquí encontraremos estas llamadas a la API, esta interacción con el servidor, para por fin poder comunicarnos con las bases de datos y culminar nuestras web apps.

¿Por qué en los fetch tenemos que usar dos `.then`?

Porque el método `.json` de `response` es asíncrono y devuelve también una promesa, así que al hacer `return response.json()` el primer `.then` devuelve una promesa y necesitamos encadenar otro `.then` para resolverla.

Ejemplos de API pública

- https://pokeapi.co/docs/v2
- https://rickandmortyapi.com/documentation
- https://www.potterapi.com/
- https://swapi.dev/