APIs

Objetivos

Comprender que son las APIs, algunos tipos y su importancia en él desarrollo web.

Aprender el concepto de API RestFull.

Comprender como funcionan las APIs, y sus estados de respuesta.

Conocer y ejecutar el método fetch

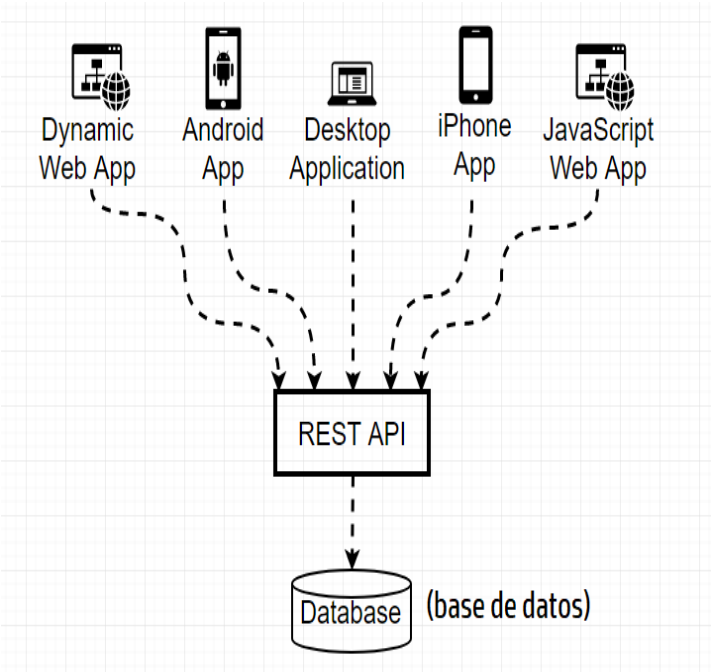
Aprender el concepto de las promesas

Async, await y try catch

Aprender a usar la biblioteca Chart.js para la renderización de gráficos

¿Qué son las APIs?

Una API (Application Programming Interface) es como un conserje o un intermediario en una biblioteca. Proporciona una forma estándar y bien definida para qué los programas o aplicaciones puedan interactuar con un servicio, aplicación o plataforma. Básicamente, es una interfaz que permite que diferentes programas se comuniquen y compartan información entre sí.



Algunos tipos de APIs

APIs Web (RESTful):
Son APIs basadas en el estilo arquitectónico REST (Representational State Transfer) y se utilizan ampliamente en aplicaciones web y servicios en línea. Siguen los principios RESTful y utilizan los métodos HTTP (GET, POST, PUT, DELETE, etc.) para realizar operaciones sobre recursos.

APIs SOAP:
Estas APIs utilizan el protocolo SOAP (Simple Object Access Protocol) para el intercambio de mensajes en el formato XML. Son más estructuradas y rígidas que las APIs REST y se han utilizado tradicionalmente en aplicaciones empresariales y sistemas heredados.

APIs de servicios web:
En general, esta categoría engloba a las APIs basadas en SOAP y REST, ya que ambos tipos permiten la comunicación a través de Internet mediante el uso de estándares como XML o JSON. Los servicios web suelen utilizarse para integrar sistemas distribuidos.

API RESTful

Una API RESTful se considera "RESTful" cuando cumple con las siguientes características y principios:

Basada en recursos:
Los recursos son entidades o datos que se pueden acceder a través de la API. Cada recurso tiene una identificación única, como una URL, y puede tener múltiples representaciones, como JSON o XML.

Operaciones HTTP:
Utiliza los métodos estándar de HTTP para realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) sobre los recursos. Los métodos más comunes utilizados en una API RESTful son GET (para obtener datos), POST (para crear recursos), PUT (para actualizar recursos) y DELETE (para eliminar recursos).

Sin estado:
Cada solicitud de cliente a la API contiene toda la información necesaria para comprender y procesar la solicitud. La API no guarda información sobre el estado de las solicitudes anteriores, lo que la hace independiente y escalable.

Representaciones:
Los recursos se representan en diferentes formatos, como JSON, XML o HTML, según la preferencia del cliente. La API indica el tipo de representación que devuelve mediante los encabezados HTTP (Content-Type).

Interacción uniforme:
Se sigue un conjunto uniforme de convenciones, como el uso de URLs para identificar recursos y los códigos de estado HTTP para indicar el resultado de la solicitud (como 200 para éxito, 404 para recurso no encontrado, etc.).

Sistema cliente-servidor:
La API RESTful sigue el modelo cliente-servidor, lo que significa que el cliente (la aplicación o sistema que realiza la solicitud) y el servidor (la API) están separados y se comunican a través de solicitudes y respuestas HTTP.

Cacheable:
Las respuestas de una API RESTful pueden ser cacheadas, lo que permite que ciertas solicitudes se almacenen temporalmente en el cliente o en intermediarios (como proxies) para mejorar la eficiencia y reducir la carga en el servidor.

¿Cómo funcionan las APIs?

Las APIs funcionan como una capa de abstracción entre el software y los datos.

¿Qué significa esto?

El cliente (software) hace un request (pedido) al servidor. El servidor procesa el pedido y envía una respuesta.

Las APIs se comunican a través de los siguientes métodos:

GET:
El método GET se utiliza para obtener datos de un servidor. Cuando realizas una solicitud GET a una API, estás pidiendo que te devuelva información específica o una lista de recursos. Esta solicitud no modifica ningún dato en el servidor, es solo para lectura de datos.

Ejemplo: cuando queremos obtener informacion de un producto.

POST:
El método POST se utiliza para enviar datos al servidor para crear recursos nuevos. Cuando realizas una solicitud POST a una API, estás enviando datos en el cuerpo de la solicitud, que se utilizarán para crear un nuevo recurso en el servidor.

Ejemplo: cuando nos creamos una cuenta.

PUT:
El método PUT se utiliza para actualizar o reemplazar un recurso existente en el servidor. Al hacer una solicitud PUT a una API, estás enviando datos en el cuerpo de la solicitud que se utilizarán para actualizar el recurso especificado.

Ejemplo: cuando queremos actualizar la informacion de un producto o actualizar nuestro perfil.

DELETE:
El método DELETE se utiliza para eliminar un recurso existente en el servidor. Al hacer una solicitud DELETE a una API, estás solicitando que se elimine el recurso especificado en la URL.

Ejemplo: cuando queremos borra un comentario etc.

Probando Con POSTMAN

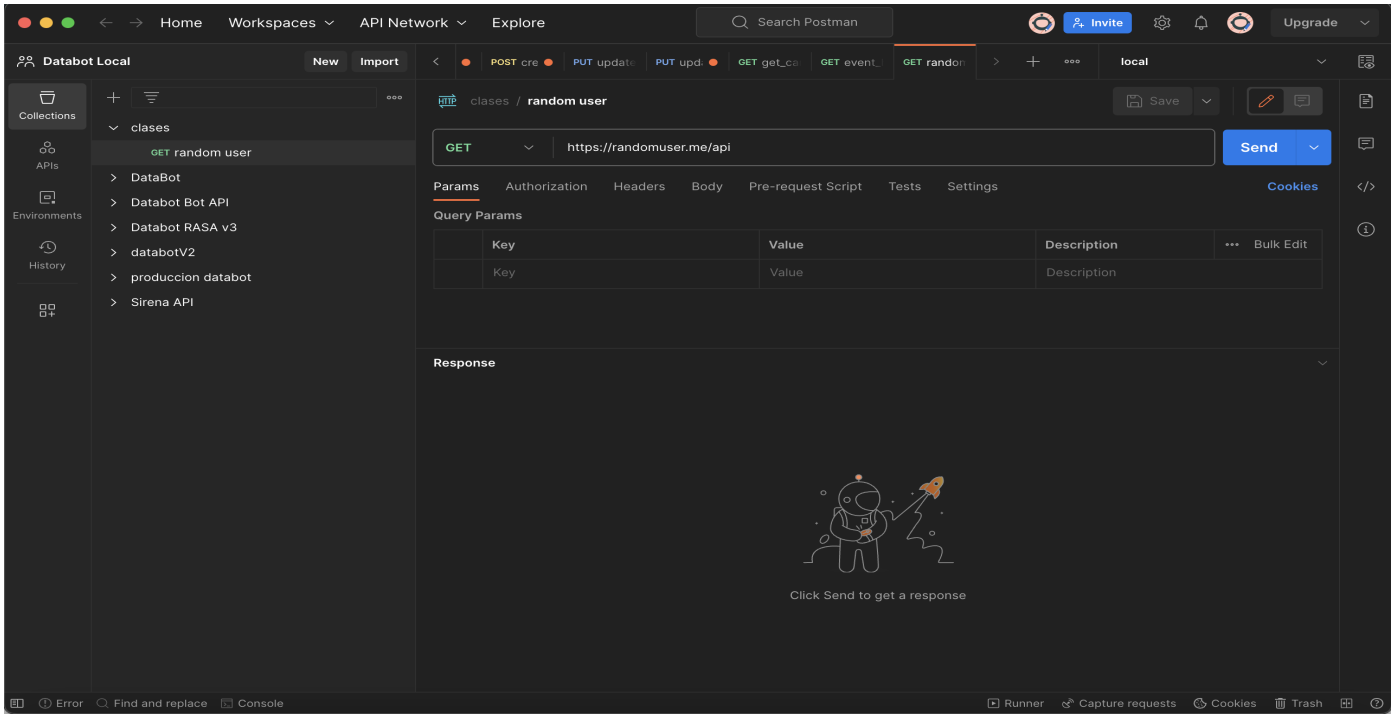
POSTMAN es una plataforma que permite y hace más sencilla la creación y el uso de APIs.

Es una plataforma gratuita si trabajas solo o de pago si quieres trabajar de manera colaborativa con tu equipo y la puedes descargar desde el siguiente enlace:

Descargar POSTMAN

Aprendiendo a Usar POSTMAN

Paso 1:
Una vez descargado POSTMAN desde el sitio web oficial veremos la siguiente interfaz.

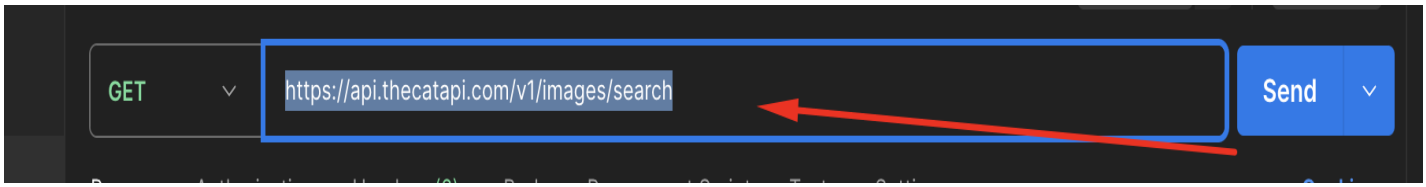


Paso 2:

Vamos a realizar nuestra primera consulta a una API, ya que es nuestro primer acercamiento en esta materia solo aplicaremos una consulta con el método GET para esto utilizaremos la API The Cat API y para esto vamos a ocupar el siguiente end point. https://api.thecatapi.com/v1/images/search

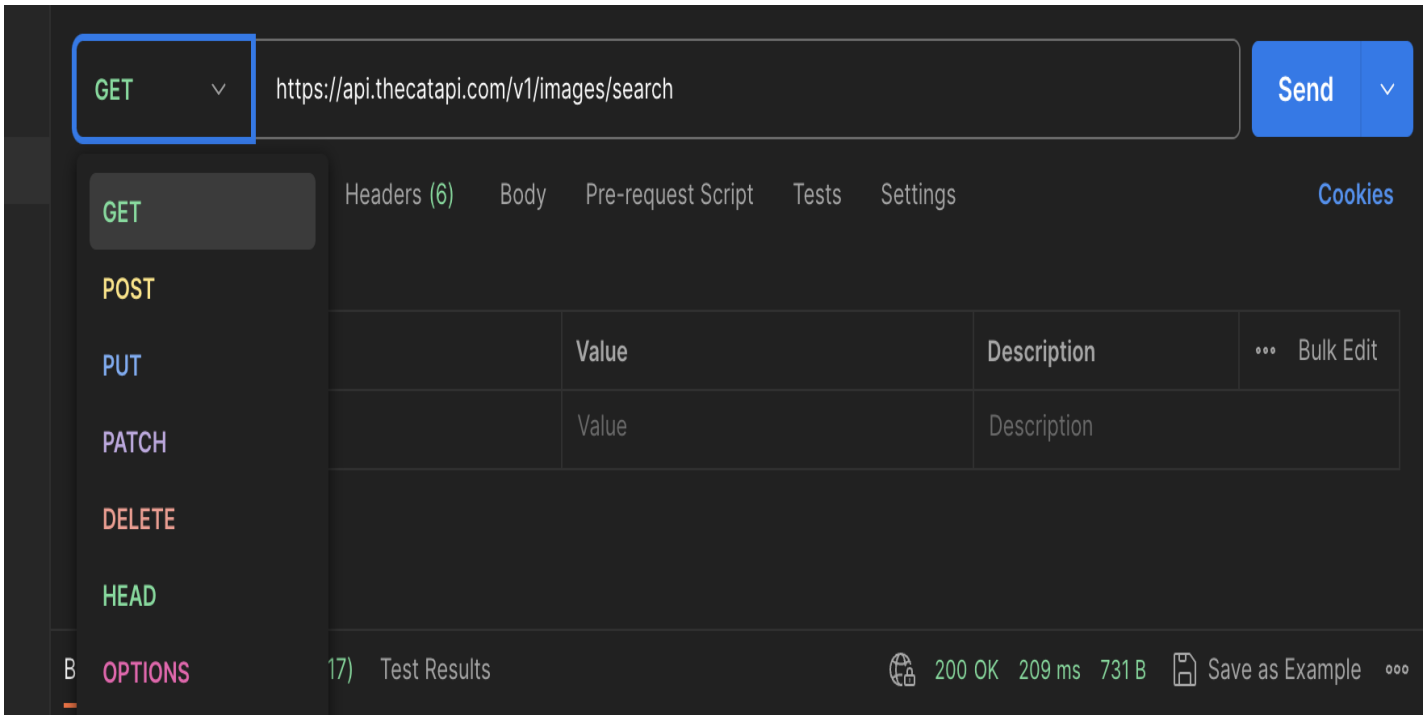
Paso 3:

Vamos a agregar el end point en el siguiente lugar de nuestra aplicación POSTMAN.

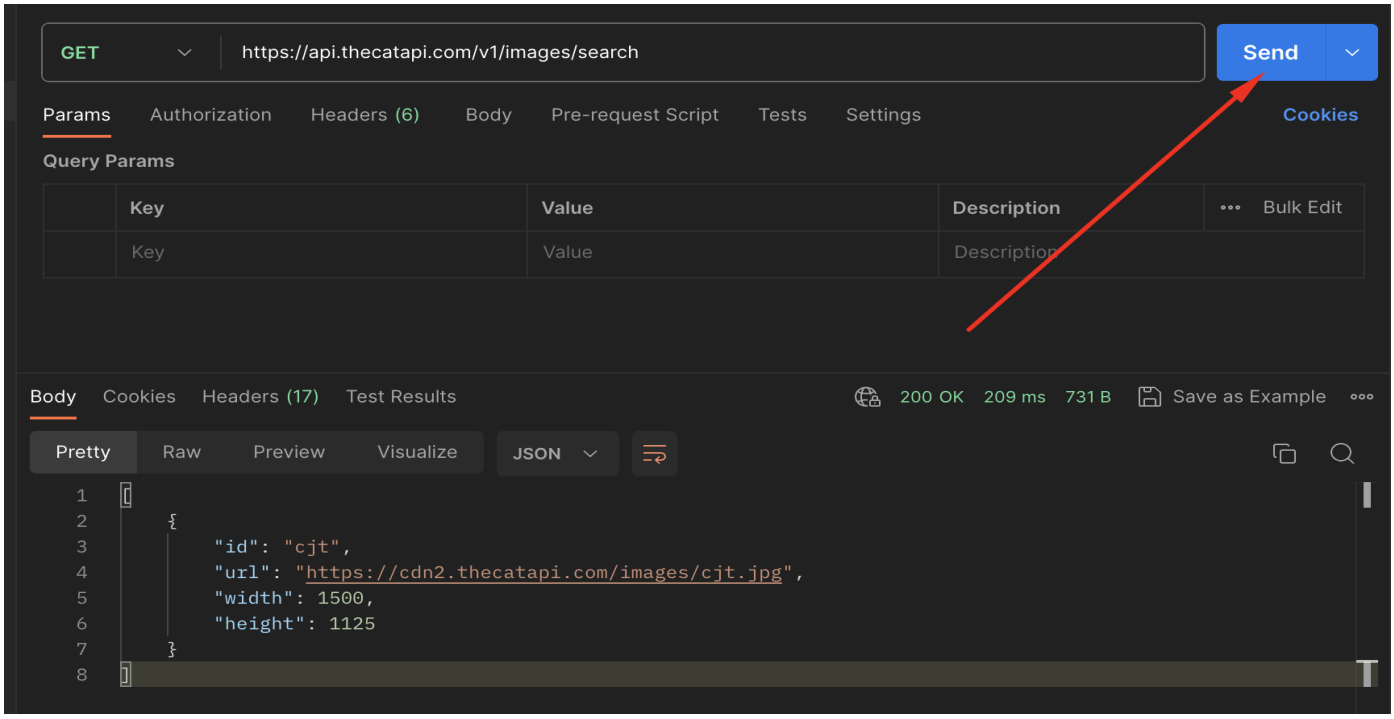


Paso 4:

Vamos a seleccionar el método a usar, en este caso es GET en el siguiente lugar de nuestra aplicación POSTMAN.



Paso 5:
Finalmente enviamos nuestra solicitud y obtendremos una respuesta como se muestra a continuación.



Si haces click en el link que nos entrega la respuesta podrás ver la foto que nos devuelve la API, además se puede apreciar que nos devuelve un código respuesta o de estado **200 ok**

Codigos de estado o de respuesta

Cuando navegas por internet, tu navegador web (como Chrome, Firefox o Safari) hace solicitudes a los servidores para obtener información, como páginas web, imágenes o datos. Cada vez que haces una solicitud, el servidor responde con un "código de estado HTTP". Este código es un número de tres dígitos que indica el resultado de tu solicitud.

¿cuáles son los códigos de estado más comunes?

200 OK:
Indica que la solicitud se ha realizado con éxito, y el servidor ha devuelto los datos solicitados en la respuesta.(GET).

201 Created:
Se utiliza cuando la solicitud ha tenido éxito y ha resultado en la creación de un nuevo recurso en el servidor.(POST)

204 No Content:
El servidor ha procesado la solicitud con éxito, pero no hay datos para devolver en la respuesta. (DELETE)

400 Bad Request:
Indica que la solicitud del cliente es incorrecta o malformada. El servidor no puede entenderla.

401 Unauthorized:
El cliente debe proporcionar credenciales válidas (como un nombre de usuario y contraseña o un token) para acceder al recurso solicitado, pero no lo ha hecho o las credenciales proporcionadas son inválidas.

403 Forbidden:
El servidor entiende la solicitud, pero se niega a autorizarla. El cliente no tiene permiso para acceder al recurso.

404 Not Found:
El servidor no pudo encontrar el recurso solicitado. Es el error más común cuando una URL o recurso no existe.

500 Internal Server Error:
Es un error genérico del servidor que indica que algo salió mal en el servidor al procesar la solicitud, pero no se ha identificado la causa exacta del error.

503 Service Unavailable:
Indica que el servidor no puede manejar la solicitud en este momento. Puede deberse a una sobrecarga temporal o mantenimiento del servidor.

504 Gateway Timeout:
El servidor, como puerta de enlace, no recibió una respuesta a tiempo de un servidor ascendente o de reenvío, mientras esperaba mantener la conexión.

La función fetch

El método fetch() es una función incorporada en JavaScript que se utiliza para realizar solicitudes de red y obtener recursos de forma asíncrona a través de la web. Con fetch(), puedes enviar peticiones HTTP a servidores para obtener datos, enviar datos al servidor o realizar otras operaciones relacionadas con la red.

Sintaxis de la función fetch:

```
fetch(url)
  .then(response => {
     Aquí se maneja la respuesta de la solicitud
     El contenido de la respuesta se encuentra en response.json(),
    response.text(), etc.
  })
  .catch(error => {
    Aquí se maneja el error, si ocurre algún problema con la solicitud
  });
```

La función fetch() devuelve una promesa que se resuelve con un objeto Response cuando se completa la solicitud. Luego, puedes utilizar los métodos proporcionados por el objeto Response para obtener los datos de la respuesta, como json() para obtener los datos en formato JSON o text() para obtener el contenido de la respuesta como texto.

¿cuando falla el fetch ?

El objeto Promise devuelto desde fetch() no será rechazado con un estado de error HTTP incluso si la respuesta es un error HTTP 404 o 500. En cambio, este se resolverá normalmente (con un estado ok configurado a false), y este solo sera rechazado ante un fallo de red o si algo impidió completar la solicitud.

Este estado de error lo podemos manejar de la siguiente manera.

Sintaxis de la función fetch manejando error en la solicitud:

```
fetch(url)
.then(response => {
  aqui se maneja el error lanzando una excepcion si el status
  es diferente a 200
  if (!response.ok)
  throw new Error("WARN", response.status);

  Aquí se maneja la respuesta de la solicitud
  El contenido de la respuesta se encuentra en response.json(),
  response.text(), etc.
})
.catch(error => {
  Aquí se maneja el error, si ocurre algún problema con la
    solicitud
});
```

Promesas

Para entender que son las promesas primero debemos saber que son las operaciones asíncronas en javascript.

Las operaciones asíncronas en JavaScript son aquellas que no se ejecutan de manera secuencial en el orden en que aparecen en el código. En lugar de esperar que una operación se complete para continuar con la siguiente, JavaScript puede realizar operaciones asíncronas mientras continúa ejecutando el resto del código sin detenerse.

entonces las promesas son:

un objeto que representa la finalización (éxito o fracaso) de una operación asíncrona y nos permite manejar el resultado de esa operación de una manera más estructurada y clara.

Para entender mejor qué es una promesa, consideremos el ejemplo de una operación asíncrona, como la descarga de un archivo o la solicitud de datos a un servidor. En JavaScript, las operaciones asíncronas no se resuelven inmediatamente, lo que significa que el resultado puede no estar disponible de inmediato. En lugar de esperar el resultado, podemos utilizar una promesa para "prometer" que manejará el resultado una vez que esté disponible, ya sea éxito o fracaso.

Las promesas tienen tres estados:

Pendiente (pending):
Cuando se crea una promesa, está en estado pendiente. Esto significa que la operación aún no se ha completado y estamos esperando su resultado.

Cumplida (fulfilled o resolved):
Cuando la operación se completa con éxito, la promesa pasa al estado de cumplida. En este punto, podemos obtener el resultado de la operación exitosa utilizando el método then() de la promesa.

Rechazada (rejected):
Si la operación falla o encuentra un error, la promesa pasa al estado de rechazada. En este caso, podemos manejar el error utilizando el método catch() de la promesa.

Async y Await

async y await son características de JavaScript que proporcionan una forma más concisa y legible de trabajar con operaciones asíncronas. Estas palabras clave se utilizan en funciones y permiten que el código asíncrono se comporte de manera más similar al código síncrono, lo que facilita la comprensión y el manejo de flujos de control complejos.

Async:
Cuando se coloca la palabra clave async antes de una función, esa función se convierte automáticamente en una función asíncrona. Esto significa que la función devolverá una promesa.

Await:
La palabra clave await solo se puede usar dentro de una función async. Cuando se coloca antes de una operación asíncrona (como una promesa), indica que la función debe esperar hasta que la operación se complete y el resultado esté disponible antes de continuar con la ejecución del código.

Sintaxis de async y await:

```
async function obtenerDatos() {
  Supongamos que fetchData() es una función que devuelve una
  promesa
  const resultado = await fetchData();
  console.log(resultado);
}

o tambien

const data = async() =>{
  const resultado = await fetchData();
  console.log(resultado);
}
```

¿qué es try catch?

son bloques de código utilizados en JavaScript para manejar errores y excepciones de manera controlada. Permiten que tu código siga ejecutándose incluso si ocurre un error sin que se detenga abruptamente, lo que mejora la robustez y la legibilidad del código.

Sintaxis de try catch:

```
try {
  Código que podría generar un error o excepción
} catch (error) {
  Código para manejar el error
}
```

try:
En este bloque, se coloca el código que quieres probar. Es el código donde puede ocurrir un error. Si algo dentro del bloque try genera una excepción o un error, el control salta al bloque catch.

catch:
El bloque catch se ejecuta solo si se produce un error en el bloque try. Dentro del bloque catch, puedes especificar cómo deseas manejar el error. El error que se produce en el bloque try se pasa automáticamente como un argumento al bloque catch, y generalmente se almacena en una variable, como error (puedes usar cualquier nombre que desees).

finally:
El bloque finally se ejecuta siempre, independientemente de si se produce un error o no en el bloque try. Esto significa que el código dentro del bloque finally siempre se ejecutará, incluso si se produce un error en el bloque try o se ejecuta una declaración return en el bloque try.

Sintaxis de try catch con finally:

```
try {
  Código que podría generar un error
  const resultado = ; Esto generará un error
  console.log("Resultado:", resultado); Esta línea nunca se ejecutará
} catch (error) {
  Manejo del error
  console.error("Ha ocurrido un error:", error.message);Imprime el
                                                        mensaje de error
} finally {
  Este bloque siempre se ejecutará, si o si
  console.log("Bloque finally: Esta línea se ejecutará siempre.");
}
```

Chart.js

Chart.js es una popular librería de JavaScript que se utiliza para crear gráficos y visualizaciones de datos interactivas en páginas web. Con Chart.js, los desarrolladores pueden generar fácilmente diferentes tipos de gráficos, como gráficos de barras, gráficos circulares (tarta/pie), gráficos de líneas, gráficos de áreas, gráficos de radar y más.

Algunas características destacadas de Chart.js son:

Sencillo de usar:
Chart.js es conocido por su facilidad de uso y su API clara y concisa. Permite a los desarrolladores crear gráficos con solo unas pocas líneas de código.

Personalización:
La librería proporciona una amplia gama de opciones de personalización para que puedas adaptar los gráficos a tus necesidades. Puedes cambiar colores, fuentes, tamaños y más.

Interactividad:
Los gráficos creados con Chart.js son interactivos y pueden responder a eventos, como clics o desplazamientos del mouse, lo que mejora la experiencia del usuario.

Adaptabilidad:
Los gráficos generados con Chart.js se adaptan fácilmente a diferentes tamaños de pantalla y dispositivos, lo que los hace adecuados para aplicaciones web responsivas.

Creando nuestro primer gráfico

A continuación crearemos nuestro primer gráfico y analizaremos las partes que componen el código de este, para esto utilizaremos él un gráfico de barras proporcionado por la librería.

Link a la documentación

Paso 1:
Agregamos el siguiente cnd "https://cdn.jsdelivr.net/npm/chart.js"> en nuestro documento HTML

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta
      name="viewport"
      content="width=device-width,
    initial-scale=1.0"
    />
    <title>Document</title>
    <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  </head>
  <body>
    <div style="width: 80%; margin: 0 auto">
      <canvas id="chart"></canvas>
    </div>
    <script src="./index.js"></script>
  </body>
</html>
```

Paso 2 analizamos el código del gráfico:

Supongamos que queremos mostrar un gráfico de la cantidad de ventas mensuales de un almacén para esto usaremos el siguiente array de objetos:

```
const ventasMensuales = [
  { mes: 'Enero', ventas: 120 },
  { mes: 'Febrero', ventas: 80 },
  { mes: 'Marzo', ventas: 150 },
  { mes: 'Abril', ventas: 100 },
  { mes: 'Mayo', ventas: 200 },
  { mes: 'Junio', ventas: 180 },
  { mes: 'Julio', ventas: 140 },
  { mes: 'Agosto', ventas: 170 },
  { mes: 'Septiembre', ventas: 90 },
  { mes: 'Octubre', ventas: 120 },
  { mes: 'Noviembre', ventas: 160 },
  { mes: 'Diciembre', ventas: 190 }
];
```

A continuación podemos apreciar el código del gráfico él que recibe los meses y las ventas

```
const renderChart = (months, sales) => {
  const ctx = document.getElementById('chart').getContext('2d');
  new Chart(ctx, {
    type: 'bar',
    data: {
      labels: months,
      datasets: [{
        label: 'Number of sales',
        data: sales,
        backgroundColor: 'rgba(54, 162, 235, 0.5)',
        borderColor: 'rgba(54, 162, 235, 1)',
        borderWidth: 1,
      }],
    },
    options: {
      responsive: true,
      maintainAspectRatio: false,
      scales: {
        y: {
          beginAtZero: true,
        },
```

```
        },
      },
    });
  };
```

En donde podemos apreciar lo siguiente:

type:
Define el tipo de gráfica que se renderiza. Entre las opciones tenemos: line, bar, radar, entre otros.

labels:
Es un arreglo de títulos o Strings que representarán los valores de la gráfica en el eje horizontal. Estos valores serán sincronizados según el orden en el que se declaren los datos.

datasets:
Es el grupo de estadísticas que se renderizan en la misma gráfica. Para los ejemplos de esta lectura solo trabajaremos con 1 estadística

label:
Será el título del grupo de datos que se definan en la data de esta estadística

backgroundColor:
El color que los datos que se expondrá en la gráfica.

data:
Es un arreglo de números que se tomarán como los valores que se insertarán en el eje vertical.

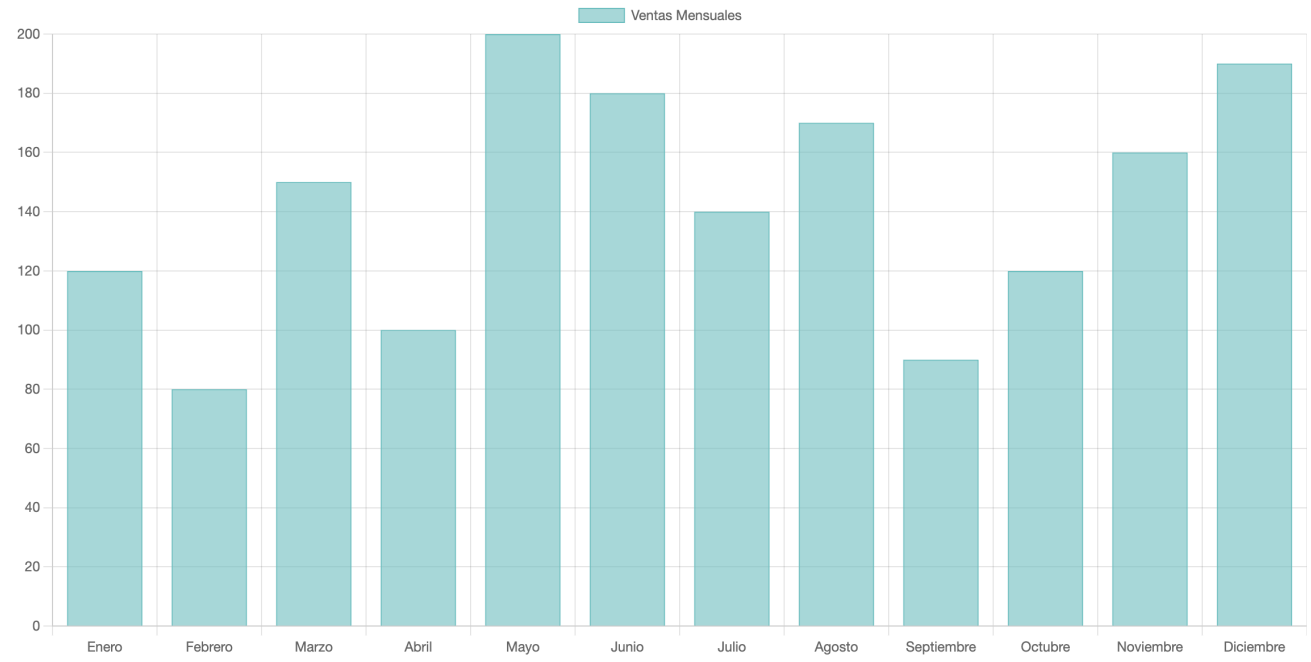
Paso 4:
Agregamos un bloque HTML con el, id llamando 'chart' en él cuál se va a renderizar nuestro gráfico, este id debe coincidir con él que se llama en la función que construye el gráfico

Paso 5:

Finalmente, debemos procesar nuestro array de ventas para entregar los datos según los requerimientos del gráfico, para los labels entregaremos un array de meses y para las ventas un array con la cantidad de ventas, para esto utilizaremos el siguiente código usando la función map.

```
// Obtener los datos necesarios para el gráfico
const months = ventasMensuales.map(venta => venta.mes);
const sales = ventasMensuales.map(venta => venta.ventas);
```

Ejecutamos nuestro archivo HTML y nuestro gráfico se debería ver como en la siguiente imagen.



Para finalizar esta, guia te dejo el siguiente ejercicio el cual es consumir la api de la nasa extraer las imágenes del rover, generar tarjetas en las cuales estas se vean reflejadas, y posteriormente crear un gráfico el de tu elección analizando los nombres de las cámaras y la cantidad de fotos.

Mucho éxito en tu aprendizaje