

Boilerplate

Los ejemplos que vamos a codificar los puedes probar en la sandbox de JavaScript, si no sabes cómo funciona, échale un vistazo al módulo de setup y si te quedan dudas contacta con tu mentor.

Introducción

Vamos a leer una lista de películas de un servidor remoto y las vamos a mostrar en pantalla.

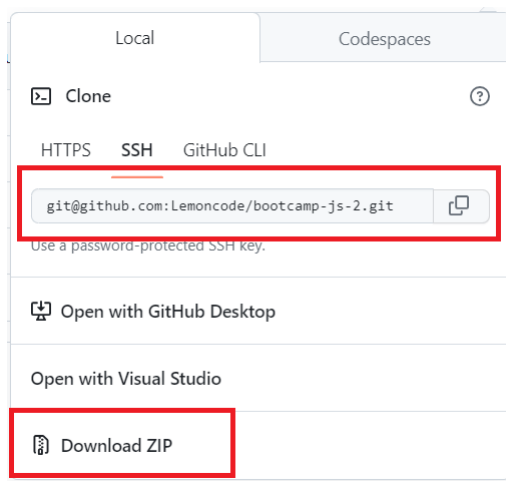
Fuente remota

Hemos creado un repositorio en GitHub con una API Rest que nos va a servir para hacer las pruebas. Lo tenemos en el siguiente [dirección](#), donde tenemos diferentes materiales que hemos ido utilizando a lo largo del bootcamp. Dentro de la carpeta *10-async/03-server-peliculas* tenemos el código de la API Rest que vamos a utilizar.

En esta dirección te puedes descargar un proyecto en el que creamos una API Rest local con dos endpoints:

- Una lista de películas.
- Una lista de actores.

Para utilizarlo tienes que clonar el repositorio de *bootcamp-js-2* en tu equipo o descargarlo en tu equipo.



Una vez clonado el repositorio de *bootcamp-js-2* en tu equipo, el siguiente paso que vamos a hacer es instalar las dependencias del proyecto, para esto tienes que ir a la carpeta *10-asincronia/03-server-peliculas* y ejecutar el siguiente comando:

En el caso de que estés en el raíz de la ruta, tendrías abrir el terminal y moverte a la ruta donde se encuentra el server

```
cd 10-async
```

```
cd 03-server-peliculas
```

Nota: se recomienda copiar el contenido de la carpeta *03-server-peliculas* en otra carpeta para no mezclarlo con el resto de material del bootcamp.

```
npm install
```

Una vez instaladas las dependencias, para arrancar el servidor tienes que ejecutar el siguiente comando:

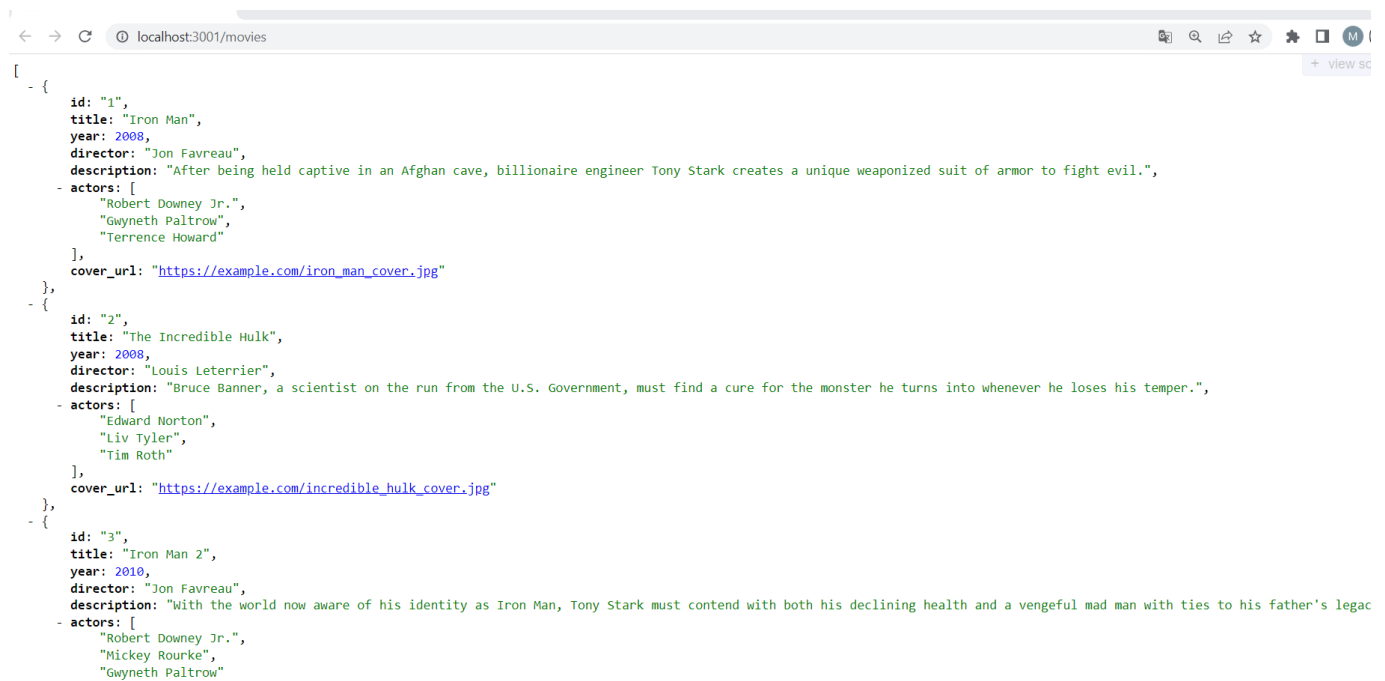
```
npm start
```

Una vez arrancado el servidor, puedes acceder a los siguientes endpoints:

Abrimos nuestro navegador web favorito y accedemos a la siguiente dirección:

- <http://localhost:3000/movies>

Y veremos que nos muestra un listado de películas.



Si accedemos a la siguiente dirección:

- <http://localhost:3000/actors>

Veremos que nos muestra un listado de actores.

```
[
  - {
    id: "1",
    name: "Robert Downey Jr.",
    - movies: [
      "1",
      "3",
      "6",
      "7",
      "22"
    ],
    bio: "Robert Downey Jr. is an American actor. He is best known for his role as Tony Stark/Iron Man in the Marvel Cinematic Universe.",
    image: "robert_downey_jr.jpg"
  },
  - {
    id: "2",
    name: "Gwyneth Paltrow",
    - movies: [
      "1",
      "3",
      "7"
    ],
    bio: "Gwyneth Paltrow is an American actress. She portrayed the character Pepper Potts in the Marvel Cinematic Universe.",
    image: "gwyneth_paltrow.jpg"
  },
  - {
    id: "3",
    name: "Terrence Howard",
    - movies: [
      "1"
    ],
    bio: "Terrence Howard is an American actor. He played the role of James Rhodes in the movie Iron Man.",
    image: "terrence_howard.jpg"
  },
  - {
    id: "4",
    name: "Edward Norton",
    movies: [

```

¿Qué es una petición HTTP?

Pero nos estaremos preguntando, ¿qué es una petición *HTTP*? *HTTP* es un protocolo de comunicación que permite transferir información entre cliente y servidor desde nuestro navegador web (entre otros).

¿Cómo funciona? Muy sencillo, el cliente pide unos datos, el servidor responde con esos datos y se cierra la conexión (de ahí que se diga que es un protocolo de petición-respuesta).

Para realizar una petición *HTTP* necesitamos saber:

- La dirección del servidor.
- El puerto del servidor (verás qué si usamos el puerto 80, no hará falta ponerlo ya que lo toma como por defecto).
- El método *HTTP* que vamos a utilizar (en el método le vamos a decir, oye que quiero leer datos, sería un GET, quiero insertar nuevos datos o modificarlos, eso sería otro método POST, PUT... iremos viéndolo en detalle).

En el caso en concreto de la API Rest que hemos creado, la dirección del servidor es *localhost*, el puerto es *3000* y el método *HTTP* que vamos a utilizar es *GET*.

Al usar el método *GET* estamos indicando que queremos obtener información del servidor.

Oye pues cuando has pedido los datos en el navegador, no has tenido que indicar que es un *GET* Correcto, esto es porque cuando lo pongo en el navegador, directamente estoy haciendo una petición *HTTP* con el método *GET*, ya veremos otras herramientas con la que podemos simular otros verbos.

¿Pasa lo mismo con los puertos? En este caso fíjate que sí indicamos el puerto (el 3000), pero si no lo indicamos, el navegador utiliza el puerto 80, que es el puerto por defecto para las peticiones *HTTP*, por ejemplo, si hacemos esta petición

```
https://rickandmortyapi.com/api/character
```

No hace falta indicar puerto, ya que esta API trabaja con el puerto por defecto 80.

Métodos HTTP

No sólo de GET vive el desarrollador, existen otros métodos *HTTP* que podemos utilizar para realizar peticiones al servidor.

Los más utilizados:

- GET: Se utiliza para obtener información del servidor.
- POST: Se utiliza para crear información en el servidor (insertar una nueva entrada).
- PUT: Se utiliza para actualizar información en el servidor (reemplazar por completo una entrada).
- PATCH: Se utiliza para actualizar información en el servidor, pero solo una parte de la información (actualizar sólo ciertos campos de una entrada).
- DELETE: Se utiliza para eliminar información en el servidor (eliminar una entrada).

Códigos de respuesta HTTP

Cuando hacemos una petición *HTTP* al servidor, este nos devuelve un código de respuesta, estos códigos de respuesta nos indican si la petición se ha realizado correctamente o no.

Lo más comunes son:

- 200: La petición se ha realizado correctamente.
- 201: La petición se ha realizado correctamente y se ha creado un nuevo recurso.
- 400: La petición no se ha realizado correctamente.
- 401: La petición no se ha realizado correctamente porque no tenemos permisos.
- 404: La petición no se ha realizado correctamente porque el recurso no existe.
- 500: La petición no se ha realizado correctamente porque ha habido un error en el servidor.

¿Por qué no me da información más detallada acerca de un error? Esto lo podemos gestionar a nivel de servidor, pero cuando estamos en producción no queremos dar esa información detallada, ya que le estamos regalando información a un hacker para que pueda atacar nuestra aplicación.

Axios

Para arrancar vamos a usar una librería llamada *axios*, esta nos ayudará a tener un primer contacto con las peticiones *HTTP*.

Más adelante veremos cómo hacer peticiones *HTTP* utilizando el método *fetch*, que es el estándar que se incorporó en ES6 como parte de API para poder interactuar con fuentes remotas de datos.

Nota importante: *axios* lo vamos a instalar en el sandbox de JavaScript, es decir, por un lado, tendremos nuestro servidor de películas arrancado y por otro lado las peticiones *HTTP* las vamos a hacer desde el sandbox de JavaScript.

Cómo instalar Axios

Para instalar *axios* tenemos que ejecutar el siguiente comando:

```
npm install axios --save
```

Cómo usar Axios

Para usar *axios* tenemos que importarlo en nuestro proyecto, para esto tenemos que añadir la siguiente línea de código en nuestro fichero *main.js*:

main.js

```
import Axios from "axios";
```

Una vez importado *axios* en nuestro proyecto, podemos hacer una petición *HTTP* al servidor, para esto tenemos podríamos intentar lo siguiente (OJO QUE ESTO ESTÁ MAL):

main.js

```
import Axios from "axios";
import "./style.css";

- console.log("Hello from main");

+ const peliculas = Axios.get("http://localhost:3000/movies");
+ console.log(peliculas);
```

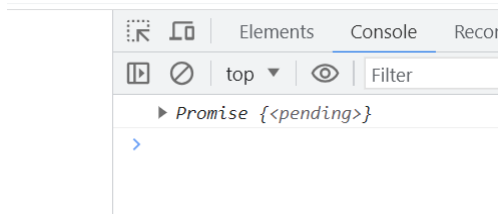
¿Qué estamos haciendo aquí?

- Importamos *axios* en nuestro proyecto.
- Hacemos una petición *HTTP* al servidor para traernos el listado de películas.

Si arrancamos nuestro Sandbox de JavaScript.

```
npm run dev
```

Y mostramos la consola del navegador, veremos que nos muestra un objeto, pero no nos muestra el listado de películas que esperábamos.



¿Qué está pasando aquí? Te acuerdas lo que vimos del event loop y que JavaScript en principio es monohebra, pues tenemos que:

- La petición al servidor va a tardar, y no podemos dar respuesta inmediata.
- Lo que nos devuelve ese *get* es una promesa ¿Lo cual? Vamos a verlo en detalle.

PROMESAS

Para explicar lo que es una promesa, vamos a pensar en un restaurante de esos "modernos", tú vas a la barra, haces un pedido (por ejemplo, una hamburguesa) y en seguida, en vez de un plato te dan un aparatito que pita cuando la hamburguesa esté lista para que vayamos a la barra a por ella, es decir:

- Ese *aparatito* es una promesa.
- Cuando pita, quiere decir que tiene los datos que le hemos pedido disponibles para que los puedas consumir.

De esta manera no saturamos la barra, ¿Te imaginas que te quedarás bloqueando la barra hasta que te dieran la hamburguesa? Pues lo mismo pasa con las peticiones asíncronas y el sistema monohebra de JavaScript.

En el caso de *axios* pasa lo mismo, cuando hacemos una petición *HTTP* al servidor, esta petición va a tardar un poco, y *axios* nos devuelve una promesa, y nosotros podemos seguir con el flujo de ejecución, y la web api del navegador encolará un mensaje cuando la promesa se haya completado (es decir, hemos recibido la respuesta del servidor), y el event loop lo recogerá y lo ejecutará.

¿Cómo funciona una promesa?

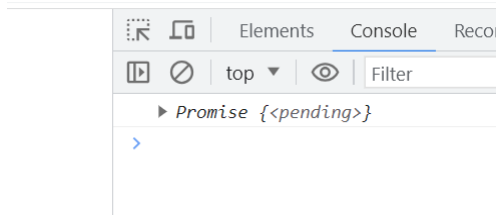
Una promesa tiene tres estados:

- **Pending:** Es el estado inicial, es cuando se crea la promesa, es decir, oye me tengo que traer algo que no te puedo devolver de forma inmediata, mientras esa petición esté en curso, te digo que estoy *pending* (algo así como "estamos trabajando en ello").
- **Fulfilled:** Es cuando la promesa se ha cumplido, es decir, todo ha ido bien, ya tengo los datos, y puedes consumirlos.
- **Rejected:** Es cuando la promesa no se ha cumplido, es decir, menudo castañazo que ha pegado esto (sea fallo en servidor o cliente), lo siento, no he sido capaz de traerte los datos.

Si quieres otro ejemplo (y este es de los que duelen), también puedes pensar en un pagaré que te da un cliente...

- Te dice oye yo para tal fecha te pagaré (pending).
- Cuando llega la fecha:
 - Si paga, el pagaré se ha cumplido (fulfilled).
 - Si no pagas, el pagaré se ha incumplido (rejected).

De momento en la consola del navegador nos muestra el estado *pending*, es decir, la promesa se ha creado, pero todavía no se ha cumplido.



Vamos a volver a machacar este concepto con el ejemplo del restaurante, tenemos que:

- Pending: Es cuando el camarero nos dice que ya ha tomado nota, que nos asigna un *aparatito* y que estemos atentos a cuando este pite.
- Fulfilled: Es cuando el *aparatito* empieza a pitar porque ya está la hamburguesa ya está lista, y tienes que ir a recogerla a la barra.
- Rejected: El perro del camarero se comió tu hamburguesa, o estaba mirando el móvil y se le ha chamuscado, o ha echado a arder la cocina, mejor que te enteres de que ha pasado para o bien pedir otra hamburguesa (nueva promesa), o salir corriendo del local.

¿Cómo podemos saber si la promesa se ha cumplido o no?

Para esto tenemos que utilizar el método *then* de la promesa.

main.js

```
import Axios from "axios";

- const peliculas = Axios.get("http://localhost:3000/movies");
+ const peliculasPromesa = Axios.get("http://localhost:3000/movies");

+ peliculasPromesa.then((response) => {
+   console.log(response.data);
+ });

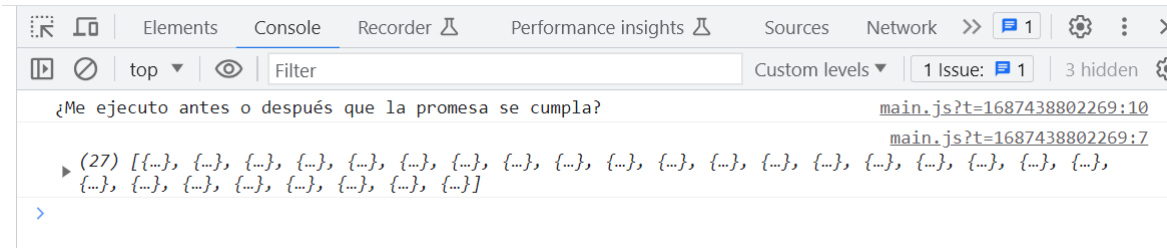
+ console.log('¿Me ejecuto antes o después que la promesa se cumpla?');
```

Aquí le estamos diciendo que cuando la promesa se cumpla, que ejecute la función que le estamos pasando como parámetro y que nos muestre por consola la *data* (los datos) de la respuesta del servidor.

¿En que orden se ejecuta esto? Para ratificar como funciona el event loop, y estas llamadas asíncronas, añadimos un *console.log* justo después de la llamada asincrónica.

Si ahora abrimos la consola del navegador, veremos que nos muestra la información que nos devuelve el servidor.

Y fíjate en el orden de los *console.log*.



Recapitulando:

- Hacemos una petición al servidor.
- Le decimos que cuando esta se resuelva ejecute una función (la que va a tratar los datos recibidos)
- Mientras, podemos seguir con nuestro hilo de ejecución, terminar de ejecutar la función en curso y cederle el testigo al event loop.

Fíjate que en realidad cuando hacemos llamadas a servidor lo que es ejecución es muy poco, la mayoría del tiempo estás en espera de que el servidor te devuelva los datos, por eso podemos utilizar concurrencia en un sólo hilo, esto es lo que llamamos operaciones de entrada/salida (E/S o I/O en inglés).

¿Qué respuesta nos devuelve el servidor?

Empecemos por entender el ejemplo que acabamos de montar *¿De dónde viene eso de data?* Esto es un detalle de implementación de *axios* cuando nos da una respuesta recibimos información de cómo ha ido la cosa y además, los datos en sí que queremos explotar, estos datos se almacenan en la propiedad *data*.

Es decir, el *happy path* (o camino de baldosas amarillas del mago de oz, o el "todo va a ir bien"), es directamente suponer que no van a haber errores y podemos leer la información directamente de *data*.

En este caso, nos devuelve un array con la lista de películas.

Vamos a explorar el resto de información que nos devuelve una llamada *get* de *axios*, para ello vamos a sacar por consola todo lo que es el *response*.

main.js

```
import Axios from "axios";
import "./style.css";

const peliculas = Axios.get("http://localhost:3000/movies");

peliculas.then((response) => {
  - console.log(response.data);
  + console.log(response);
});
```

Abrimos la consola del navegador y veremos que nos muestra la respuesta del servidor.


```
main.js?t=1687425851382:5
▼ {data: Array(27), status: 200, statusText: 'OK', headers: AxiosHeaders, config: {...}, ...} ⓘ
  ► config: {transitional: {...}, adapter: Array(2), transformRequest: Array(1), transformResponse: Array(1),
  ► data: (27) [...], [...], [...], [...], [...], [...], [...], [...], [...], [...], [...], [...], [...], [...], [...], [...], [...], [...], {
  ► headers: AxiosHeaders {cache-control: 'no-cache', content-type: 'application/json; charset=utf-8', expi
  ► request: XMLHttpRequest {onreadystatechange: null, readyState: 4, timeout: 0, withCredentials: false, u
    status: 200
    statusText: "OK"
  ► [[Prototype]]: Object
```

¿Qué nos está devolviendo el servidor?

- *config*: Este parámetro contiene la configuración utilizada en la solicitud *HTTP* que hemos realizado (url de base, cabeceras, ...)
- *data*: Son los datos en si de la petición (en este caso la lista de películas)
- *headers*: Son las cabeceras, hay algunos estándares como por ejemplo indicar que el contenido de *data* va a venir en formato *JSON* (podría venir en formato texto o crudo), y también se pueden usar para enviar una cabecera que nos permita identificarnos en el servidor.
- *request*: Axios no deja de ser un azúcar que se monta encima de *fetch* o *XMLHttpRequest* (depende de cómo se configure), en este campo esta la request en bruto (a bajo nivel).
- *status*: Es el código de respuesta *HTTP* que nos devuelve el servidor. Aquí nos devuelve un 200, que significa que la petición se ha realizado correctamente.
- *statusText*: Es el mensaje de respuesta *HTTP* que nos devuelve el servidor. Nos ha devuelto un "OK", que significa que la petición se ha realizado correctamente.

Errores

Pero ¿qué pasaría si tuviésemos un error en la petición *HTTP*?

Para simular un error en la petición *HTTP* vamos a modificar nuestro código.

Vamos a cambiar la dirección del servidor, en vez de poner `http://localhost:3000/movies` vamos a poner `http://localhost:3000/noexiste`.

main.js

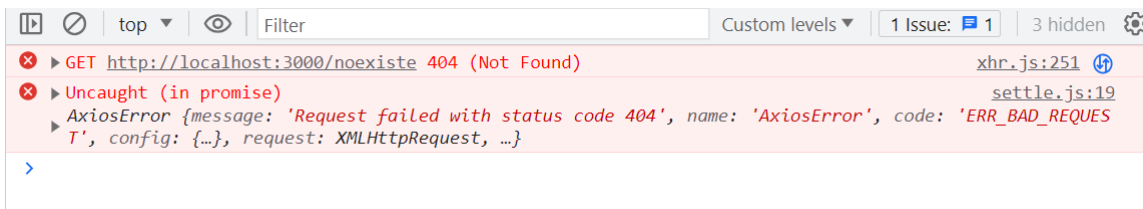
```
import Axios from "axios";

- const peliculasPromesa = Axios.get("http://localhost:3000/movies");
+ const peliculasPromesa = Axios.get("http://localhost:3000/noexiste");

peliculasPromesa.then((response) => {
  console.log(response.data);
});

- console.log("¿Me ejecuto antes o después que la promesa se cumpla?");
```

Si ahora abrimos la consola del navegador, veremos que nos muestra un error.



¿Qué ha pasado aquí?

- El servidor si lo alcanzamos.
- El endpoint *noexiste*, no lo puede resolver y nos da un error, en este caso un *404*, es el típico error de página o recursos no encontrado en el servidor (en breve veremos más mensajes de error y que quieren decir).

¿Cómo podemos gestionar los errores?

En el caso de *Axios* podemos usar el método *catch* de las promesas para gestionar los errores.

Podríamos hacer algo tal que:

main.js

```
import Axios from "axios";

const peliculasPromesa = Axios.get("http://localhost:3000/movies");

peliculasPromesa.then((response) => {
  console.log(response.data);
});

+ peliculasPromesa.catch((error) => {
+   console.log(error);
+ });
```

Incluso lo podríamos customizar un poco más, para que nos muestre un mensaje más amigable 😊.

main.js

```
import Axios from "axios";

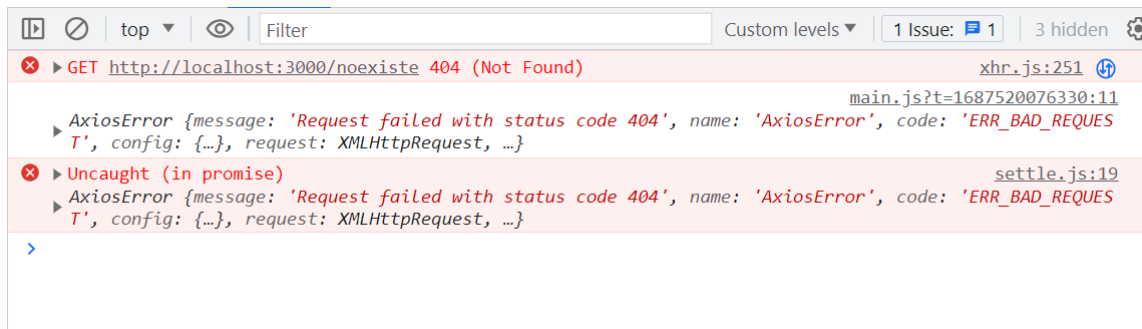
const peliculasPromesa = Axios.get("http://localhost:3000/movies");

peliculasPromesa.then((response) => {
  console.log(response.data);
});

peliculasPromesa.catch((error) => {
+   console.log("Ocurrió un error al obtener las películas:", error);
});
```

```
- console.log(error);
});
```

Si abrimos ahora la consola del navegador, veremos ese mensaje.



Fíjate que aunque capturemos el error, este seguirá saliendo por consola.

Esta aproximación no está mal, pero te puede costar un poco leer ese código, ¿Sabes qué? Puedes encadenar el *then* y el *catch*, y así te queda todo más compacto:

main.js

```
import Axios from "axios";

const peliculas = Axios.get("http://localhost:3000/movies");

- peliculas.then((response) => {
-   console.log(response.data);
- });

- peliculas.catch((error) => {
-   console.log("Ocurrió un error al obtener las películas:", error);
- });

+ peliculas
+   .then((response) => {
+     console.log(response.data);
+   })
+   .catch((error) => {
+     console.log("Ocurrió un error al obtener las películas:", error);
+   });
```

¿Qué estamos haciendo aquí? Aplicar el *patrón cadena*, esto lo vas a usar a menudo en *JavaScript*, esto lo aprenderemos más adelante.

All

Imagínate que en tu aplicación, no puedes activar los botones de edición hasta que tengamos los datos de películas y los datos de actores cargados.

¿Con lo que sabemos hasta ahora, qué podríamos intentar?

Podemos probar varias aproximaciones que te van a *oler mal*

```
import Axios from "axios";

const cuantosDatos = 0;
const peliculasPromise = Axios.get("http://localhost:3000/movies");
const actoresPromise = Axios.get("http://localhost:3000/actors");

const actores = [];
const peliculas = [];

const ponteEnModoEdicion = () => {
  console.log("Todos los datos cargados, ya me puedo poner en modo edición");
  console.log(actores);
  console.log(peliculas);
};

peliculasPromise.then((response) => {
  peliculas = response.data;
  cuantosDatos++;
  if (cuantosDatos === 2) {
    ponteEnModoEdicion();
  }
});

actoresPromise.then((response) => {
  actores = response.data;
  cuantosDatos++;
  if (cuantosDatos === 2) {
    ponteEnModoEdicion();
  }
});
```

Bueno, esto "parece que funciona", pero...:

- Hemos puesto un montón de código.
- Para dos promesas buueeenooo... imagínate que tuviéramos 5.
- ¿Cómo gestionamos errores aquí? Sería también bastante trabajoso.

Vamos a probar otra aproximación:

```
import Axios from "axios";

const peliculasPromise = Axios.get("http://localhost:3000/movies");
const actoresPromise = Axios.get("http://localhost:3000/actors");

const actores = [];
const peliculas = [];

const ponteEnModoEdicion = () => {
```

```

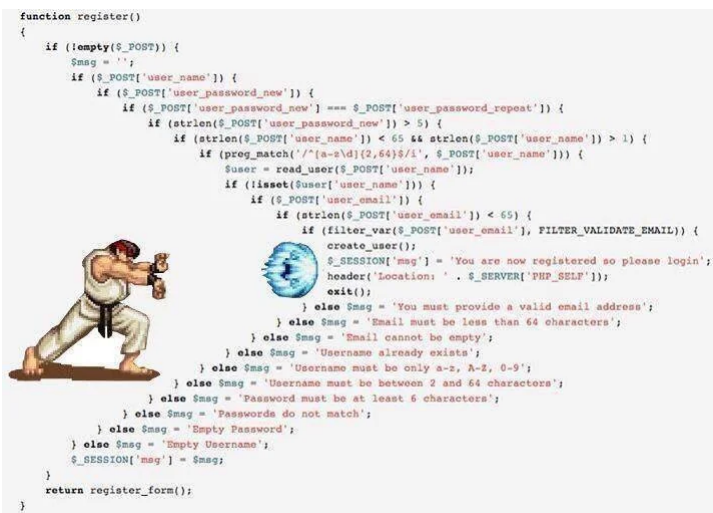
console.log("Todos los datos cargados, ya me puedo poner en modo edición");
console.log(actores);
console.log(peliculas);
};

peliculasPromise.then((response) => {
  peliculas = response.data;
  actoresPromise.then((response) => {
    actores = response.data;
    ponteEnModoEdicion();
  });
});
});

```

Aquí podríamos probar a poner un *catch* en la promesa de arriba y capturar errores, pero ¿Qué tiene esto de malo?

- Estamos montando un *hadouken* o como también se conoce un *árbol de navidad*, es decir llamada anidadas unas dentro de otras, esto hace que el código sea más difícil de leer (a esto se le llama de manera formal *complejidad ciclomática*).



```

function register()
{
  if (!empty($_POST)) {
    $msg = '';
    if ($_POST['user_name']) {
      if ($_POST['user_password_new']) {
        if ($_POST['user_password_new'] == $_POST['user_password_repeat']) {
          if (strlen($_POST['user_password_new']) > 5) {
            if (strlen($_POST['user_name']) < 65 && strlen($_POST['user_name']) > 1) {
              if (preg_match('/^[a-z\d]{2,64}$/i', $_POST['user_name'])) {
                $user = read_user($_POST['user_name']);
                if (!isset($user['user_name'])) {
                  if ($_POST['user_email']) {
                    if (strlen($_POST['user_email']) < 65) {
                      if (filter_var($_POST['user_email'], FILTER_VALIDATE_EMAIL)) {
                        create_user();
                        $_SESSION['msg'] = 'You are now registered so please login';
                        header('Location: ' . $_SERVER['PHP_SELF']);
                        exit();
                      } else $msg = 'You must provide a valid email address';
                    } else $msg = 'Email must be less than 64 characters';
                  } else $msg = 'Email cannot be empty';
                } else $msg = 'Username already exists';
              } else $msg = 'Username must be only a-z, A-Z, 0-9';
            } else $msg = 'Username must be between 2 and 64 characters';
          } else $msg = 'Password must be at least 6 characters';
        } else $msg = 'Passwords do not match';
      } else $msg = 'Empty Password';
    } else $msg = 'Empty Username';
    $_SESSION['msg'] = $msg;
  }
  return register_form();
}

```

¿No hay una manera más fácil de hacer esto? JavaScript nos da una solución, es el *all* que lo tenemos tanto a nivel de *axios* como a nivel de *promesas* en si (esto lo veremos con *fetch* más adelante).

¿Qué es lo que hace? Le pasamos un array de promesas y hasta que todas no están resueltas no se ejecuta el *then*.

Vamos a ver cómo será la solución con *promise.all*

main.js

```

import Axios from "axios";

+ const peliculasPromesa = Axios.get("http://localhost:3000/movies");
+ const actoresPromesa = Axios.get("http://localhost:3000/actors");

+ Axios.all([peliculasPromesa, actoresPromesa]).then((response) => {

```

```
+ console.log(response);
+ });
```

Si ahora abrimos la consola del navegador, veremos que nos muestra un array con la respuesta de cada una de las promesas.

[main.js?t=1687446499398:8](#)

```
▼ (2) [{...}, {...}] ⓘ
  ▶ 0: {data: Array(27), status: 200, statusText: 'OK', headers: AxiosHeaders, config: {...}, ...}
  ▶ 1: {data: Array(27), status: 200, statusText: 'OK', headers: AxiosHeaders, config: {...}, ...}
    length: 2
  ▶ [[Prototype]]: Array(0)
```

Y si quisiéramos obtener la respuesta de cada una de las promesas, tendríamos que hacer lo siguiente:

main.js

```
import Axios from "axios";

const peliculas = Axios.get("http://localhost:3000/movies");
const actores = Axios.get("http://localhost:3000/actors");

Axios.all([peliculas, actores]).then((response) => {
- console.log(response);
+ console.log(response[0].data);
+ console.log(response[1].data);
});
```

Si miramos la consola del navegador, veremos que nos muestra la información de cada una de las promesas.

[main.js?t=1687446941005:8](#)

```
▶ (27) [{...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}]
```

[main.js?t=1687446941005:9](#)

```
▶ (27) [{...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}]
```

¿Qué es lo que pasa si una de las promesas falla?

Vamos a cambiar el endpoint de *movies* por *noexiste* y añadimos un catch para capturar el error.

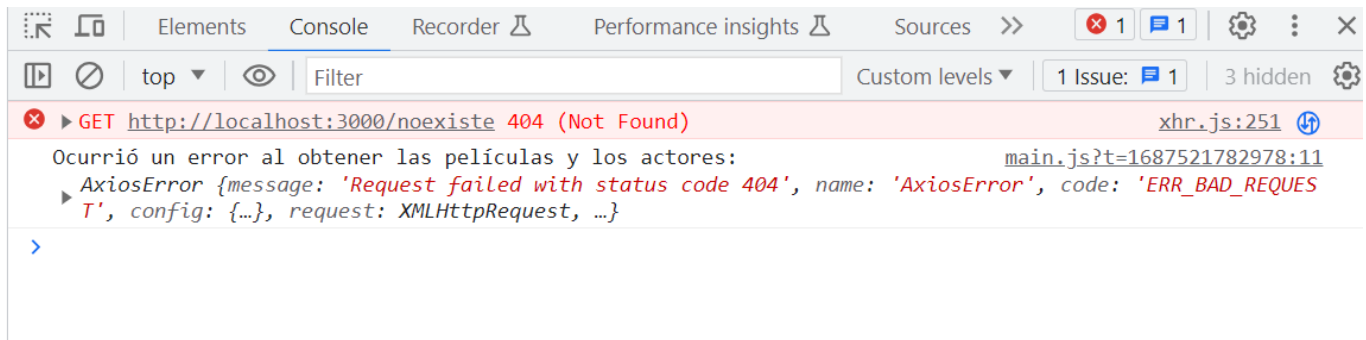
main.js

```
import Axios from "axios";

- const peliculasPromesa = Axios.get("http://localhost:3000/movies");
+ const peliculasPromesa = Axios.get("http://localhost:3000/noexiste");
const actoresPromesa = Axios.get("http://localhost:3000/actors");
```

```
Axios.all([peliculasPromesa, actoresPromesa]).then((response) => {
  console.log(response);
})
+ .catch((error) => {
+   console.log("Ocurrió un error al obtener las películas y los actores:", error);
+ });
```

Y como vemos por la consola del navegador, no sigue ejecutando el *then*, nos muestra el error y nos devuelve un 404 (not found), es decir, no ha encontrado el endpoint.



Race

Esperar a que todas las promesas se hayan completado con éxito, cubre ciertos escenarios, pero hay ocasiones en las que lo que quiero es quedarme con la más rápida ¿Comooooor? Te pongo un ejemplo:

Imagínate que tenemos que desarrollar un portal de reservas hoteleras tipo booking, es muy importante que la página de respuesta cuanto antes (por curiosidad: [Sabías que un segundo de latencia le puede hacer perder 1600 millones de dolares en venta a Amazon](#))

¿Cómo funciona esto? Tu portal de reservas pide a varios proveedores, si a varias APIs REST, es lo que se llaman *bancos de camas*, te interesa por un lado dar respuesta cuanto antes al cliente (aunque no sea el precio óptimo), y después en paralelo ya ir pidiendo los datos con más calma y actualizando la mejor oferta (algo parecido lo puedes ver en la web de vuelos *skyscanner*).

Vamos al caso de los hoteles:

- Pido disponibilidad a todos.
- Me quedo con la respuesta más rápida.

¿Cómo lo haríamos?

- Vamos a crear dos APIs que tengan el mismo listado de hoteles, pero con precio diferentes.
- Vamos a usar el método [Promise.race](#) de *JavaScript* para quedarnos con la respuesta más rápida de las dos APIs.

Antes de arrancar, asegúrate que has parado el servidor de API Rest de actores y películas.

En este [repositorio de GitHub](#) tenemos el código de las dos APIs que vamos a utilizar.

Clonamos el repositorio en nuestro equipo, o nos descargamos el código en nuestro equipo.

Una vez hecho esto, tenemos la carpeta *10-asincronia/04-banco-camas-1* y *10-asincronia/04-banco-camas-2* que son las dos API que vamos a utilizar para hacer las pruebas.

Las copiamos las dos en otra carpeta para no mezclarlo con el resto de material del bootcamp.

Navegamos a la carpeta *04-banco-camas-1*

```
cd 04-banco-camas-1
```

Instalamos las dependencias de la API.

```
npm install
```

Arrancamos la API.

```
npm start
```

Hacemos lo mismo con la carpeta *05-banco-camas-2*.

```
cd 05-banco-camas-2
```

Instalamos las dependencias de la API.

```
npm install
```

Arrancamos la API.

```
npm start
```

Veremos que una está corriendo en el puerto 3000 y la otra en el puerto 3001.



Asegúrate antes de haber parado el servidor api de películas y actores

Si ahora hacemos uso del método *Promise.race* de *JavaScript* para quedarnos con la primera respuesta que se cumpla, tendríamos que hacer lo siguiente:

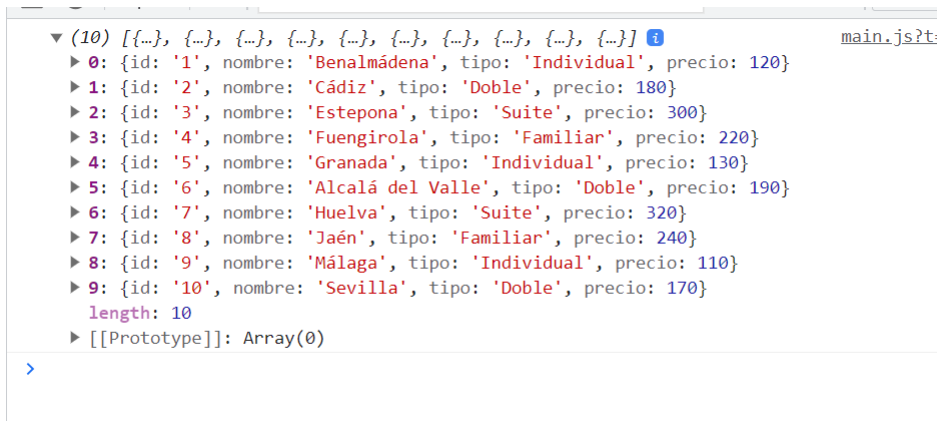
main.js


```
import Axios from "axios";

const bancoCamas1Promesa = Axios.get("http://localhost:3000/habitaciones");
const bancoCamas2Promesa = Axios.get("http://localhost:3001/habitaciones");

Promise.race([bancoCamas1Promesa, bancoCamas2Promesa]).then((response) => {
  console.log(response.data);
});
```

Si miramos en la consola vemos lo siguiente:



¿Qué ha pasado aquí? Pues que la primera promesa que se resuelve es la ganadora.

Existen trucos para saber de dónde viene la promesa ganadora, y es creando una promesa custom que envuelva a la promesa de axios y le añadir un identificador.

Creando una promesa

Hasta ahora hemos visto cómo podemos hacer peticiones *HTTP* al servidor, pero también podemos crear nuestras propias promesas.

¿Cómo podemos crear una promesa?

Para crear una promesa tenemos que utilizar la palabra reservada *new* y el constructor *Promise*.

Si miramos la [documentación de Promise](#) veremos que es una función, y esta función recibe dos argumentos, *resolve* y *reject*.

- Con *resolve* le decimos que la promesa se ha cumplido.
- Con *reject* le decimos que la promesa no se ha cumplido (que ha dado un error).

Vamos a empezar con un caso simple, sin acceder a servidor:

Si queremos crear una promesa en la que devolvemos un objeto con películas y actores, tendríamos que hacer lo siguiente:

main.js

```
const peliculas = new Promise((resolve, reject) => {
  resolve({
    movies: [
      {
        id: 1,
        title: "El señor de los anillos",
      },
      {
        id: 2,
        title: "El padrino",
      },
    ],
    actors: [
      {
        id: 1,
        name: "Al Pacino",
      },
      {
        id: 2,
        name: "Robert De Niro",
      },
    ],
  });
});
```

¿Qué estamos haciendo aquí?

- Estamos creando una promesa.
- Le estamos diciendo que se ha cumplido la promesa y que nos devuelva un objeto con películas y otro objeto con actores.

¿Cómo recuperamos la información de la promesa?

Para recuperar la información de la promesa tenemos que utilizar el método *then*.

main.js

```
const peliculasPromesa = new Promise((resolve, reject) => {
  resolve({
    movies: [
      {
        id: 1,
        title: "El señor de los anillos",
      },
      {
        id: 2,
        title: "El padrino",
      },
    ],
    actors: [
      {
```

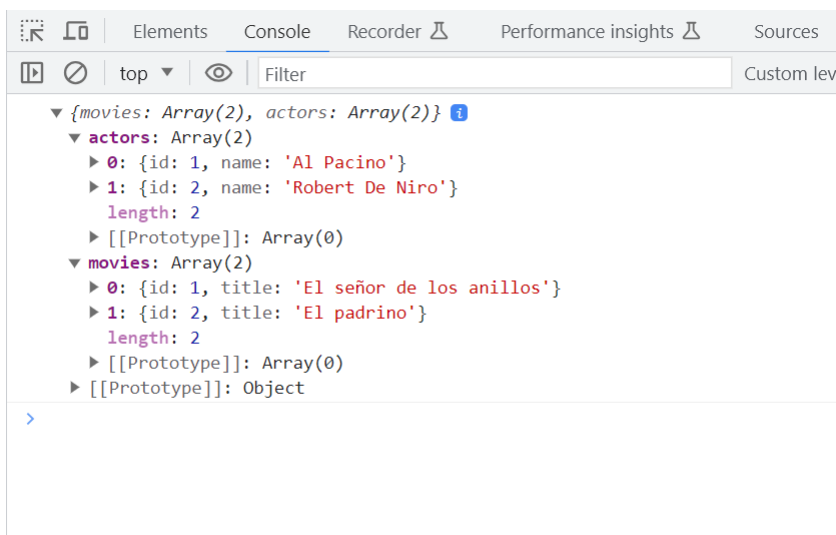
```

        id: 1,
        name: "Al Pacino",
      },
      {
        id: 2,
        name: "Robert De Niro",
      },
    ],
  });
});

+ peliculasPromesa.then((response) => {
+   console.log(response);
+ });

```

Si miramos ahora la consola del navegador, veremos que nos muestra la información que le hemos pasado a la promesa.



Pero, ¿qué pasa si queremos decirle que ha habido un fallo y no podemos completar la promesa?

Para esto tenemos que utilizar el método *reject*.

main.js

```

const peliculas = new Promise((resolve, reject) => {
-   resolve({
-     movies: [
-       {
-         id: 1,
-         title: "El señor de los anillos",
-       },
-       {
-         id: 2,
-         title: "El padrino",
-       },
-     ],
-     actors: [

```

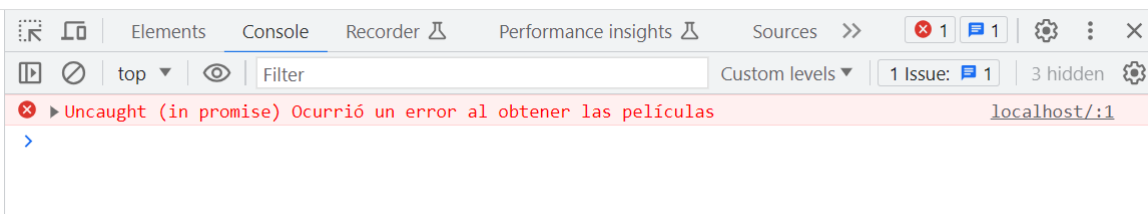
```

-     {
-       id: 1,
-       name: "Al Pacino",
-     },
-     {
-       id: 2,
-       name: "Robert De Niro",
-     },
-   ],
- });
+ reject("Ocurrió un error al obtener las películas");
});

peliculas.then((response) => {
  console.log(response);
});

```

Si miramos ahora la consola del navegador, veremos que nos muestra el error que le hemos pasado a la promesa.



Ahora vamos a ver cómo crear una promesa custom envolviendo una llamada axios de verdad, y vamos a crear una función que nos devuelva una promesa.

Antes de seguir asegurate que tienes las API Rest de bancos de camas paradas, y las de películas y actores en marcha.

Creamos un función *leeActores* que nos devuelva una promesa.

```

const leeActores = () => {
  const promise = new Promise((resolve, reject) => {
    Axios.get("http://localhost:3000/actors").then((response) => {
      resolve(response.data);
    });
  });

  return promise;
};

```

Otra forma muy común de usarlo sería quitarle el *.data*, y usar el destructuring de *JavaScript* y quedarnos con la propiedad *data*.

```
const leeActores = () => {
-   const Axios.get("http://localhost:3000/actores").then((response) => {
-       return response.data;
-   });
+   return Axios.get("http://localhost:3000/actors").then(({ data }) => data);

};
```

¿Os acordáis del ejemplo de los bancos de cama y al *race*? Vamos a ver cómo podemos identificar que api rest es la que ha ganado, partíamos de este código:

Aquí previamente paramos la API de películas y volvemos a levantar las de los bancos de cama

```
import Axios from "axios";

const bancoCamas1Promesa = Axios.get("http://localhost:3000/habitaciones");
const bancoCamas2Promesa = Axios.get("http://localhost:3001/habitaciones");

Promise.race([bancoCamas1Promesa, bancoCamas2Promesa]).then((response) => {
    console.log(response.data);
});
```

Vamos a crear una promesa custom en cada llamada y añadirle una propiedad fuente que diga si veiens del bancoA o del BancoB.

```
import Axios from "axios";

- const bancoCamas1Promesa = Axios.get("http://localhost:3000/habitaciones");
- const bancoCamas2Promesa = Axios.get("http://localhost:3001/habitaciones");

+ const bancoCamas1Promesa = new Promise((resolve, reject) => {
+   Axios.get("http://localhost:3000/habitaciones").then((response) => {
+     resolve({
+       ...response,
+       fuente: "bancoA",
+     });
+   });
+ });
+
+ const bancoCamas2Promesa = new Promise((resolve, reject) => {
+   Axios.get("http://localhost:3001/habitaciones").then((response) => {
+     resolve({
+       ...response,
+       fuente: "bancoB",
+     });
+   });
+ });
```

```
Promise.race([bancoCamas1Promesa, bancoCamas2Promesa]).then((response) => {  
+ console.log("**** FUENTE: ", response.fuente);  
  console.log(response.data);  
});
```

Async / Await

Trabajar con promesas y *then* está muy bien, pero a veces el código puede costar un poco de leer.

¿No habría una forma de escribir este código de manera lineal y que se ejecutara de forma asíncrona? Para ello, en JavaScript introdujeron *async* y *await*, un azúcar que por debajo lo que usa en realidad son promesas, pero nos hace la vida mucho más fácil a los programadores.

Fue introducido en ES2017, y depende de que navegador estes usando puede que tengas que realizar algún setup especial en tu proyecto.

Async / Await

Que nos hace falta definir:

- con *Async* indicamos que esa función va a ser asíncrona (es decir que lo que vaya a devolver sea que hay que envolverlo en una promesa).
- con *Await* indicamos que queremos esperar a que se resuelva la promesa.

Vamos a ver cómo utilizar *_async** y *_await** con una llamada a servidor, suponemos que:

- Tenemos una función específica para hacer una llamada a la API REST (leePelículas).
- Otra función específica que sirve para mostrar los datos (hace la llamada a la función previa y se supone que lo enlazaría todo con los elementos HTML etc...)

Un tema a tener en cuenta: en toda función que use *await* tenemos que poner *async*, nos ponemos manos a la obra:

Asegúrate que tienes la API de películas y actores levantada (y las de bancos de camas paradas).

main.js

```
const leePelículas = async () => {  
  const peliculas = await Axios.get("http://localhost:3000/movies");  
  console.log("2. Películas cargadas");  
  
  return peliculas;  
};  
  
const muestraDatos = async () => {  
  const datos = await leePelículas();  
  console.log(datos.data);  
  console.log("3. Datos cargados");  
};
```

```
muestraDatos();  
console.log("1. Yo me muestro antes que el resto de mensajes");
```

Y ahora viene algo que echabas de menos.... TE TOCA:

- Crear un método que se llame `leeActores` que haga una llamada a la API de actores.
- Dentro de `muestra` haz un `await` de `leeActores` y muestra los datos por consola.
- Una vez que lo tengas planteate si esta aproximación tiene alguna pega 🤔.

Pistas:

- El método `leeActores` prácticamente es fusilar `leePelículas`
- En `muestra datos` ponlo juntos después de `leePelículas` con un `await` y luego haz un `console.log` de los datos.
- Lo que va mal... dale al coco cuando veamos la solución dirás, pero si para eso teníamos algo que me permitía esperar a varias promesas en paralelo.

La solución

```
const leePelículas = async () => {  
  const peliculas = await Axios.get("http://localhost:3000/movies");  
  console.log("2. Películas cargadas");  
  
  return peliculas;  
};  
  
+ const leeActores = async () => {  
+   const peliculas = await Axios.get("http://localhost:3000/actors");  
+   console.log("2. Actores cargados");  
+  
+   return actores;  
+};  
  
const muestraDatos = async () => {  
  const datos = await leePelículas();  
  console.log(datos.data);  
+   const datosActores = await leeActores();  
+   console.log(datosActores.data);  
  console.log("3. Datos cargados");  
};
```

¿Qué problema tiene esta aproximación? Pues que primero cargamos *películas* y después cargamos *actores*, estamos perdiendo un tiempo precioso...y la solución es usar *promise.all* para cargar las dos cosas en paralelo.

```
const muestraDatos = async () => {  
-   const datos = await leePelículas();
```

```
- console.log(datos.data);
- const datosActores = await leeActores();
- console.log(datosActores.data);
+ const response = await Axios.all([leePelículas(), leeActores()]);
+ console.log(response[0]);
+ console.log(response[1]);
  console.log("3. Datos cargados");
};
```

Control de errores

¿Y cómo puedo hacer un control de errores? Pues con un try catch me vale, vamos a teclear mal un endpoint y poner un try catch:

```
const muestraDatos = async () => {
+ try {
  const response = await Axios.all([leePelículas(), leeActores()]);
  console.log(response[0]);
  console.log(response[1]);
  console.log("3. Datos mostrados");
+ }
+ catch (error) {
+   console.log("Ocurrió un error al obtener los datos:", error);
+ }
};
```

E introducimos un error:

```
const leeActores = async () => {
-   const actores = await Axios.get("http://localhost:3000/actors");
+   const actores = await Axios.get("http://localhost:3000/noexisto");

  console.log("2. Actores cargados");

  return actores.data;
};
```

Ahora lo hacemos con Fetch

Hasta ahora hemos estado utilizando *axios* para hacer peticiones *HTTP* al servidor, pero también podemos utilizar el método *fetch*, que es el estándar de JavaScript para hacer peticiones *HTTP* y se incorporó en ES6 como parte de la Fetch API.

¿Cómo funciona Fetch?

Fetch es una función que recibe como parámetro la dirección del servidor y nos devuelve una promesa, *vaya esto es parecido a Axios ¿Por qué hemos usado Axios en vez de fetch?* Porque en el caso de *fetch* tenemos que hacer algún paso adicional:

- Primero creamos la petición HTTP.
- Obtenemos la respuesta del servidor, pero no los datos, nos está diciendo, oye tengo una respuesta y el contenido está en formato JSON (podría venir en otro formato).
- Así que encadenamos a esa promesa el método *json* que nos devuelve otra promesa con los datos que nos ha devuelto el servidor.
- Toca tirar del *then* último para obtener los datos.

¿Y tanto lío para qué? Pues para darnos mayor flexibilidad, aunque si te cuento un secreto... lo primero que uno hace cuando va a hacer *fetch* es hacerse una función de ayuda para ahorrarse tantos pasos 😊.

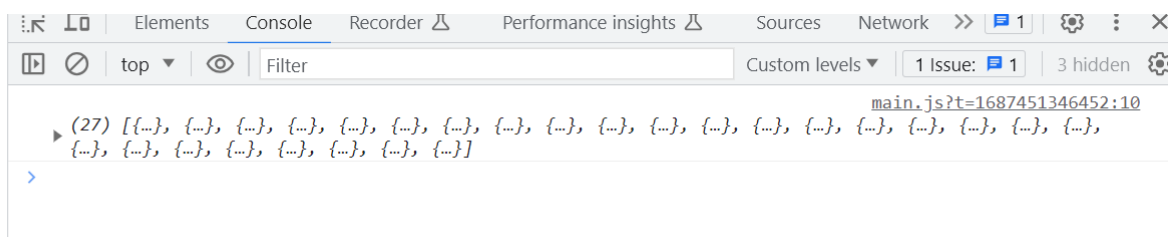
¿Cómo lo haríamos?

main.js

```
const peliculasPromesa = fetch("http://localhost:3000/movies");

peliculasPromesa
  .then((response) => {
    return response.json();
  })
  .then((data) => {
    console.log(data);
  });
```

Si ahora abrimos la consola del navegador, veremos que nos muestra la información que nos devuelve el servidor.



Te toca... haz lo mismo con los actores.

Pistas:

- Esto es una copia y pega de lo que acabamos de hacer.
- haz un fetch para los actores.
- Tira de los then que toquen y muestra los datos por consola.

```
const peliculasPromesa = fetch("http://localhost:3000/movies");

peliculasPromesa
```

```

    .then((response) => {
      return response.json();
    })
    .then((data) => {
      console.log(data);
    });

+ const actoresPromesa = fetch("http://localhost:3000/actors");

+ actoresPromesa
+   .then((response) => {
+     return response.json();
+   })
+   .then((data) => {
+     console.log(data);
+   });

```

También podemos envolver esto es una función:

```

- const actoresPromesa = fetch("http://localhost:3000/actors");

- actoresPromesa
-   .then((response) => {
-     return response.json();
-   })
-   .then((data) => {
-     console.log(data);
-   });

+ const leeActores = () => {
+   return fetch("http://localhost:3000/actors")
+     .then((response) => {
+       return response.json();
+     })
+     .then((data) => {
+       return data;
+     });
+ };
+ }
+
+ leeActores().then((data) => {
+   console.log(data);
+ });

```

¿Cómo gestionamos los errores?

Pero ¿qué pasa si tenemos un error en la petición *HTTP*?

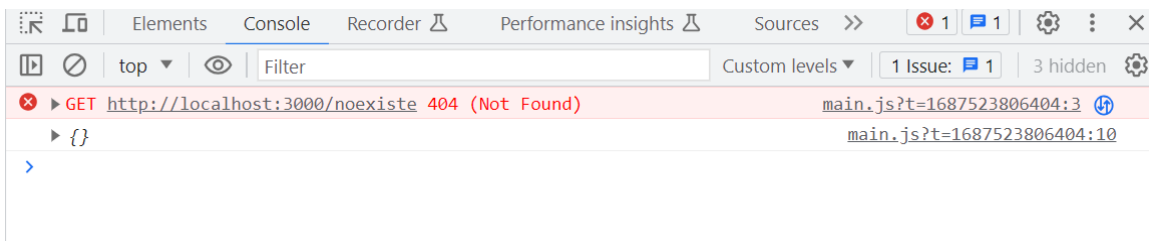
Para simular un error en la petición *HTTP* vamos a modificar nuestro código.

Vamos a cambiar, de nuevo, el endpoint del servidor, en vez de poner `http://localhost:3000/movies` vamos a poner `http://localhost:3000/noexiste`.

main.js

```
const leeActores = () => {  
  - return fetch("http://localhost:3000/actors")  
  + return fetch("http://localhost:3000/noexiste")  
    .then((response) => {  
      return response.json();  
    })  
    .then((data) => data)  
  + .catch((error) => console.log("ERROR !!!! ", error));  
};
```

Si ahora miramos la consola del navegador, vemos que nos aparece un 404 (not found), es decir, que no encuentra el endpoint.



Pero ni rastro de nuestro error, ¿Qué pasa aquí? Qué *fetch* considera que esto está dentro del flujo de ejecución esperado, y en *response* nos da más detalla, es decir que podemos lanzar el error de la siguiente manera:

- Primero comprobamos que el *response* no ha sido "ok".
- Y lanzamos un *throw* con el mensaje de error.

main.js

```
const leeActores = () => {  
  return fetch("http://localhost:3000/noexiste")  
    - .then(response => response.json())  
    + .then((response) => {  
    +   if (!response.ok) {  
    +     throw new Error("No se pudo realizar la petición");  
    +   }  
    +   return response.json();  
    + })  
    .then((data) => data)  
    .catch((error) => console.log(error));  
};
```

Async / Await con Fetch

Y cómo podemos usar *async* y *await* con *fetch*.

Vamos a hacerlo con el listado de actores.

```
const leeActores = async () => {  
  const response = await fetch("http://localhost:3000/noexiste");  
  if (!response.ok) {  
    throw new Error("No se pudo realizar la petición");  
  }  
  
  const data = response.json();  
  
  return data;  
};
```

Y podemos hacer:

```
const actores = await leeActores();  
console.log(actores);
```

Y ahora te toca... ¿Te animas a hacer lo mismo con la lista de películas? y ya que estás prueba a poner un all.

La solución:

```
const leePelículas = async () => {  
  const response = await fetch("http://localhost:3000/movies");  
  if (!response.ok) {  
    throw new Error("No se pudo conectar con el servidor");  
  }  
  const data = await response.json();  
  
  return data;  
};
```

Y donde toque en código llamamos a:

```
const actores = await leeActores();  
console.log(actores);  
+ const películas = await leePelículas();  
+ console.log(películas);
```

¿Y si usáramos *Promise.all* para hacer las dos peticiones a la vez?

```
- const actores = await leeActores();  
- console.log(actores);  
- const peliculas = await leePeliculas();  
- console.log(peliculas);  
  
+ const resultado = await Promise.all([leeActores(), leePeliculas()]);  
  
+ console.log(resultado[0]);  
  
+ console.log(resultado[1]);
```