

Callbacks y APIs (Parte II)

Trabajar con APIs

Competencias

- Consultar múltiples API utilizando Ajax y promesas para controlar el orden de los llamados.
- Determinar cuáles métodos utilizar para manejar el flujo al trabajar con Promesas.

Introducción

En el mundo de la web cada día navegamos por distintas páginas, ejecutamos aplicaciones, jugamos en los dispositivos, nos logueamos en nuestras aplicaciones favoritas como Instagram, Facebook, Twitter, vemos películas o series en Netflix, entre otros. Cada vez que realizamos una acción en un sitio web que implique ver o manejar datos dinámicos estaremos utilizando APIs para enviar y traer datos desde el servidor.

Por ende, en este capítulo veremos cómo consultar y traer datos desde fuentes externas (APIs), además, que podemos hacer con estos datos que nos entrega la fuente de información externa y cómo manejar los errores en el caso de existir, lo cual es de suma importancia para todo programador frontend o backend, saber como enviar o solicitar información a una API, además de interactuar dependiendo de la respuesta que retorne la API. De aquí la importancia del contenido que aprenderás en esta lectura, el cual, gira en torno al trabajo con fuentes de información externas a nuestro código y te transforma en todo un profesional desarrollando sitios web que utilicen información dinámica de terceros.

¿Qué es una API?

Una API es una Interfaz de Programación de Aplicaciones (*Application Programming Interface*), es un conjunto de subrutinas, funciones, procedimientos, definiciones y protocolos que permiten a aplicaciones comunicarse e integrar funcionalidades, sin importar cómo están construidas. En otras palabras, es una especificación formal sobre cómo un módulo de un software se comunica o interactúa con otro.

Los navegadores también poseen APIs, llamadas WEB APIs y que pueden ser accedidas a través de JavaScript y que permiten realizar muchas tareas en él mismo, así como también permite acceder a los recursos del protocolo HTTP para acceder a APIs externas con la WEB API XMLHttpRequest. Las [WEB APIs](#) no son más que funciones que aportan los navegadores para interactuar con los datos que este proporcione o del entorno en que se encuentre. Por nombrar algunas, tenemos Geolocation, XMLHttpRequest, setTimeout, Canvas, Audio, WEBRTC, Notifications, entre otras.

¿Para qué sirve una API?

Una de las principales funciones de las API es poder facilitar el trabajo a los desarrolladores, ahorrar tiempo y dinero. Por ejemplo, si estás creando una aplicación que es una tienda en línea, no es necesario crear desde cero un sistema de pagos u otro sistema para verificar si hay stock disponible de un producto. Para ello, se puede utilizar una API de servicio de pago ya existente, por ejemplo PayPal, además pedir al distribuidor una API que permita saber la cantidad de productos o inventario que existen.

Tipos de APIs de servicios web

Existen cuatro (4) tipos de APIs que se implementaron para servicios web, estos son: SOAP (Simple Object Access Protocol), XML-RPC, JSON-RPC y REST. Actualmente los más utilizados son SOAP y REST, los cuales se explicaran a continuación.

SOAP (Simple Object Access Protocol)

Como su sigla lo indica es un Protocolo Simple de Acceso a Objetos y sirve para que dos procesos puedan comunicarse intercambiando datos XML. Actualmente se usa para exponer servicios web a través del protocolo HTTP. Soporta sólo XML como formato de datos y se rige por un estricto estándar de mensajes, con reglas establecidas para realizar solicitudes y respuestas. En consecuencia, la estructura de los mensajes SOAP sería así:

```
<?xml version="1.0"?>

<soap:Envelope
  xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
  soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">

  <soap:Header>
  ...
  </soap:Header>

  <soap:Body>
  ...
    <soap:Fault>
    ...
    </soap:Fault>
  </soap:Body>

</soap:Envelope>.
```

El elemento SOAP Envelope es obligatorio y define que el código XML es un mensaje de SOAP. Mientras que el elemento SOAP Header es opcional y contiene información específica de la aplicación, como autenticación, pagos, entre otros. Por otra parte, el elemento SOAP Body es requerido y es el que contendrá toda la información que se necesita para hacer las solicitudes o enviar las respuestas. Mientras que el elemento SOAP Fault es opcional y sirve para indicar mensajes de error. Esto es sólo el principio en cuanto a servicios web con SOAP, y para profundizar en este y otros temas relacionados, puedes visitar la siguiente dirección web: [SOAP](#).

REST (Representational State Transfer)

Representational State Transfer o Transferencia de Estado Representacional, es cualquier interfaz entre sistemas que use HTTP para obtener datos o generar operaciones sobre esos datos en todos los formatos posibles, como XML y JSON. Actualmente se utiliza el mismo patrón, pero sin el uso del estado y éste trabaja sobre el protocolo HTTP, en conjunto con la Interfaz de Programación de Aplicaciones (API) y en el que se contribuye en tener mayor confiabilidad, eficiencia y escalabilidad. A una aplicación se le dice RESTful cuando contiene estas características.

La arquitectura REST define algunas restricciones o características para ser utilizada, como el caso de: Cliente/Servidor, Sin estado, Caché, Sistema de capas, Interfaz uniforme y Código bajo demanda.

Cliente/Servidor

Corresponde a la separación de Cliente/Servidor, cada uno debe manejar sus recursos, el cliente debe manejar la interfaz y el servidor el almacenaje de los datos. Con esto se mejora la portabilidad de las interfaces de usuario y escalabilidad para los componentes de los servidores, ya que no deben preocuparse por lo que pasa en la interfaz de usuario.

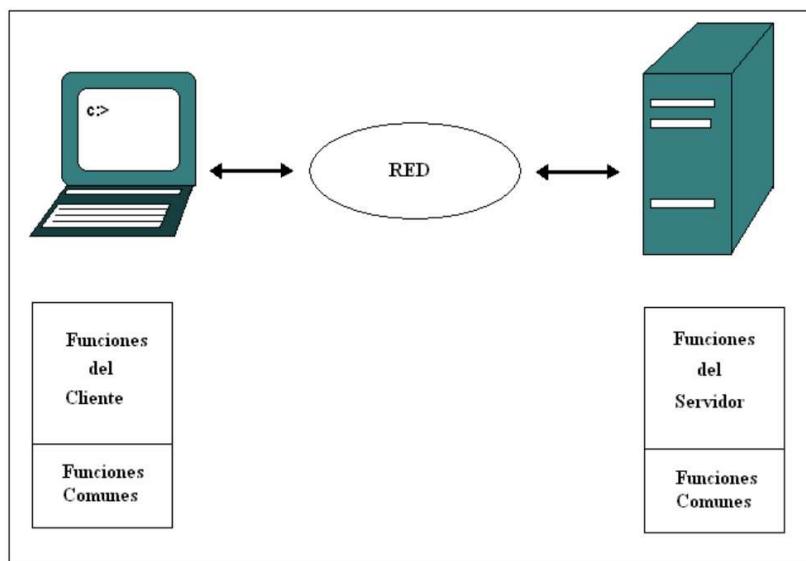


Imagen 1. Ejemplificación de separación cliente/servidor

Fuente: <http://bibing.us.es>

Sin estado

Cada petición que se realice desde el cliente al servidor, deberá contener toda la información necesaria para que el servidor comprenda lo que se está solicitando y no deberá haber ningún dato guardado con anterioridad y en el que pueda revisarlos.

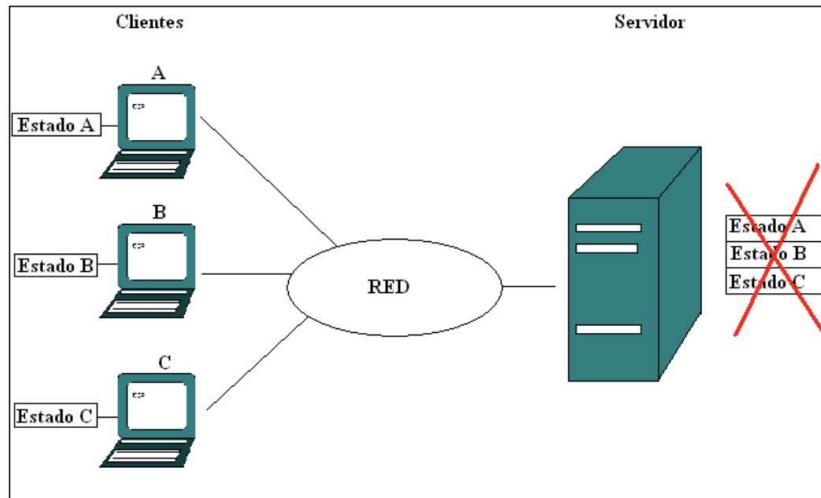


Imagen 2. Ejemplificación de remoción de estados.

Fuente: <http://bibing.us.es>

Caché

Las peticiones que se realicen podrán o no ser “cacheables”, dependerá si el cliente le otorga el permiso al servidor para obtener estas respuestas en caso de hacer las mismas solicitudes.

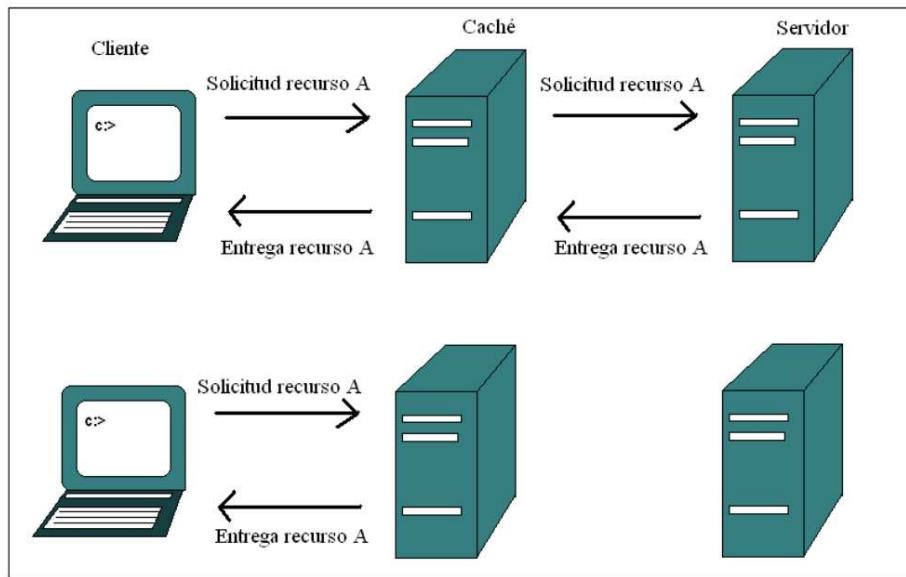


Imagen 3. Ejemplificación de uso de caché.

Fuente: <http://bibing.us.es>

Interfaz uniforme

La idea de utilizar una interfaz uniforme nace de poder estandarizar solicitudes de datos, por lo que no sería necesario hacer solicitudes de tipo createUser, updateProduct, removeTicket, etc. La interfaz de REST tiene un diseño eficiente para el manejo de grandes volúmenes de datos y con esto se optimiza la web.

Para este estándar se establecen algunas restricciones:

- Identificación de recursos
- Manipulación de recursos a través de sus representaciones
- Mensajes auto-descriptivos
- Hipermédios como el motor del estado de la aplicación.

El servidor siempre retornará el estado de la operación y será 200 si fue correcta, en caso de ser cualquier otro número, indicará que algo ocurrió de una forma no esperada, y el resultado de ésta.

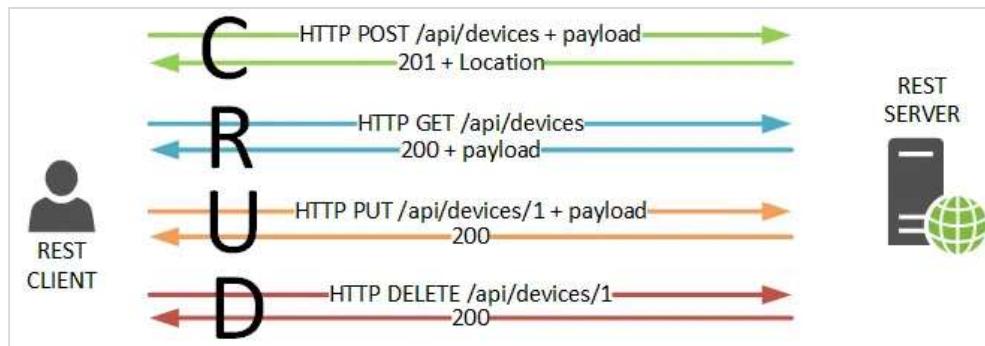


Imagen 4. Operaciones básicas de REST.

Fuente: <https://bit.ly/2TdJa34>

Sistema de capas

La idea de tener un sistema de capas es aumentar la escalabilidad de las aplicaciones, separando por componentes y limitando su comportamiento a lo que debe realizar para complementar todo el sistema.

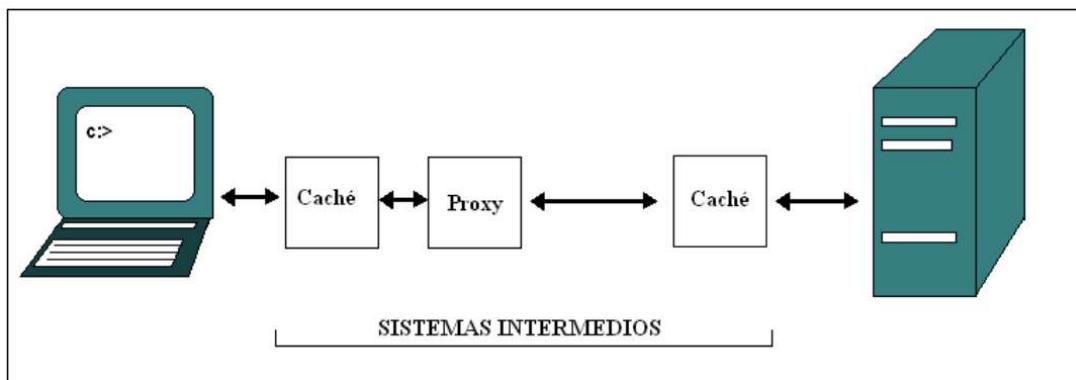


Imagen 5. Ejemplificación de sistema de capas.

Fuente: <http://bibing.us.es>

Código bajo demanda

Este ítem es opcional y permite a los clientes descargar el código y ejecutarlo en forma de scripts. Esto le da al cliente la posibilidad de extender la cantidad de funcionalidades de la aplicación.

API REST

El término API REST significa utilizar una API para acceder a aplicaciones backend, de manera que esa comunicación se realice con los estándares definidos por el estilo de arquitectura REST. En efecto, lo que utilizamos actualmente para manipular los datos es el estándar API REST, que es la combinación de API y REST, haciendo uso del protocolo HTTP y poder realizar el envío y recepción de datos al servidor.

Por otra parte, para realizar el envío y recepción de datos se utilizan las especificaciones más importantes del protocolo HTTP que son GET, POST, PUT, DELETE. Esto se utiliza en un entorno cliente/servidor, en el que el cliente realiza las peticiones y es el server quien procesa las solicitudes. Todas las solicitudes cuentan con una URL, el tipo y los datos que se requiere enviar.

URL (Uniform Resource Locator)

Localizador uniforme de recursos, es una cadena de caracteres que tiene como funcionalidad permitir localizar cada recurso en internet. Cada recurso que existe en internet posee una URL, sitios, documentos, archivos, carpetas

Tipos de solicitudes

- **GET:** Se utiliza para obtener información. Se le puede pasar parámetros a través de la misma URL.
- **POST:** Se usa para agregar nuevos datos. Los datos deben enviarse a través del Body de la solicitud.
- **PUT:** Sirve para actualizar datos y éstos deben enviarse en el body de la solicitud.
- **DELETE:** Nos indica que queremos eliminar algo. Los datos pueden enviarse en la URL o en el body de la solicitud.

Datos

Los datos pueden ser establecidos en el formato según la herramienta que se esté utilizando para realizar las solicitudes a la API. Las APIs REST no son lenguaje dependientes, por lo que se pueden crear con cualquier lenguaje de servidor como JavaScript (Node), PHP, Python o Java, por nombrar algunos, la respuesta puede ser devuelta en cualquier formato, pero por compatibilidad se utiliza JSON o XML.

Consumir APIs

Las APIs se encargan de manejar nuestras peticiones desde el browser y devolvernos los datos que les solicitamos para que podamos mostrarlos en la web. Cuando le indicamos al browser que realice una petición a la API mediante una acción, por ejemplo un click en algún botón o en la carga de una url, entonces realizamos una solicitud mediante cualquier herramienta que permita hacer solicitudes http como axios, fetch, entre otros. En los ejemplos que se desarrollarán a lo largo de este capítulo, se utilizará la WEB API fetch para hacer los request y manejar datos, ya que ofrece una interfaz estándar de uso de Ajax para el desarrollo frontend, a la vez que permite usar promesas y manejar la respuesta a esa promesa con los métodos then y catch. Todo esto nos facilita la organización del código asíncrono en las aplicaciones creadas.

Para consultar y enviar datos utilizaremos la página [JSONPlaceholder](#), que nos permite extraer y enviar datos, además de utilizarlo como API REST para hacer pruebas o prototipos. Los datos devueltos por esta API son sólo de prueba y no datos reales. Así mismo, los recursos que tiene el sitio para utilizar son:

Resources

JSONPlaceholder comes with a set of 6 common resources:

/posts	100 posts
/comments	500 comments
/albums	100 albums
/photos	5000 photos
/todos	200 todos
/users	10 users

Note: resources have relations. For example: **posts** have many **comments**, **albums** have many **photos**, ... see below for routes examples.

Imagen 6: Recursos que posee el sitio JSONPlaceholder.

Fuente: Desafío Latam

Ejercicio guiado: Solicitud a una API

Conectarse a la sección de usuarios (/users) de la [JSONPlaceholder](#) y solicitar los datos que contenga esa dirección, mediante el método fetch y empleando promesas, traer la respuesta y mostrarla en la consola del navegador web.

Para realizar este ejercicio, debemos seguir los siguientes pasos:

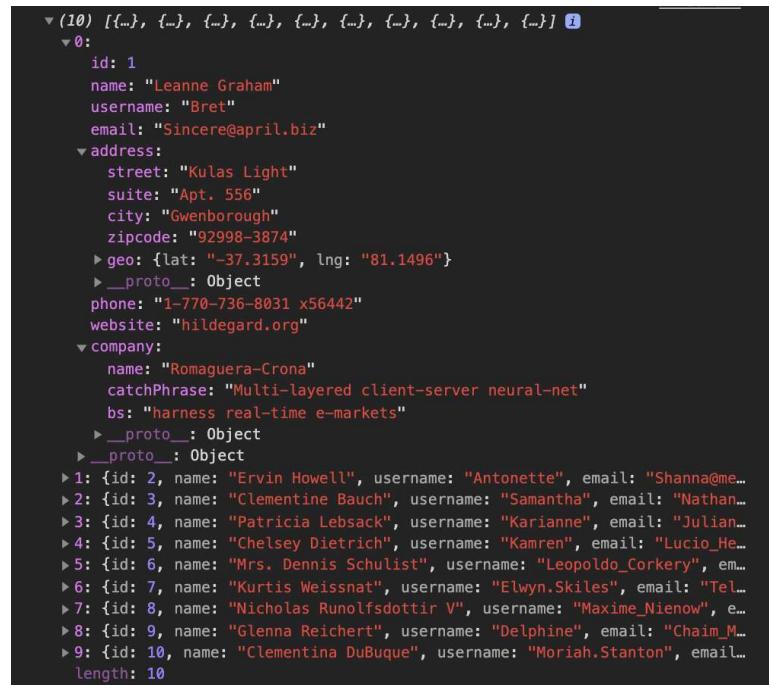
- **Paso 1:** Crear una carpeta en tu lugar de trabajo favorito y dentro de ella crea dos archivos, un index.html y un script.js. Mientras que en el index.html, debes escribir la estructura básica de un documento HTML como se muestra a continuación:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Conexión API</title>
</head>
<body>
  <h4>Conexión API - Fetch</h4>
  <script src="script.js"></script>
</body>
</html>
```

- **Paso 2:** En el archivo script.js, implementando directamente el método fetch, pasamos la URL como parámetro, luego, como este método retorna una promesa, entonces debemos utilizar el then para recibir la respuesta, pero el método fetch tiene una característica en especial y es que todas las respuestas se deben pasar a JSON, implementando el método “[.json\(\)](#)”, al finalizar de recibir y transformar la respuesta al formato JSON, se ejecuta nuevamente una promesa con el then para ahora si mostrar el resultado en la consola del navegador web.

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

- **Paso 3:** Ejecutar el index.html en el navegador web y observar el resultado en la consola, nos debe mostrar un arreglo con 10 objetos. Cada objeto debe contener la información (ficticia) de cada usuario.



```
▼ (10) [{} , {} , {} , {} , {} , {} , {} , {} , {} , {}] ⓘ
  ▼ 0:
    id: 1
    name: "Leanne Graham"
    username: "Bret"
    email: "Sincere@april.biz"
    ▼ address:
      street: "Kulas Light"
      suite: "Apt. 556"
      city: "Gwenborough"
      zipcode: "92998-3874"
      ▶ geo: {lat: "-37.3159", lng: "81.1496"}
      ▶ __proto__: Object
    phone: "1-770-736-8031 x56442"
    website: "hildegard.org"
    ▼ company:
      name: "Romaguera-Crona"
      catchPhrase: "Multi-layered client-server neural-net"
      bs: "harness real-time e-markets"
      ▶ __proto__: Object
      ▶ __proto__: Object
    ▶ 1: {id: 2, name: "Ervin Howell", username: "Antonette", email: "Shanna@me...
    ▶ 2: {id: 3, name: "Clementine Bauch", username: "Samantha", email: "Nathan...
    ▶ 3: {id: 4, name: "Patricia Lebsack", username: "Karianne", email: "Julian...
    ▶ 4: {id: 5, name: "Chelsey Dietrich", username: "Kamren", email: "Lucio_He...
    ▶ 5: {id: 6, name: "Mrs. Dennis Schulist", username: "Leopoldo_Corkery", em...
    ▶ 6: {id: 7, name: "Kurtis Weissnat", username: "Elwyn.Skiles", email: "Tel...
    ▶ 7: {id: 8, name: "Nicholas Runolfsdottir V", username: "Maxime_Nienow", e...
    ▶ 8: {id: 9, name: "Glenna Reichert", username: "Delphine", email: "Chaim_M...
    ▶ 9: {id: 10, name: "Clementina DuBuque", username: "Moriah.Stanton", email...
    length: 10
```

Imagen 7: Resultado de la consola del navegador.

Fuente: Desafío Latam

Una vez obtenidos estos datos ya podemos hacer que se visualicen en nuestra página, mediante grillas, formularios, o cualquier otro componente de HTML y CSS.

Ejercicio propuesto (1)

Utilizando el método fetch y promesas, muestra en la consola del navegador web la información suministrada por la API: <https://swapi.dev/api/people/>.

Solicitud de múltiples APIs en secuencia

Existen ocasiones que para solicitar datos a una API, dependemos de algunos valores que nos retornan otras APIs y para esto necesitamos solicitar datos a múltiples APIs en forma secuencial. Para ello, implementaremos el método fetch pero esta vez trabajado de forma asincrónica con Async / Await.

Ejercicio guiado: Múltiples solicitudes

Se solicita mostrar la información de cada usuario que realizó o escribió comentarios en un Blog. Por ende, se debe traer de la siguiente [API](#) una lista de comentarios (/posts) para luego utilizar de la lista de posts el ID del usuario que realizó el comentario, el atributo donde se encuentra el ID lleva por nombre “userId”. Después de extraer este dato en particular, se debe solicitar toda la información de cada uno de los usuarios que escribieron o realizaron uno de los 100 posts que entrega la API de JSONPlaceholder.

Para realizar este ejemplo, debemos seguir los siguientes pasos:

- **Paso 1:** Crear una carpeta en tu lugar de trabajo favorito y dentro de ella crea dos archivos, un index.html y un script.js. Mientras que en el index.html, debes escribir la estructura básica de un documento HTML como se muestra a continuación:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Conexión API</title>
</head>
<body>
  <h4>Solicitud de múltiples APIs en secuencia</h4>
  <script src="script.js"></script>
</body>
</html>
```

- **Paso 2:** En el archivo script.js, vamos a dividir el código en cinco partes o bloques de código para comprender mejor la estructura de lo que debemos hacer. La primera parte es simplemente declarar una constante con una la dirección “URL” base de la API. Luego, como segunda parte o bloque de código, se debe crear una función denominada “request” que se encargará de solicitar los datos. Seguidamente en el tercer y cuarto bloque de código, se debe crear una función “getPosts” y luego una función “getUser” para crear las urls y consumirlas. Finalmente, en el quinto bloque de código, se debe ejecutar la función creada “getPosts”, para extraer mediante las funciones de orden superior (map), los id de los usuarios.

```
// bloque 1
const baseUrl = 'https://jsonplaceholder.typicode.com';
// bloque 2
const request = async () => {}
// bloque 3
const getPosts = async () => {}
// bloque 4
const getUser = async () => {}
// bloque 5
getPosts().then(() => {})
```

- **Paso 3:** Trabajando ahora en el bloque 2, la función request debe recibir una URL como parámetro para poder consultar mediante el método fetch la API de acuerdo a la URL recibida, logrando así traer y retornar la información. A la consulta de la API y a la transformación de datos a JSON se le agrega la expresión await para completar la función asíncrona.

```
// bloque 2
const request = async (url) => {
  const results = await fetch(url)
  const response = await results.json()
  return response;
}
```

- **Paso 4:** En el tercer bloque de código, se crea o construye la URL para consultar los posts de la API, interpolando a la URL base creada anteriormente el término posts.

```
// bloque 3
const getPosts = async () => {
  const url = `${baseUrl}/posts`;
  return request(url);
}
```

- **Paso 5:** En el cuarto bloque, se creará la dirección URL para poder consultar los usuarios por individual. Por consiguiente, esta función recibe como parámetro el id que se desea consultar, además se construirá la URL interpolando el término "/users/" y el parámetro recibido "id", retornando la URL construida a la función `request` creada anteriormente.

```
// bloque 4
const getUser = async (id) => {
  const url = `${baseUrl}/users/${id}`;
  return request(url);
}
```

- **Paso 6:** Ya el último bloque, se encargará primeramente de ejecutar la función denominada `getPosts`, para poder obtener todos los 100 post que proporciona la API. Luego cuando llegue la información, se deben extraer los ids de los usuarios en un arreglo aparte. Esto se hace con el propósito de recopilar los ids de los usuarios para posteriormente poder solicitar toda la data de un usuarios a la vez. Finalmente con ayuda de `Promise.all` y del `map`, se puede ir pasando el id de cada usuario a la función `getUser` para solicitar a la API la información de cada usuario en particular.

```
// bloque 5
getPosts().then(posts => {
  const usersIds = posts.map(p => p.userId);
  const setOfUsers = new Set(usersIds);
  const users = [...setOfUsers];
  Promise.all(users.map(userId => getUser(userId)))
    .then(response => console.log(response))
});
```

- **Paso 7:** Ejecuta el archivo index.html en el navegador web, luego observa el comportamiento del código en la consola y podrás observar el siguiente resultado.

```
▼ (10) [{} , {} , {} , {} , {} , {} , {} , {} , {} , {}] ⓘ
  ► 0: {id: 1, name: "Leanne Graham", username: "Bret", email: "Sincere@april.biz", ...}
  ► 1: {id: 2, name: "Ervin Howell", username: "Antonette", email: "Shanna@melissa..."}
  ► 2: {id: 3, name: "Clementine Bauch", username: "Samantha", email: "Nathan@yesen..."}
  ► 3: {id: 4, name: "Patricia Lebsack", username: "Karianne", email: "Julianne.OCo..."}
  ► 4: {id: 5, name: "Chelsey Dietrich", username: "Kamren", email: "Lucio_Hettinge..."}
  ► 5: {id: 6, name: "Mrs. Dennis Schulist", username: "Leopoldo_Corkery", email: "..."}
  ► 6: {id: 7, name: "Kurtis Weissnat", username: "Elwyn.Skiles", email: "Telly.Hoe..."}
  ► 7: {id: 8, name: "Nicholas Runolfsdottir V", username: "Maxime_Nienow", email: ...}
  ► 8: {id: 9, name: "Glenna Reichert", username: "Delphine", email: "Chaim_McDermo..."}
  ► 9: {id: 10, name: "Clementina DuBuque", username: "Moriah.Stanton", email: "Rey..."}
  length: 10
```

Imagen 8: Resultado de la consola del navegador

Fuente: Desafío Latam

Ejercicio propuesto (2)

Realizar un programa en JavaScript que permita mostrar todos los post realizados en una página web y luego, de acuerdo al autor de cada post, mostrar la información por individual de cada usuario. Toda la información debe mostrarse en la consola del navegador web. Los post los puedes encontrar en: <http://demo.wp-api.org/wp-json/wp/v2/posts>, y la identificación del autor se encuentra en el atributo con el nombre author. La información de cada usuario la puedes encontrar en: <https://demo.wp-api.org/wp-json/wp/v2/users/> (id).

Solicitud de múltiples APIs a la vez

A veces es necesario realizar múltiples llamadas a la vez para obtener lo que necesitamos mostrar, tal cual como cuando explicamos `Promise.all` en Promesas (Promise) del capítulo anterior.

Ejercicio guiado: Publicaciones de un usuario

Se solicita mostrar la información de un usuario en específico que haya realizado publicaciones en el blog. Por ende, debe traer primeramente de una [API](#) la lista de publicaciones (`/posts?userId=id`) realizados por ese usuario en particular, así como la información exclusiva del usuario que entrega la API de JSONPlaceholder. Por consiguiente, en este ejercicio vamos a realizar una llamada múltiple, a los datos y post del usuario con id "1".

Para realizar este ejemplo, debemos seguir los siguientes pasos:

- **Paso 1:** Crear una carpeta en tu lugar de trabajo favorito y dentro de ella crea dos archivos, un `index.html` y un `script.js`. Mientras que en el `index.html` debes escribir la estructura básica de un documento HTML como se muestra a continuación:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Conexión API</title>
</head>
<body>
  <h4>Solicitud de múltiples APIs a la vez</h4>
  <script src="script.js"></script>
</body>
</html>
```

- **Paso 2:** En el archivo script.js, vamos a dividir el código en cinco bloques para comprender mejor la estructura de lo que debemos hacer. La primera parte es simplemente declarar una constante con la dirección “URL” base de la API. Luego, en el segundo bloque de código, se debe crear una función denominada “request” que se encargará de solicitar los datos. Seguidamente en el tercer y cuarto bloque de código, se debe crear una función “getPosts” y luego una función “getUser” para crear las urls y consumirlas respectivamente desde la API. Finalmente, en el quinto bloque de código, creamos una constante para el id del usuario, la que tendrá el valor de 1 y en la siguiente línea de código se debe ejecutar la promesa con Promise.all, pasando como argumento las dos funciones que retorna la promesa.

```
// bloque 1
const baseUrl = 'https://jsonplaceholder.typicode.com';

// bloque 2
const request = async () => {}

// bloque 3
const getPosts = async () => {}

// bloque 4
const getUser = async () => {}

// bloque 5
const userId = 1;
Promise.all([getUser(userId), getPosts(userId)]).then().catch()
```

- **Paso 3:** Trabajando ahora en el bloque 2, la función request debe recibir una URL como parámetro para poder consultar mediante el método fetch la API de acuerdo a la URL recibida, logrando así traer y retornar la información. A la consulta de la API y a la transformación de datos a JSON se le agrega la expresión await para completar la función asíncrona.

```
// bloque 2
const request = async (url) => {
  const results = await fetch(url)
  const response = await results.json()
  return response;
}
```

- **Paso 4:** En el tercer bloque de código, se crea o construye la URL para consultar los posts de la API pero para un usuario en particular, interpolando a la URL base creada anteriormente el término `posts` más la identificación para especificar que deseamos ubicar los post del usuario con un id en específico, `?userId=id`. El id será la variable que declaramos anteriormente en el bloque 5, en este caso será el número 1, pero puede ser cualquier otro número que corresponda a un usuario.

```
// bloque 3
const getPosts = async (id) => {
  const url = `${baseUrl}/posts?userId=${id}`;
  return request(url);
}
```

- **Paso 5:** En el cuarto bloque, se creará la dirección URL para poder consultar el usuario por individual, es decir, solo la información de ese usuario indicado. Por consiguiente, esta función recibe como parámetro el id que se desea consultar, además se construirá la URL interpolando el término “/users/” y el parámetro recibido “id”, retornando la URL construida a la función `request` creada anteriormente en el segundo bloque.

```
// bloque 4
const getUser = async (id) => {
  const url = `${baseUrl}/users/${id}`;
  return request(url);
}
```

- **Paso 6:** Ya el último bloque, se encargará primeramente de crear la variable donde asignaremos el valor del id del usuario. Luego de ejecutar la instrucción `Promise.all` para pasar como un arreglo todas las promesas que queremos que se ejecuten en simultáneo y asíncronamente regresen un resultado, como es el caso de `getUser(userId)` y `getPosts(userId)`, enviando como argumento el id del usuario y esperando entonces (then) como respuesta la información solicitada. Pero, en el caso de existir un error en el código o en la solicitud a la API, se debe atrapar (catch) ese error y mostrarlo en la consola del navegador.

```
// bloque 5
const userId = 1;
Promise.all([getUser(userId), getPosts(userId)])
    .then(resp => {
        console.log('resp', resp)
    })
    .catch(err => console.log('err', err))
```

- **Paso 7:** Ejecutar el archivo index.html en el navegador web, luego observa el comportamiento

```
VM31710:23
resp
▼ (2) [{...}, Array(10)] ①
  ▼ 0:
    ► address: {street: "Kulas Light", suite: "Apt. 556", city: "Gwenborough", zip...
    ► company: {name: "Romaguera-Crona", catchPhrase: "Multi-layered client-server...
      email: "Sincere@april.biz"
      id: 1
      name: "Leanne Graham"
      phone: "1-770-736-8031 x56442"
      username: "Bret"
      website: "hildegard.org"
    ► __proto__: Object
  ▼ 1: Array(10)
    ► 0: {userId: 1, id: 1, title: "sunt aut facere repellat provident occaecati e...
    ► 1: {userId: 1, id: 2, title: "qui est esse", body: "est rerum tempore vitae...
    ► 2: {userId: 1, id: 3, title: "ea molestias quasi exercitationem repellat qui...
    ► 3: {userId: 1, id: 4, title: "eum et est occaecati", body: "ullam et saepe r...
    ► 4: {userId: 1, id: 5, title: "nesciunt quas odio", body: "repudiandae veniam...
    ► 5: {userId: 1, id: 6, title: "dolorem eum magni eos aperiam quia", body: "ut...
    ► 6: {userId: 1, id: 7, title: "magnam facilis autem", body: "dolore placeat q...
    ► 7: {userId: 1, id: 8, title: "dolorem dolore est ipsam", body: "dignissimos ...
    ► 8: {userId: 1, id: 9, title: "nesciunt iure omnis dolorem tempora et accusan...
    ► 9: {userId: 1, id: 10, title: "optio molestias id quia eum", body: "quo et e...
    length: 10
  ► __proto__: Array(0)
  length: 2
  ► __proto__: Array(0)
```

Imagen 9: Resultado de la consola del navegador.

Fuente: Desafío Latam

Como vemos en los resultados, se obtuvo un arreglo con 2 elementos, el primero contiene los datos del usuario y el segundo contiene un arreglo de 10 elementos con datos de los posts de ese usuario en particular.

Ejercicio propuesto (3)

Realizar un programa en JavaScript que permita mostrar la información exclusiva del personaje de la serie animada Rick and Morty llamado “Rick Sánchez”, además de todos los residentes disponibles en la misma locación o ubicación del personaje. Toda la información debe mostrarse en la consola del navegador web. El personaje lo puedes encontrar en: <https://rickandmortyapi.com/api/character/1>, mientras que la ubicación con todos los residentes del lugar que habita el personaje la puedes ubicar en: <https://rickandmortyapi.com/api/location/1>. El parámetro común en este caso es el id correspondiente al número 1.

Solicitud de múltiples APIs a la vez, utilizando la información

Hasta el momento, ya logramos conectarnos a una API y traer la información, mostrando todo el contenido en la consola del navegador web a la vez pero sin especificar o dividir. Por lo tanto, en esta sección y partiendo del ejemplo anterior, veamos cómo utilizar estos datos que nos retornó la petición para mostrarlos separados y ordenadamente.

Ejercicio guiado: Publicaciones de un usuario, utilizando la información

Partiendo del ejercicio “Publicaciones de un usuario”, donde se hace la solicitud en paralelo a la API con dos promesas, para luego mostrar en la consola del navegador los datos del usuario y los post realizados por el mismo, en este ejercicio debemos mostrar esos datos de forma ordenada e individual.

- **Paso 1:** Para que no modifiquemos el ejemplo anterior, vamos a crear primeramente una carpeta en tu lugar de trabajo y dentro en ella crea dos archivos, un index.html y un script.js. Mientras que en el index.html debes escribir la estructura básica de un documento HTML como se muestra a continuación:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Conexión API</title>
</head>
<body>
  <h4>Solicitud de múltiples APIs a la vez</h4>
  <script src="script.js"></script>
</body>
</html>
```

- **Paso 2:** En el archivo script.js, vamos a copiar todo lo realizado en el ejercicio anteriormente, hasta la parte final donde se utiliza el Promise.all para ejecutar las promesas en paralelo y recibir la respuesta asíncronamente.

```
const baseUrl = 'https://jsonplaceholder.typicode.com';
const request = async (url) => {
  const results = await fetch(url)
  const response = await results.json()
  return response;
}
const getUser = async (id) => {
  const url = `${baseUrl}/users/${id}`;
  return request(url);
}
const getPosts = async (id) => {
  const url = `${baseUrl}/posts?userId=${id}`;
  return request(url);
}
const userId = 1;
Promise.all([getUser(userId), getPosts(userId)])
  .then(resp => {
    console.log('resp', resp)
  })
  .catch(err => console.log('err', err))
```

- **Paso 3:** Ahora dentro del último bloque, en el Promise.all, después de recibir y mostrar la respuesta en la consola del navegador, lo primero que vamos a hacer es dividir la información en dos partes, como el resultado viene en un arreglo con espacios, entonces creamos una variable para los post y otra para el usuario, pasando el elemento correspondiente del arreglo a cada nueva variable. Esto se hace para poder obtener directamente el objeto en el caso del usuario que contiene toda la información.

```
Promise.all([getUser(userId), getPosts(userId)])
  .then(resp => {
    console.log('resp', resp)
    const posts = resp[1];
    const user = resp[0];
  })
  .catch(err => console.log('err', err))
```

- **Paso 4:** Ahora ya podemos manipular los datos, es decir, obtener el nombre, dirección, ciudad o cualquiera de los datos que aparece en el objeto con los datos del usuario, y también los datos de los posts que ha publicado, como el title y el body. En el caso de los datos del post, como nos encontramos con un arreglo se debe implementar una estructura repetitiva que itere ese arreglo para poder acceder a los objetos que contiene cada espacio del arreglo, en este caso utilizaremos un forEach para iterar el arreglo, podemos mostrar el título y el cuerpo de cada post realizado por el usuario.

```
Promise.all([getUser(userId), getPosts(userId)])
.then(resp => {
  const posts = resp[1];
  const user = resp[0];
  const name = user.name;
  const address = user.address;
  const email = user.email;
  const city = address.city;

  console.log('name', name);
  console.log('address', address.suite, address.street);
  console.log('geo', `${address.geo.lat},${address.geo.lng}`);
  console.log('email', email);
  console.log('city', city);
  posts.forEach(element => {
    const title = element.title;
    const body = element.body;
    console.log('title', title);
    console.log('body', body);
  });
})
.catch(err => console.log('err', err))
```

- **Paso 5:** Finalmente, solo queda por ejecutar el archivo index.html en el navegador web, luego acceder a la consola para poder visualizar todos los resultados que indicamos para mostrar en el paso anterior. Quedando como resultado final:

```
name Leanne Graham
address Apt. 556 Kulas Light
geo -37.3159,81.1496
email Sincere@april.biz
city Gwenborough
<empty string>
title sunt aut facere repellat provident occaecati excepturi optio reprehenderit
body quia et suscipit
suscipit recusandae consequuntur expedita et cum
reprehenderit molestiae ut ut quas totam
nostrum rerum est autem sunt rem eveniet architecto
title qui est esse
body est rerum tempore vitae
sequi sint nihil reprehenderit dolor beatae ea dolores neque
fugiat blanditiis voluptate porro vel nihil molestiae ut reiciendis
qui aperiam non debitis possimus qui neque nisi nulla
title ea molestias quasi exercitationem repellat qui ipsa sit aut
body et iusto sed quo iure
voluptatem occaecati omnis eligendi aut ad
voluptatem doloribus vel accusantium quis pariatur
molestiae porro eius odio et labore et velit aut
title eum et est occaecati
```

Imagen 10: Resultado de la consola del navegador.

Fuente: Desafío Latam

Ejercicio propuesto (4)

Realizar un programa con JavaScript que utiliza la [API mindicador](#) para obtener todos los indicadores económicos. Para ello, debes crear las funciones para obtener los valores de: Dólar, Euro UF, UTM, IPC, agregándole al final de la url el código que quieras obtener, estos son: dolar, euro, uf, utm e ipc. Por consiguiente, se debe listar la fecha y los valores de la propiedad "serie" y utilizar promesas para obtener todos los valores a la vez. Ejemplo de la URL: <https://mindicador.cl/api/dolar>, <https://mindicador.cl/api/ipc>

Manejo de errores

Competencias

- Detallar los distintos tipos de excepciones que ocurren al ejecutar un código en JavaScript para ubicar errores rápidamente.
- Implementar excepciones con Throw / Reject para generar errores sin restricción del tipo de excepción.
- Capturar errores con catch para mostrar mensajes personalizados de errores.

Introducción

JavaScript como muchos otros lenguajes tiene funciones para el manejo de errores en nuestras aplicaciones. Estos errores pueden ser de muchos tipos, como por ejemplo solicitar datos a una API y que ésta nos devuelva un JSON que está mal formado, o que se está generando una división y el dividendo está en 0, variables no definidas, tratar de acceder a una propiedad inexistente en un objeto o cualquier imprevisto que ocurra. Es por esto que necesitamos tener en nuestras aplicaciones manejadores de errores. Debido a esto, en este capítulo se aprenderán cuales son los tipos de errores y su utilización, además de los tipos derivados de la clase Error, como son RangeError, ReferenceError, SyntaxError, TypeError, EvalError y URIError.

Finalmente, aprenderemos sobre el uso del método catch para manejar las excepciones de las promesas y utilizaremos el bloque try..catch para manejar el resto de los errores. Por lo tanto, la importancia de este capítulo radica en aprender a manejar los errores, identificarlos y saber que puedes hacer con ellos, permitiendo generar distintas respuestas personalizadas a los usuarios de tu sitio web.

Excepciones

Las excepciones son imprevistos que ocurren durante la ejecución de una aplicación y que provocan que éstas no funcionen de la manera que se espera. También podemos establecer nuestros propios errores de acuerdo a alguna funcionalidad que deseamos que la provoque.

Necesitamos poder controlar cualquier tipo de error que ocurra durante la ejecución de nuestro código y en caso de no ser así, podemos encontrarnos con excepciones no controladas como ocurrió en el ejemplo de promesa rechazada de la Lectura Callbacks y APIs (Parte I), capítulo Promesas (Promise).

Para el caso de una promesa, cuando ésta se rechaza, automáticamente se vuelve un error y debemos continuar la ejecución con **.catch**. En otros casos de excepción vamos a crear y poner nuestro código dentro del bloque **try/catch/finally** para que nuestras aplicaciones no interrumpan su flujo normal. Cuando ocurre alguna excepción en la ejecución de nuestro código, comienza la propagación del error a través de la pila de llamadas hasta encontrar un controlador del error que ha ocurrido o romper la aplicación si llega al nivel superior de la pila.

Tipos de errores

JavaScript posee la clase Error, que es el tipo predefinido por el lenguaje para lanzar las excepciones. Existen también otros tipos derivados de esta clase y algunos de ellos son RangeError, ReferenceError, SyntaxError, TypeError, EvalError y URIError.

- **Error:** Permite establecer un mensaje de error personalizado.

```
try {  
    throw new Error("Ups! Ha ocurrido un error");  
} catch (e) {  
    console.log(e.name + ": " + e.message);  
}
```

Lo que nos retorna el error, es el nombre (name) del tipo de error y el mensaje (message) que le pasamos de argumento.

```
Error: Ups! Ha ocurrido un error
```

- **RangeError:** Ocurre cuando un número está fuera del rango permitido por el lenguaje.

```
let a = []; a.length = a.length - 1;
```

Lo que se muestra en este código es la creación de un arreglo al cual tratamos de modificar el largo restándole 1, lo que nos genera una excepción, ya que el largo de un arreglo no puede ser menor a 0.

Uncaught RangeError: invalid array length

- **ReferenceError:** Ocurre cuando se hace referencia a variables no declaradas.

```
const x = y;
```

En este caso estamos declarando la constante **x** y asignándole la variable **y**, que aún no ha sido declarada, por lo que nos genera una excepción.

Uncaught ReferenceError: y is not defined

- **SyntaxError:** Ocurre cuando hay un error de sintaxis en nuestro código.

```
funcction getValue(){  
    return 2;  
};  
getValue();
```

Aquí vemos claramente que hay un error al escribir la palabra **function**, lo que genera de inmediato un error de sintaxis, ya que dentro del lenguaje JavaScript no existe la palabra reservada **funcction**.

Uncaught SyntaxError: unexpected token: identifier

- **TypeError:** Ocurre cuando un valor no es del tipo esperado.

```
const x = {};
x.y();
```

Definimos el objeto “x” e intentamos acceder a la función “y” que no está declarada en el objeto, esto provoca un error de tipo de dato esperado, indicándonos qué “y” no es de tipo función.

Uncaught TypeError: x.y is not a function

- **EvalError:** Ocurre cuando hay un error al evaluar una expresión con la función **eval**. Hay que mencionar que evaluar expresiones con esta función es una mala práctica, por lo que no se debería usar. Si realmente necesita usarla, entonces hay algún problema con la lógica de lo que está desarrollando. Esta clase ya no es parte del estándar ECMAScript, pero sigue activa para retrocompatibilidad. Por ende, la recomendación general es no usarla y partiendo de esto no se explicará ningún ejemplo en código.
- **URIError:** Ocurre cuando se codifica o decodifica una URL utilizando las funciones **encodeURI**, **decodeURI**, **encodeURIComponent** o **decodeURIComponent**.

```
decodeURIComponent("http://%google.com");
```

Vemos aquí que la función **decodeURIComponent** no puede decodificar la URL que fue pasada por argumento y genera la excepción.

Uncaught URIError: malformed URI sequence

Vistos los diferentes tipos de errores que pueden ocurrir a lo largo de la programación de nuestras aplicaciones web en la consola del navegador, es momento de trabajar con las posibles respuestas que podemos generar cuando ocurran estas situaciones. Por ende, para este documento emplearemos el uso de la clase **Error** y utilizaremos el comando **Throw** para emitir una excepción cuando lo requiera nuestro código o aplicación.

Ejercicio propuesto (5)

Del siguiente código y sin ejecutar en el navegador, ¿Cuál crees que sería el error que se mostrará en la consola? Verifica ahora tu respuesta ejecutando el código directamente en la consola de navegador web.

```
var a = new Array(4294967295);
var b = new Array(-1);

var num = 2.555555;
document.writeln(num.toExponential(4));
document.writeln(num.toExponential(-2));

num = 2.9999;
document.writeln(num.toFixed(2));
document.writeln(num.toFixed(105));

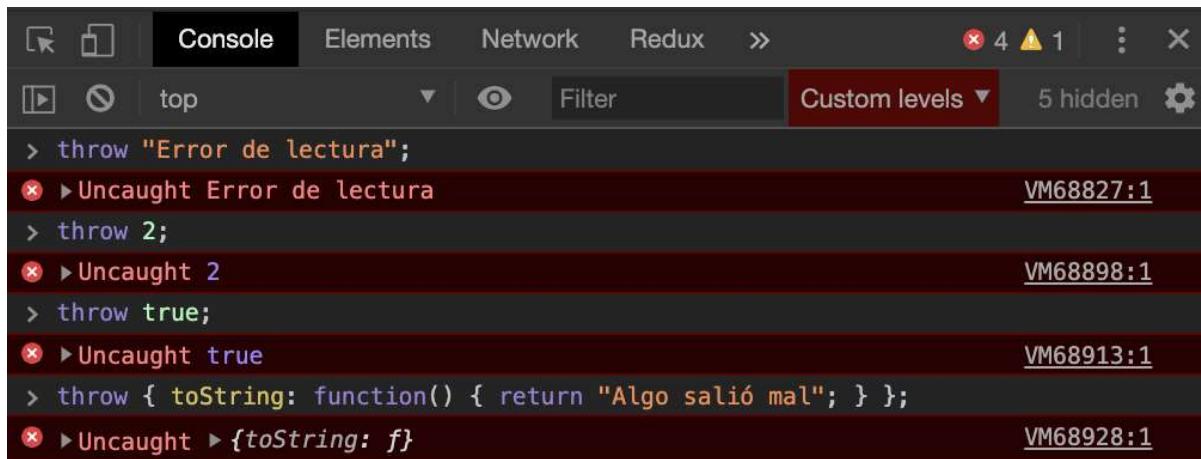
num = 2.3456;
document.writeln(num.toPrecision(1));
document.writeln(num.toPrecision(0));
```

Throw

Trabajando en nuestro código siempre es necesario validar el funcionamiento correcto e incorrecto de lo que estamos haciendo, y para hacerlo podemos crear nuestros propios errores para que después podamos controlar qué fue lo que ocurrió en un determinado punto de la aplicación.

Este comando permite enviar al navegador una excepción tal cual como si de una real se tratase. Veamos su funcionamiento y cómo crear nuestros propios tipos de errores, a través del siguiente código:

```
throw "Error de lectura"; // Tipo string
throw 2; // Tipo número
throw true; // Tipo booleano
throw { toString: function() { return "Algo salió mal"; } }; // tipo objeto
```



The screenshot shows a browser's developer tools console tab labeled 'Console'. It displays several error messages with their respective file paths and line numbers:

- > throw "Error de lectura"; VM68827:1
- ✖ ▶ Uncaught Error de lectura VM68827:1
- > throw 2; VM68898:1
- ✖ ▶ Uncaught 2 VM68898:1
- > throw true; VM68913:1
- ✖ ▶ Uncaught true VM68913:1
- > throw { toString: function() { return "Algo salió mal"; } }; VM68928:1
- ✖ ▶ Uncaught ▶ {toString: f} VM68928:1

Imagen 11. Resultado de la consola del navegador.

Fuente: Desafío Latam

Como se puede ver, el uso de **throw** permite generar errores sin restricción del tipo de excepción, eso permite que podamos definir nuestras propias excepciones y controlar de mejor manera los posibles errores.

Además, podemos hacer uso de los tipos predeterminados del lenguaje para indicar los tipos de errores ocurridos, por ejemplo, podemos crear un error de tipo **Error** o **SyntaxError**.

```
throw new Error("Algo salió mal.");
throw new SyntaxError("Está mal escrito el DNI.");
```

Para poder utilizar **throw** correctamente no basta con ponerla en una función y ya, o en una función con múltiples condiciones, veamos un ejemplo.

```
const showError = () => {
    throw 'Ha ocurrido un error';
}

showError();
```

Efectivamente nos sirve para mostrar un error, pero el error en sí no está controlado.

```
Uncaught Ha ocurrido un error
```

Para que los mensajes de error estén controlados por la aplicación, debemos utilizar el bloque *Try...Catch...Finally*, que veremos más adelante y sólo ahí recién estará controlada la excepción.

Ejercicio guiado: Excepciones personalizadas con Throw

Simular un error levantando una excepción con “Throw” que indique que “Ha ocurrido un error” dentro del bloque try para ser atrapada con el bloque catch. Para ello, sigue los siguientes pasos:

- **Paso 1:** Abre tu navegador web preferido y ve a la consola. Seguidamente sobre ella, vamos a copiar un código, en este caso, como vamos a simular un error y probar la estructura Try...Catch, crea una constante con el nombre de `showError`, sobre ella una función sin parámetros mediante ES6. Dentro de la función, establecemos el primer bloque “try” para ejecutar y forzar un error personalizable mediante la sentencia `throw` con un mensaje: ‘`Ha ocurrido un error`’.

```
const showError = () => {
  try {
    throw 'Ha ocurrido un error';
  }
}
```

- **Paso 2:** Ahora agregamos el bloque “catch” para atrapar el error personalizado creado en el bloque try. Recibiendo como parámetro con la letra “e” y mostrando en consola el parámetro.

```
const showError = () => {
  try {
    throw 'Ha ocurrido un error';
  }
  catch(e) {
    console.log(e)
  }
}
showError();
```

- **Paso 3:** Finalmente hacemos el llamado a la función `showError()`, y observamos el resultado con un correcto manejo del error.

```
Ha ocurrido un error
```

Ejercicio propuesto (6)

Realizar una función que genere un error en el bloque del “try” indicando que las variables no están declaradas. Para atrapar el error se debe implementar el “catch”.

Método catch

Para el manejo de errores en las promesas utilizaremos el método **catch**, en el que viene como argumento el detalle de la excepción retornada por ésta, es decir, el detalle del error ocurrido durante el proceso. Recordemos que debemos devolver la promesa rechazada (`reject`) para que podamos utilizar el método de manejo de error.

Ejercicio guiado: Método catch

Crear una función que retorne una promesa, pero en este caso forzaremos el error, rechazando la promesa y creando un error.

Ahora sigamos los pasos a continuación para realizar este ejemplo.

- **Paso 1:** Abre tu navegador web preferido y ve a la consola. Seguidamente, vamos a copiar un código directamente sobre la consola del navegador web, en este caso, como vamos a rechazar una promesa y probar la estructura `Try...Catch`, crea una función con el nombre de `showError`, sobre ella retornamos primeramente una nueva promesa, utilizando como parámetros (`resolve` y `reject`), luego dentro de la promesa utiliza el `reject` para retornar un nuevo error con el mensaje `"Ha ocurrido un error!"`.

```
function showError () {  
    return new Promise((resolve, reject) => {  
        reject(new Error("Ha ocurrido un error!"));  
    })  
}
```

- **Paso 2:** Finalmente, se debe llamar a la `showError` y como esta función retorna una promesa, debemos prepararnos para recibir la respuesta, ya sea correcta (`resolve`) o regrese un error por cualquier situación (`reject`). En el caso de retornar una respuesta satisfactoria el `then` la recibiría y mostraría por la consola. Pero, como ya sabemos que estamos rechazando la promesa, es decir, está retornando un error, entonces el `catch` se encargará de atraparla y mostrar el mensaje que trae el objeto.

```
showError()
.then(resolve =>{
    console.log(resolve)
})
.catch(err => {
    console.log(err.message)
})
```

- **Paso 3:** Ejecutar el código anterior en la consola del navegador web, el resultado mostrado sería el siguiente:

```
Ha ocurrido un error!
Promise { <state>: "pending" }
```

Ejercicio propuesto (7)

Desarrollar un programa con JavaScript que permita indicar a un alumno, si su respuesta de verdadero o falso estuvo acertada o no. Utiliza promesas, `then...catch` y funciones para generar el código.

Try, Catch, Finally

Es un bloque de control en el que podemos controlar las excepciones imprevistas y en la que podemos determinar qué realizar al momento de un error y si debemos continuar con la ejecución de nuestra aplicación o debemos detener el flujo.

```
try {  
}  
} catch (error) {  
}  
} finally {  
}
```

- El bloque **try** se encarga de ejecutar todas las instrucciones que posea y evalúa en cada una si se ejecutó de forma correcta.
- El bloque **catch** permite obtener cuál fue el error que se produjo en el bloque try, en el que podemos acceder a la información del error mediante una variable.
- El bloque **finally** permite que siempre se puedan ejecutar sentencias posterior a try y catch, habiendo evaluado de forma correcta el try o si es que hubiera algún error en catch, sí o sí ejecutará lo que está en este bloque.

El único bloque obligatorio es try, pero debe estar acompañado por uno de los dos otros bloques. Puede ser try...catch o try...finally. En el caso de try...finally no se controlarán las excepciones como se harían con try...catch.

Ejercicio guiado: Try... catch.. finally

Mostrar tres valores de una variable que contiene distintos tipos de datos. Los dos primeros valores a mostrar deben estar en los datos de la variable indicada, mientras que el tercero no debe existir. Se debe utilizar los bloques de control try...catch...finally.

- **Paso 1:** Abre tu navegador web preferido y ve a la consola. Luego, vamos a declarar una constante denominada "json" con los datos de un post ficticio.

```
const json = {
  "articles": [
    {
      "id": "1",
      "title": "Mi primer artículo",
      "body": "Nunca había escrito un artículo."
    },
    {
      "id": "2",
      "title": "Mi segundo artículo",
      "body": "Este es mi segundo artículo."
    }
}
```

- **Paso 2:** Seguidamente sobre la misma consola del navegador web, trabajamos con la expresión `try` para mostrar con `console.log` los títulos de los artículos que se encuentran disponibles en la posición "0, 1 y 2" del arreglo `"articles"`. En este caso estaremos generando un error voluntariamente ya que la posición dos [2]: `json.articles[2].title` del arreglo no existe. Este error nos servirá para trabajar con la estructura `catch..`

```
try {
  console.log(json.articles[0].title);
  console.log(json.articles[1].title);
  console.log(json.articles[2].title);
}
```

- **Paso 3:** Ahora para poder atrapar los errores, vamos a crear la estructura `catch`, dentro de ella mostraremos en la consola el nombre del error y el mensaje que este genera. Recuerda que el error es atrapado como un objeto al cual podemos acceder a sus propiedades.

```
catch(e) {  
    console.log('Errores');  
    console.log(e.name, e.message);  
}
```

- **Paso 4:** Finalmente, agregamos el `finally` para mostrar el último mensaje por defecto cuando todo se cumpla.

```
finally {  
    console.log('Finalizado el try...catch. Este mensaje aparece  
siempre.');
```

- **Paso 5:** Ejecutar el código en la consola del navegador web, encontraremos el siguiente resultado.

Mi primer artículo	debugger eval code:13:13
Mi segundo artículo	debugger eval code:14:13
Errores	debugger eval code:18:13
TypeError json.articles[2] is undefined	debugger eval code:19:13
Finalizado el try...catch. Este mensaje aparece siempre.	debugger eval code:22:13

Imagen 12. Resultado de la consola del navegador.

Fuente: Desafío Latam

La forma en que se ejecutan los bloques lo podemos visualizar en el siguiente diagrama de flujo.

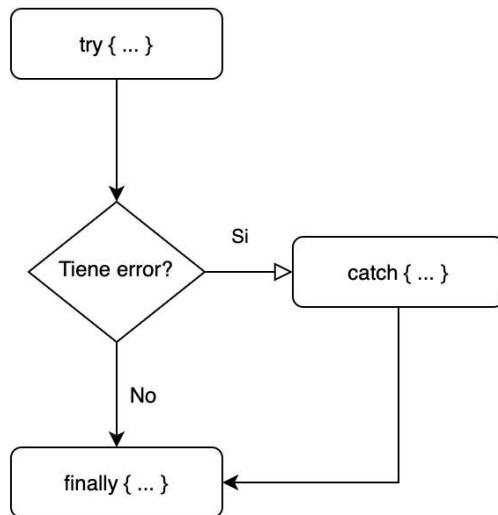


Imagen 13. Diagrama de flujo de try, catch, finally.

Fuente: Desafío Latam

Ejercicio propuesto (8)

De los datos indicados a continuación. Muestra por consola los valores del nombre y apellido de cada persona, incluyendo una tercera persona ficticia, utilizando la estructura try...catch...finally. Indicando el nombre del error y el mensaje en una ventana con la instrucción alert.

```
const json = {
  "persona": [
    {
      "id": "1",
      "nombre": "Juan",
      "apellido": "Romero."
    },
    {
      "id": "2",
      "nombre": "Jocelyn",
      "apellido": "Perez."
    }
  ]
}
```

Buenas prácticas en el manejo de errores

El manejo de los errores en nuestras aplicaciones debe ser aplicado de manera correcta y para esto hay que tener algunas cosas bien claras y conocer cuáles serían las mejores prácticas a utilizar para esto.

Una de las mejores formas de manejar los errores es utilizar el bloque **try...catch...finally** en el que podemos capturar errores lógicos y de ejecución. El uso de **throw** en combinación con **try..catch** para generar excepciones propias y controlar la aplicación de mejor manera. Mientras que el método **onerror** del objeto **window** permite capturar los errores en tiempo de ejecución. También se puede utilizar **onerror** en el elemento **img** de html para capturar el error cuando no existe la imagen cargada. Por ende, al utilizar **try...catch**, podemos obtener la lista completa del stack en el bloque catch al momento de generarse el error.

Ejercicio guiado: Depurando errores

Acceder al stack desde el bloque catch, mediante la construcción de tres funciones, donde la primera función pase valores a una segunda función que no los reciba y a su vez llame a una tercera función que tenga la estructura try...catch, forzando un error desde el try para acceder al stack de ese error en el catch.

- **Paso 1:** Abre tu navegador web preferido y ve a la consola. Seguidamente, vamos a declarar crear la primera función llamada `trace()`, que contenga el bloque `try...catch`. Para generar y atrapar el error.

```
function trace(){
  try {
    throw new Error('miErrorpersonalizado');
  }
  catch(e) {
    console.log(e.stack);
  }
}
```

- **Paso 2:** Seguidamente, creamos la segunda función que no recibirá ningún parámetro y llamará a la función `trace` para ejecutar la estructura `try...catch`.

```
function b(){
  trace();
}
```

- **Paso 3:** Luego, creamos la función que llamará a la función creada anteriormente b(), pasando algunos parámetros, creando así la cadena de llamados.

```
function a(){
    b(1, 'texto', undefined, {});
}
```

- **Paso 4:** Al ejecutar todo este código en la consola del navegador web, el resultado obtenido será:

```
a();
```

- **Paso 5:** Queda por agregar al final el llamado a esta primera función denominada a(), para que se ejecuten todas las funciones en cadena y se genere el resultado en la consola del navegador web.

```
Error: miErrorPersonalizado
at trace (<anonymous>:3:15)
at b (<anonymous>:10:5)
at a (<anonymous>:14:5)
at <anonymous>:17:1
```

VM15889:5

Imagen 14. Resultado de la consola del navegador

Fuente: Desafío Latam

Es necesario tener algunas consideraciones para controlar errores en funciones que se ejecutan de manera asíncrona:

- Es conveniente realizarlo con promesas, ya que en caso de haber error, se puede rechazar la promesa al finalizar el proceso.
- Funciona de la misma forma en una “función asíncrona” (declarada con async) y maneja los errores con el método catch de la promesa retornada.
- Lo que no se puede realizar, es manejar los errores con try..catch en funciones asíncronas, ya que éste es síncrono y por lo tanto jamás capturará el error, puede parecer como que no ocurriese nada.

Ejercicio propuesto (9)

Crear la función **whereToSee** a la que se pueda pasar el argumento **direction**, que tiene 2 valores posibles, **left** y **right**. La función debe retornar siempre lo opuesto a lo que se envía. Se deben manejar los errores en caso de que se envíe a la función otro valor que no sean los posibles, retornando el error con la clase Error.

Ejercicio propuesto (10)

Crea una función con el nombre de **ask** y luego crea dos (2) variables denominadas "num1 y num2", a ambas variables se le deben asignar la función **prompt** para solicitar un número cualquiera. Posteriormente se debe validar que realmente se ingresaron los dos números y devolver la suma de ambos. Pero, si no se puede realizar la suma porque el usuario no ingresó ambos números se debe devolver un error mediante el levantamiento de una excepción con **throw**.

Felicidades, llegaste al final de la lectura, pero aún quedan muchas cosas por aprender sobre JavaScript, como el caso de las distintas ramas que podemos seguir en el mundo de la programación con este maravilloso lenguaje, al igual de las dos vertientes de la programación como lo es el Frontend y el Backend, que frameworks intervienen, como interactúan estos dos mundos en JavaScript. Por ende, para entender cómo se desarrollan las webs actuales y cuáles serían los conocimientos que deben tener y quienes son los que se encargan de cada parte de ellas, como las diferencias entre un Frontend y un Backend, que son los que se encargan de la parte visual y manejo del servidor. Al igual que las herramientas y frameworks, y cómo se asocian con cada perfil. te invitamos a revisar en Material Complementario los siguientes documentos:

- Material de Apoyo Lectura - Frontend y Backend
- Material de Apoyo Lectura - Frameworks y Herramientas

Resumen

A lo largo de esta lectura cubrimos:

- Consultar múltiples API utilizando Ajax y promesas para controlar el orden de los llamados.
- Determinar cuáles métodos utilizar para manejar el flujo al trabajar con Promesas.
- Detallar los distintos tipos de excepciones que ocurren al ejecutar un código en JavaScript para ubicar errores rápidamente.
- Implementar excepciones con Throw / Reject para generar errores sin restricción del tipo de excepción.
- Capturar errores con catch para mostrar mensajes personalizados de errores.

Solución de los ejercicios propuestos

1. Utilizando el método fetch y promesas, muestra en la consola del navegador web la información suministrada por la API: <https://swapi.dev/api/people/>.

```
fetch('https://swapi.dev/api/people/')
  .then(response => response.json())
  .then(json => console.log(json.results))
```

2. Realizar un programa en JavaScript que permita mostrar todos los post realizados en una página web y luego, de acuerdo al autor de cada post, mostrar la información por individual de cada usuario. Toda la información debe mostrarse en la consola del navegador web. Los post los puedes encontrar en: <http://demo.wp-api.org/wp-json/wp/v2/posts>, y la identificación del autor se encuentra en el atributo con el nombre author. La información de cada usuario la puedes encontrar en: <https://demo.wp-api.org/wp-json/wp/v2/users/> (id).

```
const baseUrl = 'http://demo.wp-api.org/wp-json/wp/v2';

const request = async (url) => {
  const results = await fetch(url);
  const response = await results.json();
  return response;
}

const getPosts = async () => {
  const url = `${baseUrl}/posts`;
  return request(url);
}

const getUser = async (id) => {
  const url = `${baseUrl}/users/${id}`;
  return request(url);
}

getPosts().then(posts => {
  const autorIds = posts.map(p => p.author);
  const setOfUsers = new Set(autorIds);
  const users = [...setOfUsers];
  Promise.all(users.map(userId => getUser(userId)))
})
```

```
.then(response => console.log(response))  
});
```

3. Realizar un programa en JavaScript que permita mostrar la información exclusiva del personaje de la serie animada Rick and Morty llamado “Rick Sánchez”, además de todos los residentes disponibles en la misma locación o ubicación del personaje. Toda la información debe mostrarse en la consola del navegador web. El personaje lo puedes encontrar en: <https://rickandmortyapi.com/api/character/1>, mientras que la ubicación con todos los residentes del lugar que habita el personaje la puedes ubicar en: <https://rickandmortyapi.com/api/location/1>. El parámetro común en este caso es el id correspondiente al número 1.

```
const baseUrl = 'https://rickandmortyapi.com/api';  
  
const request = async (url) => {  
    const results = await fetch(url);  
    const response = await results.json();  
    return response;  
}  
  
const getLocations = async (id) => {  
    const url = `${baseUrl}/location/${id}`;  
    return request(url);  
}  
  
const getUser = async (id) => {  
    const url = `${baseUrl}/character/${id}`;  
    return request(url);  
}  
  
const userId = 1;  
Promise.all([getUser(userId), getLocations(userId)])  
    .then(resp => {  
        console.log('resp', resp)  
    })  
    .catch(err => console.log('err', err))
```

4. Realizar un programa con JavaScript que utiliza la [API mindicador](#) para obtener todos los indicadores económicos. Para ello, debes crear las funciones para obtener los valores de: Dólar, Euro UF, UTM, IPC, agregándole al final de la url el código que quieras obtener, estos son: dolar, euro, uf, utm e ipc. Por consiguiente, se debe listar la fecha y los valores de la propiedad "serie" y utilizar promesas para obtener todos los valores a la vez. Ejemplo de la URL: <https://mindicador.cl/api/dolar>, <https://mindicador.cl/api/ipc>

```
const baseUrl = 'https://mindicador.cl/api';

const request = async (url) => {
    const results = await fetch(url);
    const response = await results.json();
    return response;
}

const getDolar = async (indicador) => {
    const url = `${baseUrl}/${indicador}`;
    return request(url);
}

const getEuro = async (indicador) => {
    const url = `${baseUrl}/${indicador}`;
    return request(url);
}

const getUf = async (indicador) => {
    const url = `${baseUrl}/${indicador}`;
    return request(url);
}

const getUtm = async (indicador) => {
    const url = `${baseUrl}/${indicador}`;
    return request(url);
}

const getIpc = async (indicador) => {
    const url = `${baseUrl}/${indicador}`;
    return request(url);
}

const indicadores = ['dolar', 'euro', 'uf', 'ipc', 'utm'];
Promise.all([getDolar(indicadores[0]), getEuro(indicadores[1]),
getUf(indicadores[2]), getUtm(indicadores[3]), getIpc(indicadores[4])])
    .then(resp => {
        console.log(resp)
        resp.forEach((element) => {
            console.log(`Moneda: ${element.codigo}, Fecha: ${element.fecha}, Valor: ${element.serie}`)
        })
    })

```

```
 ${element.serie[0].fecha}, Valor: ${element.serie[0].valor} )  
});  
}  
.catch(err => console.log('err', err))
```

5. Del siguiente código y sin ejecutar en el navegador, ¿Cuál crees que sería el error que se mostrará en la consola? Verifica ahora tu respuesta ejecutando el código directamente en la consola de navegador web.

El error a mostrar es del tipo RangeError, debido al primer error en la segunda línea de código. Ya que no se puede iniciar un arreglo con extensión o largo de -1 o cualquier número negativo.

6. Realizar una función que genere un error en el bloque del “try” indicando que las variables no están declaradas. Para atrapar el error se debe implementar el “catch”.

```
const showError = () => {  
  try {  
    throw 'Las variables no se encuentran declaradas';  
  }  
  catch(e) {  
    console.log(e)  
  }  
}  
  
showError();
```

7. Desarrollar un programa con JavaScript que permita indicar a un alumno, si su respuesta de verdadero o falso estuvo acertada o no. Utiliza promesas, then...catch y funciones para generar el código.

```
const isCorrect = false;

const answer = (data) => {
    return new Promise((resolve, reject) => {
        if(!data.isCorrect){
            reject('La respuesta no es correcta :(')
        } else {
            resolve('Correcto!')
        }
    })
}

answer({ isCorrect })
.then(resp => {
    console.log(resp);
})
.catch(err => {
    console.log(err);
})
```

8. De los datos indicados a continuación. Muestra por consola los valores del nombre y apellido de cada persona, incluyendo una tercera persona ficticia, utilizando la estructura try...catch...finally. Indicando el nombre del error y el mensaje en una ventana con la instrucción alert.

```
const json = {
    "persona": [
        {
            "id": "1",
            "nombre": "Juan",
            "apellido": "Romero."
        },
        {
            "id": "2",
            "nombre": "Jocelyn",
            "apellido": "Perez."
        }
    ];
}

try {
    console.log(`El nombre de la persona ${json.persona[0].id} es:
${json.persona[0].nombre}`);
    console.log(`El nombre de la persona ${json.persona[1].id} es:
${json.persona[1].nombre}`);
    console.log(`El nombre de la persona ${json.persona[2].id} es:
${json.persona[2].nombre}`);
} catch(e) {
    console.log('Errores');
    console.log(e.name, e.message);
} finally {
    console.log('Finalizado el try...catch. Este mensaje aparece
siempre.');
}
```

9. Crear la función **whereToSee** a la que se pueda pasar el argumento **direction**, que tiene 2 valores posibles, **left** y **right**. La función debe retornar siempre lo opuesto a lo que se envía. Se deben manejar los errores en caso de que se envíe a la función otro valor que no sean los posibles, retornando el error con la clase Error.

```
let whereToSee = (valor) => {
    try {
        if (valor === 'right') {
            return 'left';
        } else if(valor === 'left'){
            return 'right';
        } else {
            throw new Error('El valor enviado no es permitido');
        }
    } catch (error) {
        return error;
    }
}
console.log(whereToSee('ddd'));
```

10. Crea una función con el nombre de **ask** y luego crea dos (2) variables denominadas "num1 y num2", a ambas variables se le deben asignar la función **prompt** para solicitar un número cualquiera. Posteriormente se debe validar que realmente se ingresaron los dos números y devolver la suma de ambos. Pero, si no se puede realizar la suma porque el usuario no ingresó ambos números se debe devolver un error mediante el levantamiento de una excepción con **throw**.

```
let ask = () => {
    let num1 = parseInt(prompt('Ingrese el primer número'));
    let num2 = parseInt(prompt('Ingrese el segundo número'));

    try {
        if (num1 && num2) {
            return num1 + num2;
        } else {
            throw new Error('No se puede realizar la suma');
        }
    } catch (error) {
        return error;
    }
}
console.log(ask());
```