CREAR PACKAGE.JSON

pwd: Saber donde estoy ubicado.  
mkdir: Creación de carpeta  
cd: Moverse a carpetas  
git init: Inicias Git  
npm init: Le da nombre, versión, entre otras cosas al proyecto  
code . : Inicia el editor de códigos

RECURSO

**Platzi Fake Store API**

Fake store rest API for your e-commerce or shopping website prototype.

<https://fakeapi.platzi.com/>

exercise.js

1

2

3

4

export·function·execCallback(callback)·{

··window.setTimeout(callback,·2000);

}



Intentos

Correr Pruebas

GuíaVistaPruebas

**Should execute the callback**

**Should execute the callback after 2s**

¡Felicidades, todas las pruebas pasaron!

# XMLHTTPRequest

8/26

**𝗖𝗹𝗮𝘀𝗲 #𝟳: 𝗫𝗠𝗟𝗛𝗧𝗧𝗣𝗥𝗲𝗾𝘂𝗲𝘀𝘁𝟳/𝟮𝟭** 📤  
.  
📲 **XMLHttpRequest** es un objeto de JS que permite hacer peticiones hacia servicios en la nube(URLs o APIs).  
.  
📪 **Existen 5 estados en un llamado XMLHttpRequest:**  
.

* **0** → Se ha inicializado.
* **1** → Loading (cargando).
* **2** → Se ha cargado.
* **3** → Procesamiento si existe alguna descarga.
* **4** → Completado.  
  .

📫 **Métodos y propiedades:**  
.  
**xmlhttp.open()** → Prepara la petición para ser enviada tomando tres parámetros: prótocolo, url, asíncrono (true).  
**xmlhttp.readyState** → Retorna el estado de la petición.  
**xmlhttp.onreadystatechange** → Un eventHandler que es llamado cuando la propiedad readyState cambia.  
**xmlhttp.status** → Retorna el estado de la respuesta de la petición. (200,400,500)  
**xmlhttp.send()** → Envía la petición.  
.  
📬 **Características del protocolo http:**  
.  
**Verbos**: Los verbos indican acciones que están asociadas a peticiones y recursos, es decir, sirven para la manipulación de recursos cliente/servidor. Los Verbos http son:

* **GET** → Solicita un recurso.
* **HEAD** → Solicita un recurso pero sin retornar información, la estructura de esta petición es igual que get tanto en su headers como estatus. Es útil cuando vamos a utilizar API, para comprobar si lo que vamos a enviar esta correcto y puede ser procesado.
* **POST** → Sirve para la creación de recursos en el servidor.
* **PUT** → Actualiza por completo un recurso, reemplaza todas las representaciones actuales del recurso de destino con la carga útil de la petición.
* **PATCH** → Actualiza parcialmente un recurso.
* **DELETE** → Elimina un recurso.  
  .

📭 **Los códigos de estados del servidor:**  
.  
El código de estado (status codes) sirve para describir el estado de la petición hecha al servidor.

* **1xx** → Indican que la petición fue recibida por el servidor, pero está siendo procesada por el servidor.
* **2xx** → Indican que la petición fue recibida, aceptada y procesada correctamente.
* **3xx** → Indican que hay que tomar acciones adicionales para completar la solicitud.
* **4xx** → Indican errores del lado del cliente que hizo mal una solicitud.
* **5xx** → Indican errores del servidor. Suelen aparecer cuando existe un fallo en la ejecución en el servidor.  
  .

📧 **Los códigos más comunes a la hora de interactuar con una API son:**  
.

* **200** → OK → Indica que todo está correcto.
* **201** → Created → Todo está correcto cuando se hizo una solicitud POST, el recurso se creó y se guardó correctamente.
* **204** → No Content → Indica que la solicitud se completó correctamente pero no devolvió información. Este es común cuando se hacen peticiones con el verbo DELETE.
* **400** → Bad Request → Indica que algo está mal en la petición (no encontró algo).
* **401** → Unauthorized → Significa que antes de hacer una solicitud al servidor nos debemos autenticar.
* **403** → Forbidden → Indica que no tenemos acceso a ese recurso aunque se esté autenticado.
* **404** → Not Found → Indica que no existe el recurso que se está intentando acceder.
* **500** → Internal Server Error → Indica que algo falló, es un error que retorna el servidor cuando la solicitud no pudo ser procesada.  
  .  
  Fuente: [aquí](https://en.wikipedia.org/wiki/List_of_HTTP_status_codes)  
  .

🖍️ **Ejemplo en VSC:**  
.

1. Ir a la consola y ubicarnos en la carpeta del proyecto y escribir el comando para instalar el paquete **XMLHttpRequest**:  
   **npm i xmlhttprequest**
2. Ir al VSC y crear un archivo llamado **challenge.js** en la ruta **src/callback**. El archivo queda:

**const** XMLHttppRequest = requiere('xmlhttprquest'); //llamado al XmlHttpRequest

**const** API = 'https://api.escuelajs.co/api/v1'; //API en mayúscula porque es una referencia que no va a cambiar

**function** **fetchData**(urlApi, callback){ //urlApi: no confundir y colocar API

**let** xhttp = **new** XMLHttppRequest(); //referencia a new XMLHttpRequest

xhttp.open('GET', urlApi, **true**); //petición "obtener" con true para habilitarlo

xhttp.onreadystatechange = **function**(event) { //escucha diferentes estados de la solicitud y conocer cuando está disponible la información

**if**(xhttp.readyState === 4) { //si el estado ha sido completada la llamada

**if**(xhttp.status === 200 ){ //el servido responde de forma correcta

callback(**null**, JSON.parse(xhttp.responseText)); //dentro de xhttp.responseTex recibimos lo que entrega el servidor en texto y se hace la transformación en JSON

}

} **else** {

**const** error = **new** Error('Error' + urlApi);

**return** callback(error,**null**); //es null porque no se está regresando ningún dato

}

}

xhttp.send();

}

La nueva forma de hacer peticiones a una API es el [fetch](https://developer.mozilla.org/es/docs/Web/API/Fetch_API" \t "_blank).

🤞🏼 **𝗖𝗹𝗮𝘀𝗲 #𝟭𝟬: 𝗤𝘂é 𝘀𝗼𝗻 𝗹𝗮𝘀 𝗽𝗿𝗼𝗺𝗲𝘀𝗮𝘀 𝟭𝟬/𝟮𝟭** 🤞🏼  
.  
🪃 Un **callback** devuelve una función en los parámetros, cuando llamamos varias veces un callback, estaremos colocando muchas lineas de código y sería engorroso, por eso nacen las promesas, éstas optimizan y permiten leer mejor el código con pocas lineas.  
.  
🫱🏼‍🫲🏾 Las **promesas** son asíncronas, por lo que el código continuará su ejecución normalmente y luego dirá si la promesa se resolvió o se rechazó. Por lo que varias promesas pueden llegar a entrar en ejecución al mismo tiempo.  
.  
Las promesas pueden suceder:  
.

* Ahora
* En el futuro
* Nunca  
  .

🛠️ **Para crear una promesa:**  
.  
Utilizamos la palabra reservada **new** seguida de la palabra **Promise** que es el constructor de la promesa. Este constructor recibe un único parámetro que es una función, la cuál a su vez, recibe otros dos parámetros: **resolve** y **reject**.

* El parámetro **resolve** se utiliza para cuando la promesa devuelve el valor correctamente.
* El parámetro **reject**, se usa en el que caso de que no funcione.  
  .  
  📝 Ejemplo:

**const** promise = **new** Promise(**function** (resolve, reject){

resolve('hey!');

});

.  
🗃️ Una Promesa puede estar en uno de los siguientes estados:  
.

1. **Pendiente** **pending** → Una promesa inicia en este estado: no cumplida, no rechazada:

* Una promesa inicialmente está pendiente.

1. **Cumplida** **fulfilled** → Significa que la operación se completó satisfactoriamente, .then(va => …)

* Cuando llamamos a resolve entonces la promesa pasa a estar resuelta.
* Cuando una promesa se resuelve entonces se ejecuta la función que pasamos al método .then

1. **Rechazada** **rejected** → significa que la operación falló, .catch(err => …)

* Si llamamos a reject pasa a estar rechazada (obtenemos un error que nos va a indicar la razón del rechazo).
* Si la promesa es rechazada entonces se ejecuta la función que pasamos a .catch  
  .

📝 Ejemplo con then y catch:  
.

* Para probar el código, en el proyecto se crea la carpeta llamada **promise** dentro de la carpeta **src**.
* Se crea el archivo **index.js** en la ruta: src/promise
* El código del ejemplo queda así:

//ejemplo de contar vacas

**const** cows = 15; //valor inicial de vacas

**const** countCows = **new** Promise(**function**(resolve, reject){

//solo si el número de vacas supera 10, se llama al resolve

//de lo contrario: se llama a reject

**if**(cows > 10){

resolve(`We have ${cows} cows on the farm`);

} **else** {

reject("There is no cows on the farm");

}

});

//con solo .then se obtiene el resultado de la promesa de acuerdo a resolve o reject

//con .catch podemos obtener más información de un futuro error que se presente

//con .finally podemos imprimir un mensaje que indica que ya se ejecutó la promesa

countCows.then((result) => {

console.log(result);

}).catch((error) => {

console.log(error);

}).finally(() => console.log('Finally'));

//se usan arrow function () =>

* Para ver el resultado por la consola de VSC, seleccionar el código y dar en Run Code, se puede ir probando cambiando la variable inicial cows  
  .

🏹 Si hay dudas de las funciones ***arrow*** en el enlace hay ejemplos: [aquí](https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Functions/Arrow_functions)

## Crea una función de delay que soporte asincronismo

En este desafío tienes la función delay la cual se espera que un tiempo específico retorne un mensaje

La función deberá recibir dos parámetros:

* time: el tiempo de espera
* message: el mensaje que debe imprimir después del tiempo de espera

La función delay debe retornar una promesa para poderlo usarlo de forma asíncrona.

Nota: Debes usar la función setTimeout con el namespace window para poder monitorear su uso en la ejecución de pruebas, ejemplo:

window.setTimeout(() => {

// code

})

Ejemplo 1:

Input:

delay(2000, "Hello after 2s")

.then((message) => console.log(message))

Output:

// after 2s

"Hello after 2s"

Ejemplo 2:

Input:

delay(3000, "Hello after 3s")

.then((message) => console.log(message))

Output:

// after 3s

"Hello after 3s"

export function delay(time, message) {

  return new Promise((resolve, reject) => {

    window.setTimeout(() => {

      resolve(message);

    }, time);

  });

}

delay(2000, 'Hello!');