Guía de ejercicios 6 - APIS

 ¡Hola! Te damos la bienvenida a esta nueva guía de ejercicios.

¿En qué consiste esta guía?

En la siguiente guía podrás trabajar los siguientes aprendizajes:

- Consultar una API y mostrando por consola la respuesta
- Presentar los datos de una API en una página web
- Captura errores usando try catch
- Utilizar un plugin de JavaScript para renderizar gráficas

```
async function getRandomUser(){
  const res = await fetch("https://randomuser.me/api")
  const data = await res.json()
  console.log(data)
  const element = document.querySelector(".user")
  element.innerHTML = data.results[0]['email']
}
getRandomUser()
```

Tabla de contenidos

¡Comencemos!

Introducción a APIs

Una interfaz de programación de aplicaciones (API) es un conjunto de herramientas, definiciones y protocolos que se usa para diseñar e integrar software de aplicaciones. Permite que un producto o servicio se comunique con otros productos y servicios, sin la necesidad de saber cómo se implementan.

Transferencia de Estado Representacional (REST)

Existen múltiples tipos de APIs. En esta guía trabajaremos con un tipo muy utilizado de APIs llamado APIs Rest el cual permite comunicar distintas aplicaciones a través del protocolo HTTP de una forma sencilla A partir de ahora cuando hablemos de APIs estaremos asumiendo que estas son APIs Rest.

O sea, podemos crear páginas web o aplicaciones que interactúen con otras aplicaciones a través de APIs sin tener que preocuparnos de cómo fueron creadas estas aplicaciones o que tecnología que hay detrás de ellas. Podemos utilizarlas si estas están disponibles en internet.

Hay APIs para distintos propósito:

- Datos del clima
- Servicios de geolocalización
- Indicadores financieros
- Actualizar redes sociales
- Automatización de acciones en campañas de marketing

Cliente y servidor

Las APIs REST funcionan bajo una lógica de cliente y servidor.

Imagen 1. Introducción a APIs REST
Fuente: Desafío Latam

El cliente hace un request (pedido) al servidor. El servidor procesa el pedido y envía una respuesta. Cuando consumimos APIs estamos trabajando del lado del cliente solo tenemos que preocuparnos de enviar correctamente el pedido y luego realizar las acciones correspondientes con la respuesta.

Haciendo un request

Para conectarnos a una API lo hacemos a través de un **request** (pedido) a un endpoint. Un endpoint es un punto de acceso y es muy similar a la dirección de una página web.

¿Cómo hacer un request?

Podemos hacer un **request** de las siguientes formas:

- Ingresando el endpoint como dirección en el navegador (esto solo servirá en algunos casos)
- Utilizar el plugin de Visual Studio Code llamado Thunder Client o herramientas como Postman

- Utilizando Javascript u otro lenguaje de programación
- Utilizando el comando curl desde la línea de comandos.

Probando un endpoint con el navegador

Partiremos probando un endpoint desde el navegador, para esto ingresaremos a https://randomuser.me/api el cual es un endpoint de una API que genera datos al azar de personas, es una buena API para experimentar y aprender.

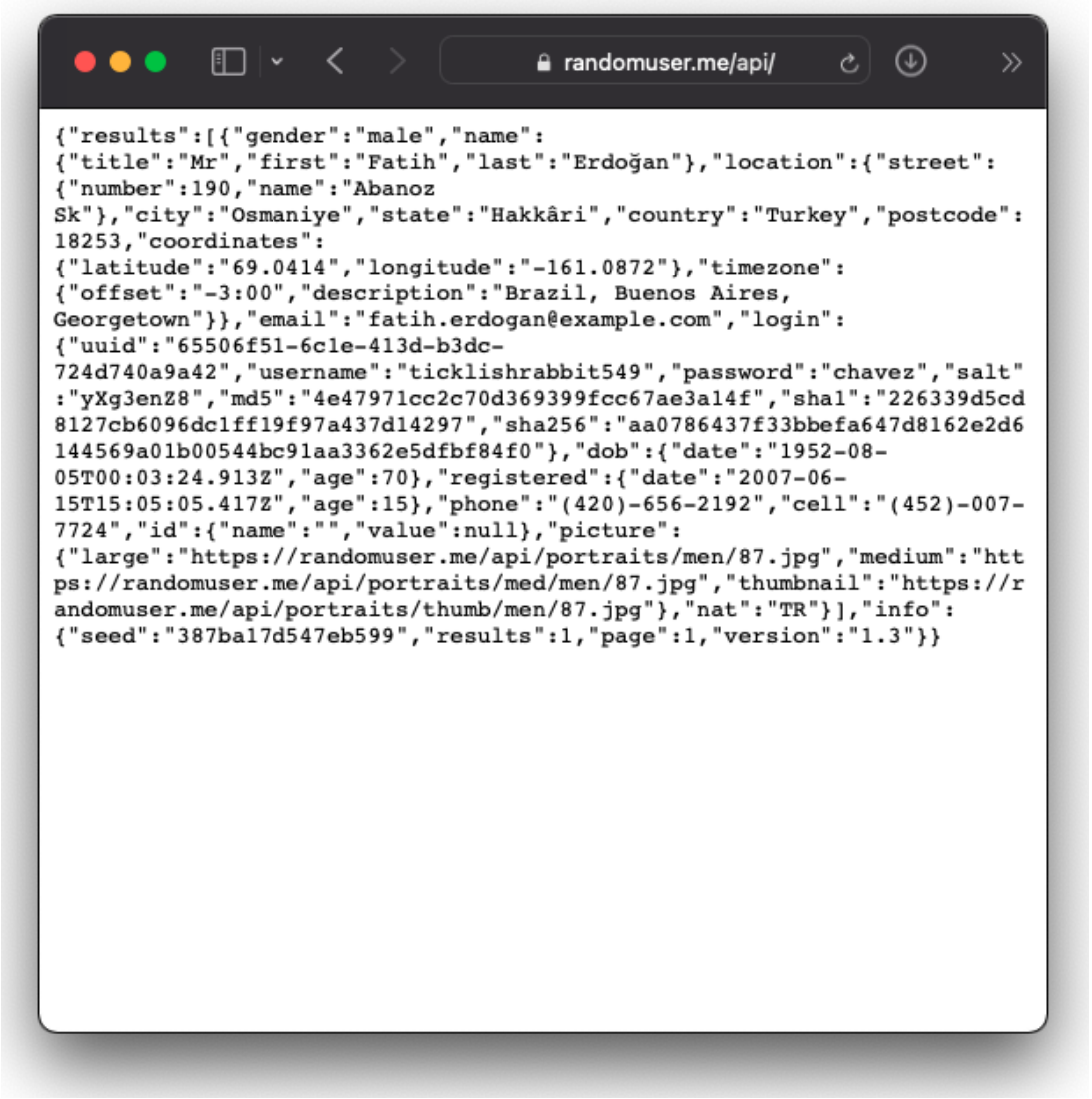

Imagen 2. Consultando una API REST con el navegador
Fuente: Desafío Latam

Podemos ver en el navegador que es similar a una página web, pero solo hay datos, esto es normal. Las APIs están hechas para ser consumidas por programas. A lo largo de la guía crearemos programas que obtengan estos datos y modifiquen el DOM de una página web a partir de ellos.

Probando un endpoint con Thunder Client

Utilizando el navegador estamos muy limitados respecto a los tipos de pedido que podemos hacer, para poder hacer otros instalaremos el plugin de visual studio code llamado Thunder Cliente.

Dentro de visual studio code busca en extensiones thunder cliente e instala la herramienta.

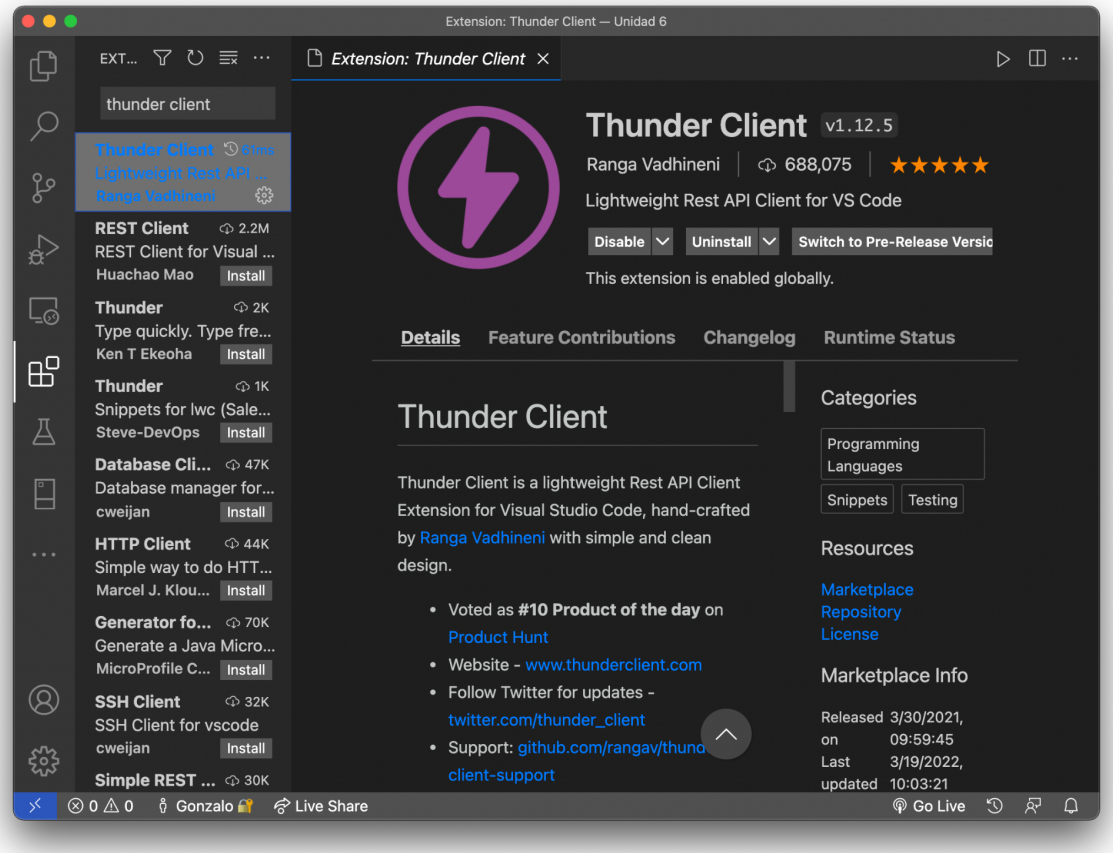

Imagen 3. Buscando el paquete Thunder Client
Fuente: Desafío Latam

Una vez instalada la herramienta dentro del menú lateral de visual studio tendrás el ícono de thunder client, una vez que entres a esta sección presiona el botón con el texto new request

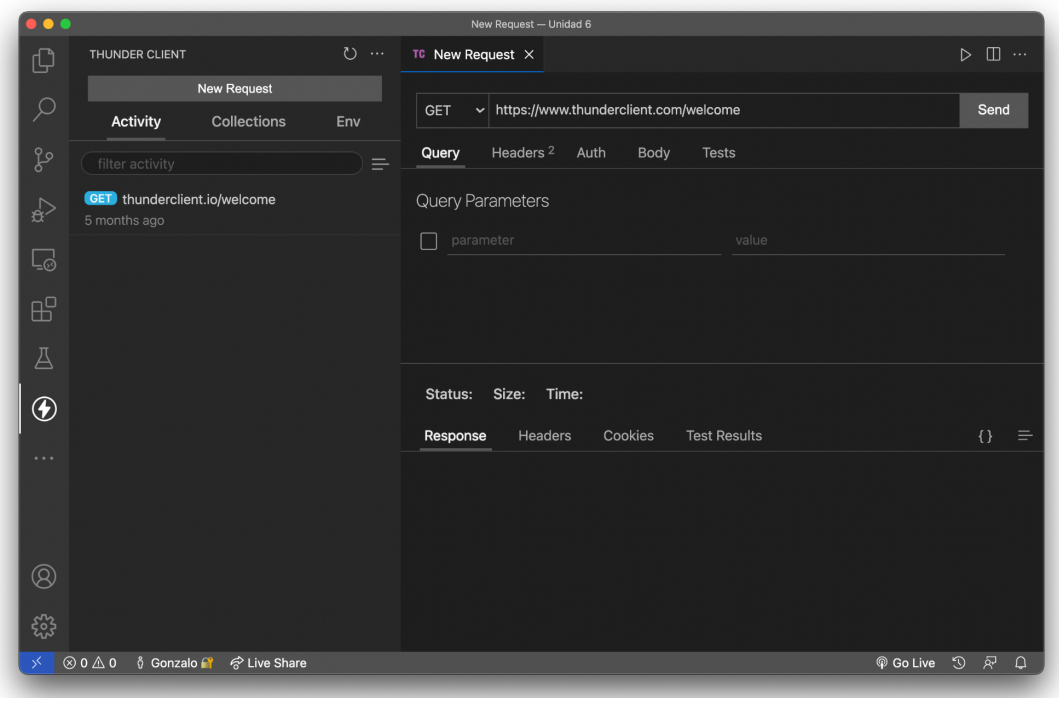

Imagen 4. Haciendo un request con Thunder Client
Fuente: Desafío Latam

En el panel de la derecha ahora podremos ver mucha más información que le podemos enviar a la API, por ahora simplemente ingresaremos la URL https://randomuser.me/api y haremos click en send y podremos observar la respuesta en el mismo editor de código

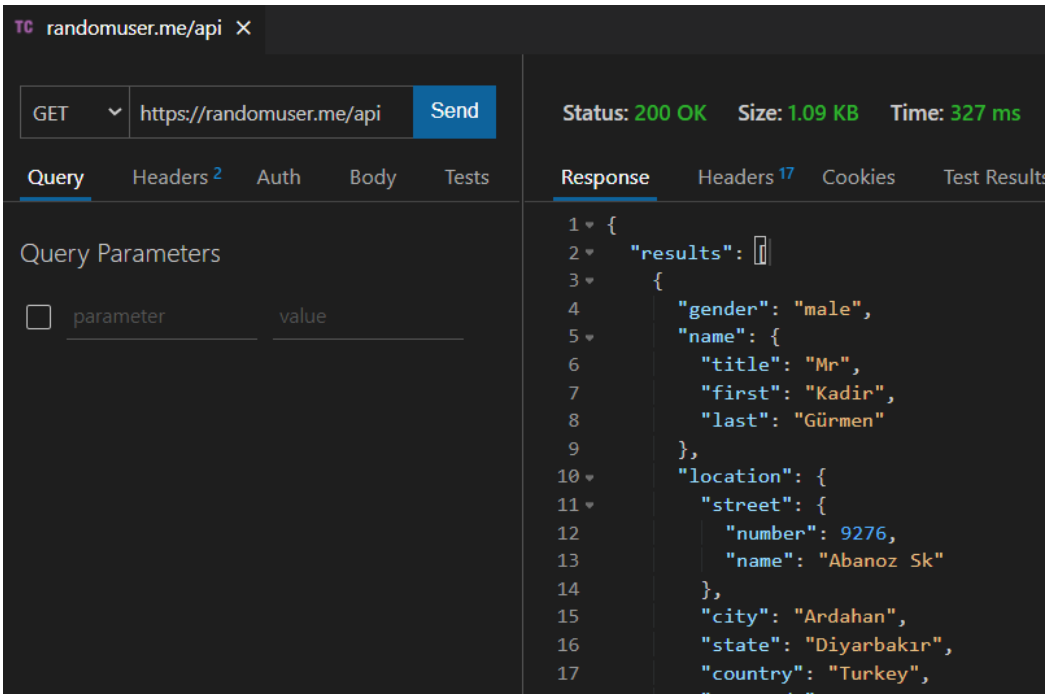

Imagen 5. Observando la respuesta desde Thunder Client
Fuente: Desafío Latam

Partes de un endpoint

Hasta el momento hemos hablado del endpoint como si fuera únicamente una URL pero realmente un endpoint tiene 2 partes importantes.

Un endpoint se compone de una ruta (URL) y un método

1. Una ruta (PATH)
2. Un método

Uno realiza una acción sobre una Ruta por ejemplo `<get> <randomUser.me/api>` donde podríamos traducir get como obtener.

Métodos HTTP

Para indicar el tipo de operación que se llevará a cabo, existen métodos HTTP para cada caso y de esta forma no tener verbos en las rutas.

Existen 4 acciones básicas HTTP que son utilizadas para interactuar con servidores:

- **GET:** Este método se utiliza para obtener un recurso específico o una colección de datos.
- **POST:** Este método se usa para crear recursos nuevos.
- **PUT:** Este método se usa para actualizar un recurso.
- **DELETE:** Este método se usa para eliminar recursos.

Existen otras 3 como Patch, Head y Options

Dentro de thunder client podemos hacer click donde dice get para seleccionar la opción que necesitemos. Esta no la tenemos que saber de memoria, saldrá en la documentación de la API.

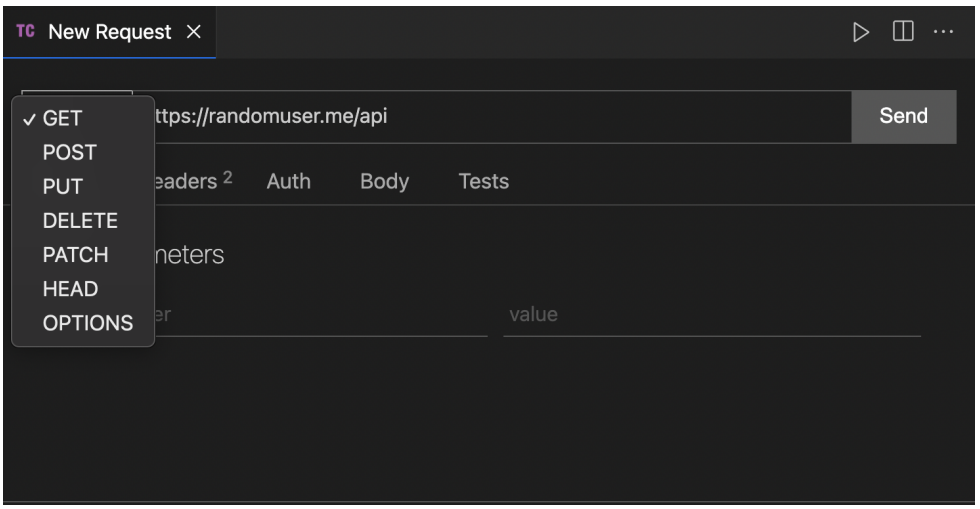

Imagen 6. Tipos de pedidos posibles desde Thunder Client
Fuente: Desafío Latam

Solo podemos hacer el request con el navegador cuando el método es GET

Anatomía de una respuesta

Una respuesta se compone de dos partes importantes:

1. El código de respuesta
2. El cuerpo de la respuesta

Códigos de respuestas (Response Codes)

El código de respuesta nos permite entender si la consulta fue realizada con éxito, si hay un problema con la consulta realizada por parte del cliente o bien, el servidor presenta un problema al momento de realizar la consulta.

Algunos códigos de respuesta bien frecuente son:

- 200 (OK)
- 201 (CREATED)
- 204 (NO CONTENT)
- 400 (BAD REQUEST)
- 403 (FORBIDDEN)
- 404 (NOT FOUND)
- 500 (INTERNAL SERVER ERROR)

Particularmente lo que esperamos obtener es obtener el código 200 para la mayoría de las respuestas.

Cuerpo de la respuesta

La respuesta puede venir en distintos formatos, usualmente al consultar APIs vendrán en formato de texto plano llamado json el cual es un formato que es muy fácil de traspasar a Javascript y a otros lenguajes de programación. El siguiente texto es un ejemplo válido:

```
{
  "customer" : {
    "name": "Scylla Buss"
    "email": "scyllabuss1@some.com"
  }
}
```

Si bien JSON parece ser Javascript, no lo es, los datos están en este formato de texto plano y si queremos utilizarlo tendremos que aprender a transformar estos datos a JS. Lo estudiaremos más adelante.

Actividad 1: Usando una API Rest

En la página de https://mindicador.cl/ hay documentada una API que sirve para tener distintos indicadores económicos de Chile. Realiza requests a los distintos endpoints utilizando Thunder Client y observa la respuesta.

Resumen

- Una API Rest es como un sitio web pero para ser consumida por programas en lugar de usuarios finales.
- Una API Rest funciona bajo una lógica cliente / servidor. El cliente realiza pedidos a la API y el servidor responde.
- Para recibir pedidos una API tiene endpoints, los endpoints vendrían siendo como las distintas páginas web de un sitio web
- Cada endpoint se compone de una acción (o método) más una ruta, ejemplo: get https://url-a-una-api/usuarios Podemos averiguar los endpoints disponibles
- Podemos hacer requests de tipo `<get>` a una API ocupando el navegador. Para realizar otro tipo de acciones deberemos ocupar Thunder Client, Postman, la línea de comando o directamente desde un lenguaje de programación como aprenderemos

posteriorment

- Una respuesta de una API se compone de un código de respuesta y un cuerpo.
- Un código de respuesta 200 indica que todo está bien. Un 500 indica que hay un problema con la API. Para conocer todos los tipos de respuesta posible buscar HTTP STATUS CODES
- El cuerpo de la respuesta probablemente venga en formato JSON, (una respuesta también podría estar en HTML, XML u otro formato)

Requests desde JavaScript

A partir de ahora aprenderemos a consultar endpoints utilizando JavaScript y luego modificar el DOM con la respuesta obtenida.

El método fetch

Desde JavaScript podemos consultar una API utilizando el método fetch, el siguiente código muestra un uso básico de la instrucción.

```
async function getRandomUser(){
  const res = await fetch("https://randomuser.me/api")
  const data = await res.json()
  console.log(data);
}

getRandomUser()
```

El código anterior lo podemos agregar dentro de una página web vacía en el body con la etiqueta script. Al abrir la página veremos lo siguiente en la consola del navegador:

Imagen 7. Console.log de un fetch a randomUser.me
Fuente: Desafío Latam

Revisemos el código:

1. Por ahora ignoremos todo lo que diga async y await
2. Fetch hace un request al endpoint
3. `res.json` Transforma los resultados para que podamos leerlos fácilmente como un objeto de Javascript
4. Si quisiéramos consultar otra API solo tendríamos que cambiar el endpoint (y el nombre de la función para ser coherentes)
5. Guardamos en `res` la respuesta de la consulta. Este objeto contiene información cómo el código de estado HTTP, las cabeceras, si existió un problema de comunicación y el cuerpo de la respuesta.
6. Con `res.json()` podemos parsear el formato JSON y luego leer respuesta como si fuera cualquier otro objeto de JavaScript

Async / Await

Los métodos fetch y json() devuelven un tipo de dato especial llamado promesa, estas promesas deben resolverse en algún momento pero esto puede demorar, para esperar el resultado utilizaremos await.

await espera que una promesa se resuelva. Una regla importante es que que await solo puede ser utilizada dentro de una función declarada async.

Actividad 2: Consultando los datos de Pikachu

Realicemos el siguiente ejercicio en donde debas crear una función async/await para consultar el siguiente endpoint:

- https://pokeapi.co/api/v2/pokemon/pikachu

Y mostrar la data resultante por consola.

La respuesta

Modificando el DOM con datos de una API

Podemos darle muchas utilidades a los datos que obtenemos de las APIs que consultamos cómo por ejemplo crear un Dashboard que presente las principales ciudades de Chile con sus temperaturas actuales.

Y ¿De donde vamos a sacar la información de las temperaturas? Existen varias APIs que nos ofrecen esta data, una de ellas es la API GAEL que nos ofrece diferentes datos, entre ellos el siguiente endpoint: https://api.gael.cloud/general/public/clima

Con el que obtendremos los datos climáticos de las principales ciudades de Chile

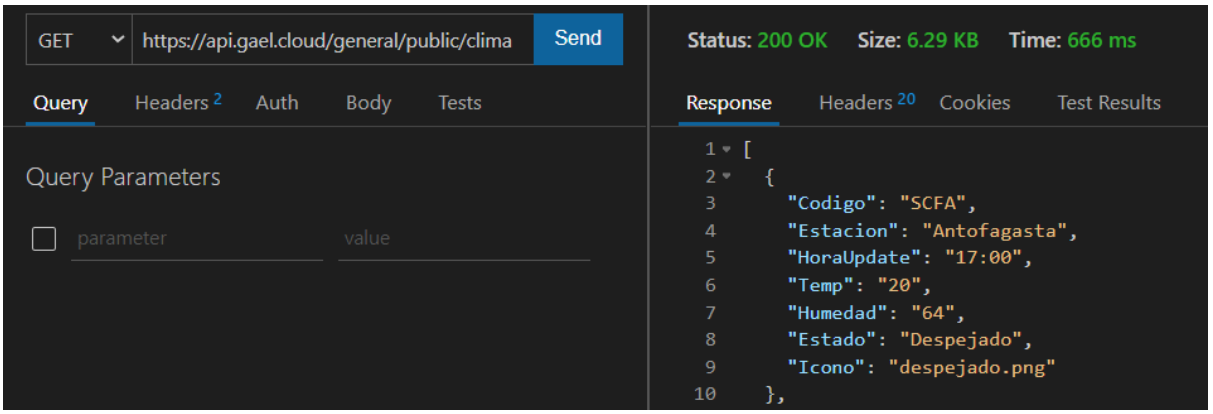

Imagen 8. Consultando climas
Fuente: Desafío Latam

Realicemos el ejercicio del Dashboard con los Climas de las ciudades de Chile ocupando los siguientes códigos:

HTML

```
<h1>Ciudades - Climas</h1>
<section class="climas"></section>
```

CSS

```
body {
  text-align: center;
  padding: 0px 40px;
}

.climas {
  display: grid;
  grid-template-columns: repeat(4, 1fr);
  gap: 20px;
}

.clima {
  background: black;
```

```
  color: white;
}
```

Ahora procedamos con:

1. Crear la referencia de la sección de climas del DOM

```
const climasSection = document.querySelector(".climas");
const apiURL = "https://api.gael.cloud/general/public/clima";
```

2. Crear una función para obtener los climas de la API GAEL

```
async function getClimas() {
  const res = await fetch(apiURL);
  const climas = await res.json();
  return climas;
}
```

3. Crear una función que renderice los climas en el DOM

```
async function renderClimas() {
  const climas = await getClimas();
  let template = "";

  climas.forEach((clima) => {
    template += `
          <div class="clima">
              <h3>${clima.Estacion}</h3>
              <p>${clima.Temp}°C</p>
          </div>
        `;
  });
  climasSection.innerHTML = template;
}

renderClimas()
```

Ejecutando el script anterior podremos observar en el navegador que tenemos como resultado un Dashboard de Climas de las principales ciudades de Chile:

Imagen 9. Dashboard de Climas
Fuente: Desafío Latam

Actividad 3: Usuarios de JSONPlaceholder

Realicemos el siguiente ejercicio en donde debas crear galería de Cards con el nombre, correo y número de teléfono de los 10 usuarios que nos entrega la API JSONPlaceholder en el siguiente endpoint: https://jsonplaceholder.typicode.com/users

Realiza los siguientes pasos:

1. Crear una sección en el HTML y seleccionar el elemento del DOM utilizando JavaScript
2. Crear una función para obtener los usuarios de JSONPlaceholder
3. Crear una función que renderice los usuarios en el DOM
4. Realiza ajustes estéticos que creas convenientes

El resultado deberá ser el siguiente:

Imagen 10. Usuarios de JSONPlaceholder
Fuente: Desafío Latam

AJAX

Otro nombre para lo que hemos visto hasta ahora es AJAX, Asynchronous JavaScript And XML. Prácticamente consiste en obtener datos de un servidor y producto de esto actualizar la página.

AJAX no necesariamente se utiliza en conjunto con una API. El método fetch podría traer documentos o información de cualquier página web y producto de eso actualizar la página sin necesidad de actualizar, esto podría ser útil por ejemplo para agregar un comentario en una página donde esté corriendo un video. Si bien no cubriremos ejemplos de esto, el flujo de trabajo para lograrlo es similar, se ocupa la instrucción fetch y producto de la respuesta se actualiza la página.

Sentencias Try y Catch

Cuando hay un error en nuestro código, el navegador muestra el error e interrumpe la lectura del código. Con la sentencia catch podemos definir un flujo a seguir si lo que está dentro de try llega a fallar.

¿Qué puede fallar?

Cuando trabajamos con API la API puede estar caída o muy lenta y no devolvernos la respuesta a nuestra consulta, en ese caso vamos a querer mostrar un mensaje de error en el DOM al usuario o quizas reintentar el llamado, para este tipo de situaciones try y catch son muy útiles.

Veamos el try catch en acción con el siguiente código:

```
async function getSomething() {
  try {
    const res = await fetch("https://estapaginanoexiste.cl");
    const data = await res.json();
    console.log(data);
  } catch (e) {
    alert(e.message);
  }
}

getSomething();
```

Si ejecutamos el código en un navegador, veremos como aparece una ventana emergente con el mensaje de error correspondiente a lo que haya sucedido dentro del bloque try.

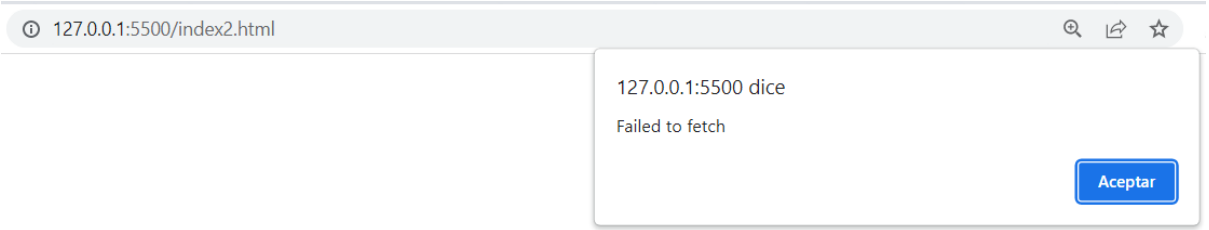

Imagen 11. Mostrando el error al usuario
Fuente: Desafío Latam

La descripción de este mensaje de error está definida por el lenguaje de programación, no obstante podemos escribir nuestros propios mensajes de error con la declaración *throw* que veremos a continuación.

Actividad 4: Otro ejemplo de captura de errores

Realicemos el siguiente ejercicio en donde dado el siguiente fetch:

```
fetch("http://otroejemplo.cl");
```

Escribe una función async await que contenga un try catch en donde se ejecute la consulta y se capture el error.

El mensaje de error debe ser mostrado al usuario como contenido HTML de una etiqueta span.

Imagen 12. Otro ejemplo de error capturado
Fuente: Desafío Latam

Plugins de JavaScript (Chart Js)

Las bibliotecas y los plugins son códigos disponibles que podemos integrar en nuestras aplicaciones para añadir nuevas funcionalidades. Existen muchos plugins para JavaScript que nos pueden ayudar a lograr funcionalidades muy interesantes.

Veamos un ejemplo de cómo ocupar Plugins usando Chart Js para renderizar una gráfica interactiva.

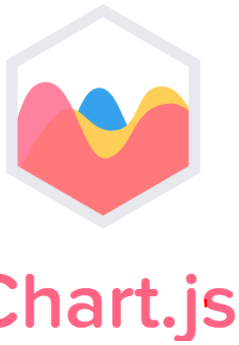

Imagen 13. Chart.js
Fuente: Desafío Latam

Vemos un ejemplo de cómo ocupar este Plugin agregando el siguiente CDN:

```
<script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
```

Y ocupando el siguiente endpoint: https://api.gael.cloud/general/public/sismos
Con este endpoint obtendremos los últimos 15 registros de sismos en Chile, y ésta será la data que vamos a graficar con Chart Js.

Lo siguiente será:

1. Agregar un encabezado y una etiqueta canva que será utilizada por el Plugin para renderizar la gráfica

```
<h1>Últimos Sismos en Chile</h1>
<div class="grafica">
  <canvas id="myChart"></canvas>
</div>
```

2. Crear una función para obtener y retornar la Data preparada que vamos a ocupar en nuestra gráfica

```
async function getAndCreateDataToChart() {
  const res = await
fetch("https://api.gael.cloud/general/public/sismos");
  const sismos = await res.json();

  const labels = sismos.map((sismo) => {
    return sismo.Fecha;
  });

  const data = sismos.map((sismo) => {
    const magnitud = sismo.Magnitud.split(" ")[0];
    return Number(magnitud);
  });

  const datasets = [
    {
      label: "Sismo",
      borderColor: "rgb(255, 99, 132)",
      data
    }
  ];
  return { labels, datasets };
}
```

3. Crear una función para renderizar la gráfica utilizando los datos que retorna la función anterior

```
async function renderGrafica() {
  const data = await getAndCreateDataToChart();
  const config = {
    type: "line",
    data
  };
  const myChart = document.getElementById("myChart");
  myChart.style.backgroundColor = "white";
  new Chart(myChart, config);
}

renderGrafica();
```

Ahora podemos observar en nuestro navegador cómo se renderiza nuestra gráfica con los últimos 15 sismos registrados en Chile:

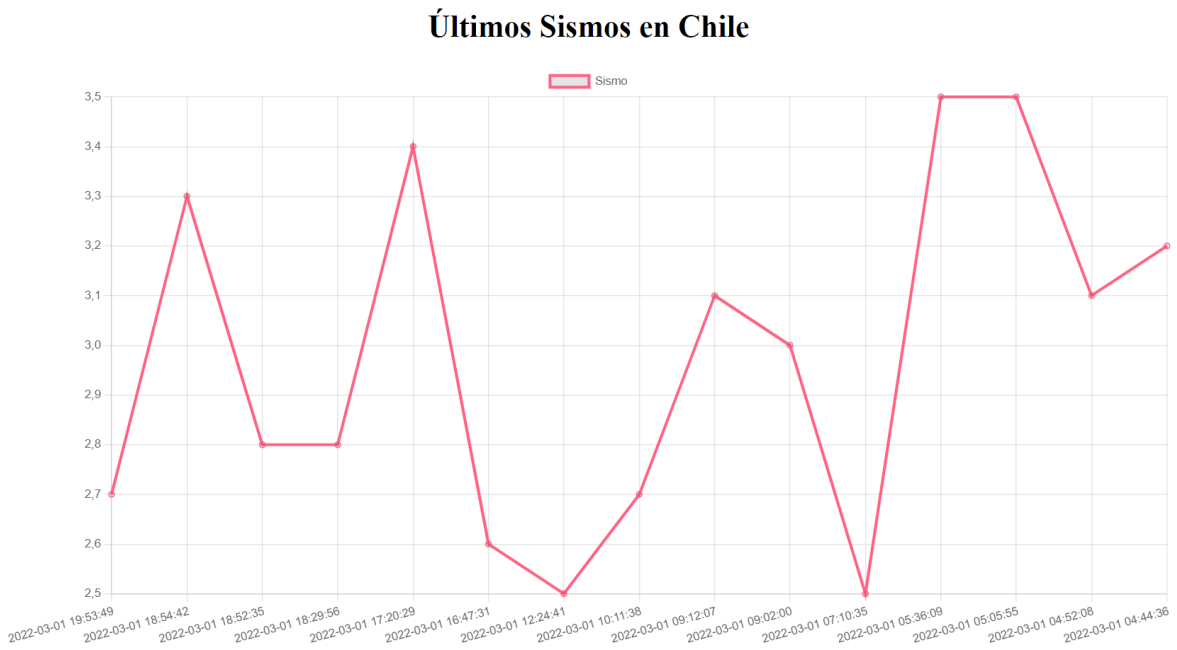

Imagen 14. Gráfica de Sismos
Fuente: Desafío Latam

Esta API de sismos entrega el valor de magnitud en diferentes unidades, por lo que en este ejemplo en particular por motivos solo demostrativos se estamos juntando valores que no deberían agruparse por no tener la misma unidad.

Objeto de configuración

Para entender mejor cómo se construye el objeto de configuración para la gráfica veamos otro ejemplo de gráfica donde mostraremos los valores de diferentes monedas nacionales e internacionales en pesos chilenos ocupando el siguiente endpoint de la API GAEL:

https://api.gael.cloud/general/public/monedas

Lo primero será entender que el objeto de configuración tiene la siguiente estructura:

```
{
  type: String,
  data: {
    labels: Array,
    datasets: [{
      label: String,
      backgroundColor: String,
      data: Array,
    }]
  }
}
```

En donde:

- **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. En el caso de ser de tipo *line* será el color de la línea.
- **data:** Es un arreglo de números que se tomarán como los valores que se insertarán en el eje vertical.

Entendiendo la estructura del objeto de configuración para la gráfica, ahora realicemos un ejercicio que grafique las diferentes monedas que nos entrega la API GAEL.

Para esto necesitamos indispensablemente una etiqueta *canvas* con un identificador que servirá para renderizar la gráfica.

HTML

```
<div>
  <canvas id="myChart"></canvas>
</div>
```

Ahora procedamos a escribir nuestro script siguiendo estos pasos:

1. Crea una función que obtenga las monedas de la API GEAL

```
async function getMonedas() {
  const endpoint = "https://api.gael.cloud/general/public/monedas";
  const res = await fetch(endpoint);
  const monedas = await res.json();
  return monedas;
}
```

2. Crea una función que prepare el objeto de configuración para la gráfica

```
function prepararConfiguracionParaLaGrafica(monedas) {
  // Creamos las variables necesarias para el objeto de configuración
  const tipoDeGrafica = "line";
  const nombresDeLasMonedas = monedas.map((moneda) => moneda.Codigo);
  const titulo = "Monedas";
  const colorDeLinea = "red";
  const valores = monedas.map((moneda) => {
    const valor = moneda.Valor.replace(",", ".");
    return Number(valor);
  });

  // Creamos el objeto de configuración usando las variables anteriores
  const config = {
    type: tipoDeGrafica,
    data: {
      labels: nombresDeLasMonedas,
      datasets: [
```

```
        {
          label: titulo,
          backgroundColor: colorDeLinea,
          data: valores
        }
      ]
    }
  };
  return config;
}
```

En esta función estamos creando todas las variables necesarias para la preparación del objeto de configuración.

En la creación de la variable *valores* es necesario cambiar las comas(“,”) por puntos (“.”) para poder ocupar el `Number()` y parsear el valor que originalmente viene en String.

3. Crea y ejecuta una función que renderice la gráfica ocupando las funciones anteriores

```
async function renderGrafica() {
  const monedas = await getMonedas();
  const config = prepararConfiguracionParaLaGrafica(monedas);
  const chartDOM = document.getElementById("myChart");
  new Chart(chartDOM, config);
}

renderGrafica();
```

Ahora ejecutemos al ejecutar nuestra aplicación tendremos la gráfica de las monedas:

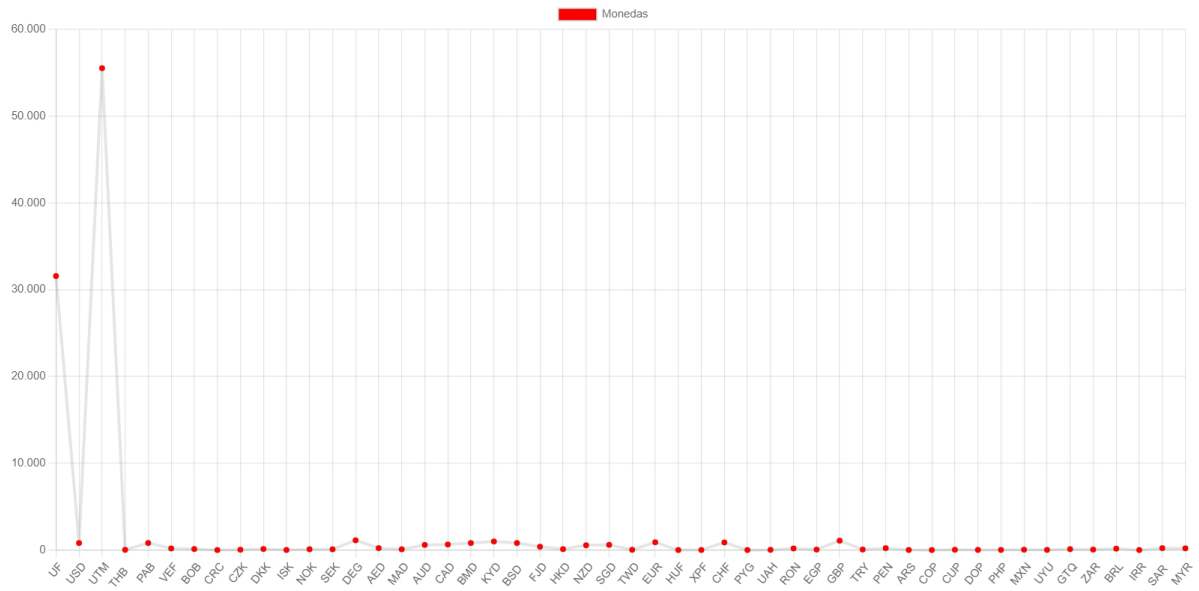

Imagen 15. Gráfica de monedas
Fuente: Desafío Latam

Actividad 5: Gráfica de estadísticas

Realicemos el siguiente ejercicio donde deberás graficar las estadísticas de Pikachú. Para esto será necesario usar fetch para consultar el siguiente endpoint:

- https://pokeapi.co/api/v2/pokemon/pikachu

Posteriormente preparar la data de las estadísticas para que sirva como conjunto de datos para Chart Js.

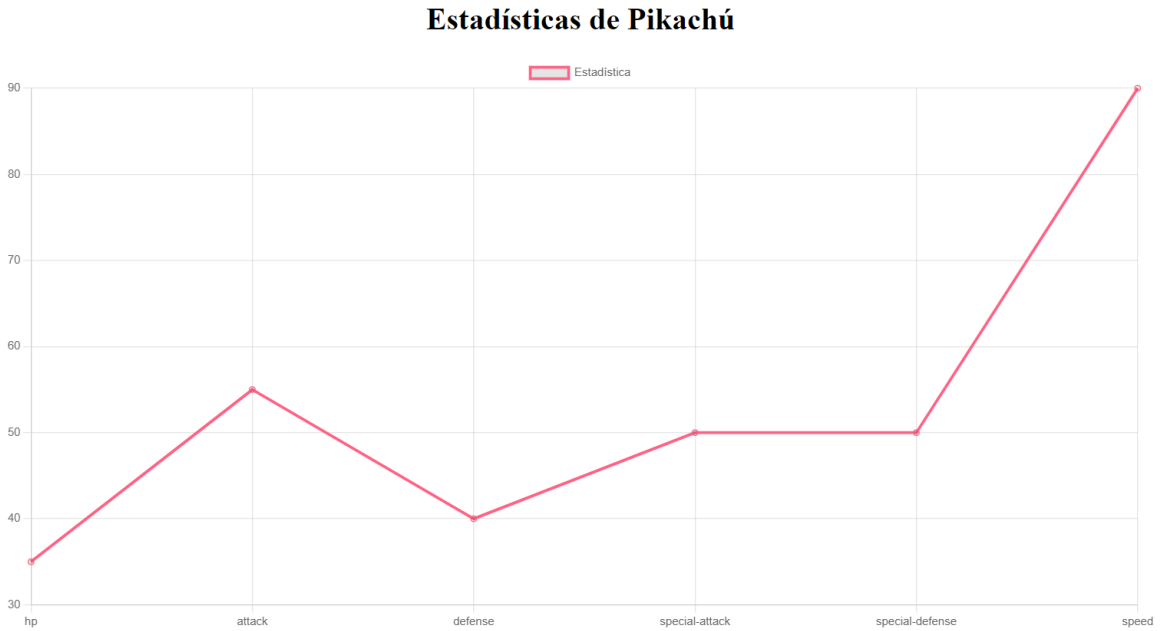

Imagen 12. Gráfica de stats de Pikachú
Fuente: Desafío Latam

Resumen

Hay funciones como fetch que devuelven promesas, el código que ejecutemos a continuación debe esperar que esta promesa se resuelva de forma de poder seguir trabajando. Para esperar que se resuelva ocuparemos await. (Existe otra opción con .then)

Las funciones *Async/Await* declaran que el bloque se ejecutará de forma asíncrona y se deberá esperar que los procesos con la palabra *await* por delante termine para continuar la ejecución del bloque.

La sentencia *Try Catch* nos ayuda a capturar errores que puedan ocurrir en un bloque de código.

En la web existen diferentes plugins de JavaScript que podemos integrar a nuestras aplicaciones para agregar nuevas funcionalidades

Chart Js es un plugin de JavaScript para la renderización de gráficas interactivas. Es necesario preparar los datos de la manera que el plugin defina para que la gráfica funcione correctamente.