1. JavaScript asincrónico
    1.1. JSON

- ¿Qué es un JSON?:

JSON (Javascript Object Notation) es un formato ligero de *intercambio de datos*. Se utiliza comúnmente para *transmitir datos* entre un **servidor** y una **aplicación web**. **JSON** es fácilmente legible por humanos y también fácilmente parseable por máquinas.

Un documento **JSON** consiste en pares **clave-valor**, donde las claves son *cadenas de texto* y los valores pueden ser *cadenas de texto*, *números*, *booleanos*, *objetos*, *matrices* o *nulos*. Estos pares **clave-valor** se agrupan en **objetos**, y los objetos pueden anidarse para representar estructuras más complejas.

```
{
        "nombre": "Juan",
        "edad": 30,
        "ciudad": "Ciudad Ejemplo",
        "casado": false,
        "hobbies": ["lectura", "programación"]
}
```

- ¿Para qué sirve un JSON?:

01. **JSON** es utilizado para transmitir información entre **servidores** y **clientes** en **aplicaciones web** y **servicios** *API*.
02. Funciona como formato de almacenamiento ligero y compresible, ya sea en sistemas de bases de datos *NoSQL* o archivos locales.
03. En arquitecturas *RESTful*, **JSON** es común para enviar y recibir datos entre clientes y servidores.
04. Facilita la comunicación entre sistemas heterogéneos gracias a su simplicidad y legibilidad.
05. Su sintaxis es una extensión de la notación de **objetos** en **JavaScript**, facilitando su integración en **aplicaciones web** basadas en este lenguaje.

    1.2. API REST

- ¿Qué es una API REST?:

Una **API REST** (Representational State Transfer) es un conjunto de **reglas** y **convenciones** para el diseño de *servicios web* que utilizan el protocolo **HTTP**. Está basada en los principios **REST**, una *arquitectura* que se centra en la *representación de recursos* y en las operaciones estándar sobre esos recursos.

- En una **API REST**, todo se considera un **recurso**, que puede ser un **objeto**, un **servicio** o incluso un **concepto abstracto**. Los recursos son identificados por **URLs**.
- Las *operaciones* sobre **recursos** se realizan mediante los *verbos* **HTTP** adecuados. Por ejemplo, **GET** para obtener información, **POST** para crear recursos, **PUT** y **PATCH** para actualizarlos, y **DELETE** para eliminarlos.
- **Sin Estado**: La comunicación entre el **cliente** y el **servidor** *no guarda información* sobre el **estado actual** del **cliente** en el **servidor** entre solicitudes. Cada solicitud

del **cliente** al **servidor** contiene *toda la información necesaria* para **entender** y **procesar** la solicitud.

- API REST CRUD:

Es un acrónimo que representa operaciones básicas realizadas sobre los datos en un sistema. Estas operaciones son fundamentales en el contexto de la programación y la gestión de bases de datos.

C. (Create) POST: Agrega un nuevo registro o recurso al sistema.
R. (Read) GET: Recuperar información existente en el sistema.
U. (Update) PUT: Modificar o actualizar un registro o recurso existente en el sistema.
D. (Delete) DELETE: Eliminar un registro o recurso existente en el sistema.

1.3. Callbacks

(Ver Carpeta)
Los *callbacks* son funciones que se pasan como parámetros de otra función que luego se invoca dentro de la función externa para completar algún tipo de rutina o acción. . Sirven para complementar la funcionalidad función que recibe el *callback*.

1.4. Funciones asíncronas

- Funciones asincrónicas:

Las funciones asíncronas en **JavaScript** son aquellas que permiten la ejecución de operaciones no bloqueantes mediante el uso de **promesas** o la estructura **async/await**. Permiten realizar operaciones que pueden llevar tiempo, como *solicitudes de red* o *lectura de archivos*, sin detener la ejecución del programa.

- ➔ Promesas
- ➔ Estructura async/await
- ➔ Observables

- Promesas:

Una promesa representa un **valor** que puede estar **disponible** *ahora, en el futuro o nunca*. Se utiliza comúnmente para manejar operaciones **asíncronas** como *solicitudes a servidores, lectura/escritura de archivos, entre otras*.

ESTADOS:

- ➔ **Pendiente (Pending)**: El estado inicial cuando la promesa está creada pero aún no se ha completado o rechazado.
- ➔ **Cumplida (Fulfilled)**: La operación asociada con la promesa se completó con éxito, y el resultado se hizo disponible.
- ➔ **Rechazada (Rejected)**: La operación asociada con la promesa fallo, y se proporciona una razón para el fallo.

Las promesas proporcionan una forma más legible y manejable de trabajar con código asíncrono en comparación con el uso de callbacks anidados. Además, permiten encadenar operaciones asíncronas de manera más clara mediante el método **.then()**.

```
let miPromesa = new Promise(function(resolve, reject) {
        // Simulando una tarea asíncrona
        setTimeout(function() {
                let exito = true; // Cambiar a false para simular un fallo
                if(exito) {
                        resolve('Éxito'); // La promesa se cumple
                } else {
                        reject('Fallo'); // La promesa se rechaza
                }
        }, 2000); // Simula que la operación tarda 2 segundos en completarse
});

miPromesa.then(function(valor) {
        console.log(valor); // Se ejecuta cuando la promesa se cumple
}).catch(function(razon) {
        console.log(razon); // Se ejecuta cuando la promesa falla
});
```

- Async/Await:

Son *palabras clave* que trabajan en conjunto para simplificar la sintaxis y hacer que el **código asíncrono** se vea como el **código síncrono**. **Async** se utiliza para declarar una función asíncrona, mientras que **Await** se utiliza dentro de una función **async** y espera a que una **promesa** se *resuelva* antes de continuar con la ejecución del código.

```
async function miFuncionAsincrona() {
        try {
                let resultado = await algunaOperacionAsincrona();
                console.log(resultado);
        } catch (error) {
                console.log(error);
        }
}
```

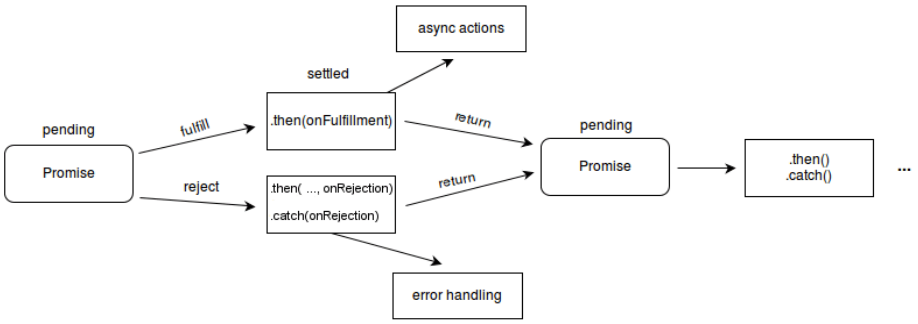
2. Promesas

2.1. Promesas - Parte 1

(Ver Carpeta)

new Promise

El objeto **Promise** representa la eventual finalización (o falla) de una operación asincrónica y su valor resultante.



Como práctica, se las suele invocar dentro de una variable junto a un callback o función anónima que recibe 2 parámetros que son, asu vez tambien, callbacks:

- **resolve**: se usa para marcar la promesa como cumplida (fulfilled) y proporciona un valor que se pasará a **.then()**. Una vez que se llama a **resolve()**, cualquier código dentro de **.then()** en la cadena de promesas se ejecutará.
- **reject**: se usa para marcar la promesa como rechazada (rejected) y pasar un error que se maneja en **.catch()**. Una vez que se llama a **reject()**, cualquier código dentro de **.catch()** se ejecutará.

Promise.then()
Método que retorna una promesa. Recibe 2 argumentos: funciones de callback para los casos de *success* y *reject*.

Promise.catch()
Método que retorna una promesa y solo se ejecuta en los casos en los que la promesa se marca como *reject*.

Promise.finally()
Método que devuelve una promesa. Cuando la promesa se resuelve, sea exitosa o rechazada, la función de callback especificada será ejecutada. Esto ofrece una forma de ejecutar código sin importar cómo se haya resuelto la promesa. Ayuda a evitar tener código duplicado tanto en el **.then()** como en el **.catch()**.

2.2. Promesas - Parte 2

(Ver Carpeta)
Se pasa en **.then()** un método externo.

2.3. Promise All

(Ver Carpeta)
Promise.all(iterable)
Método que devuelve una promesa que termina correctamente cuando todas las promesas en el argumento iterable han sido concluídas con éxito, o bien rechaza la petición con el motivo pasado por la primera promesa que es rechazada. Recibe como argumento un array de promesas.

3. Manejo de JS asincrónico

3.1. Primer acercamiento a Fetch

(Ver Carpeta)
fetch(url, opciones)
Método que lanza el proceso de solicitud de un recurso de la red. Esto devuelve una promesa que resuelve al objeto que representa la respuesta a la solicitud realizada, está disponible en prácticamente cualquier contexto desde el que se pueda necesitar solicitar un recurso.

- **url**:

Define el recurso que se quiere solicitar. Puede ser un string o un objeto Request (**new Request(url)**).

- **opciones**:

Objeto de opciones que contiene configuraciones para personalizar la solicitud.

3.2. Async/Await

(Ver Carpeta)

async function name() {...}
const name = async () => {...}

Declaración de funciones que define una función asíncrona, la cual devuelve un objeto de función asíncrona. Es una palabra clave que puede ser utilizada para definir funciones asíncronas dentro de expresiones.

await expresión

Operador que es usado para esperar a una Promise. Sólo puede ser usado dentro de una función **async** function.

Devuelven un objeto de tipo promesa.

3.3. Manejo de errores

(Ver Carpeta)

try…catch…finally

Instrucción que se compone de un bloque **try** y un bloque **catch**, un bloque **finally** o ambos. El código del bloque **try** se ejecuta primero y, si se produce una excepción, se ejecutará el código del bloque **catch**. El código del bloque **finally** siempre se ejecutará antes de que el flujo de control salga de la construcción.

La sentencia **try** siempre comienza con un bloque **try**. Después, debe haber un bloque **catch** o un bloque **finally**. También es posible tener bloques **catch** y **finally**. Esto nos da tres formas para la sentencia **try**:

- **try...catch**
- **try...finally**
- **try...catch...finally**

catch(razón)

El método **catch()** también retorna una promesa y solo se ejecuta en los casos en los que la promesa se marca como Reject. Se comporta igual que al llamar. El argumento que recibe es la razón del rechazo. La promesa devuelta por **catch()** es rechazada si lanza un error o retorna una promesa que a su vez se rechaza, de cualquier otra manera la promesa es resuelta.

throw expresión

Lanza una excepción definida por el usuario con una **expresión** a lanzar que especifica el valor de la excepción.

throw new Error(error);

Objetos que son lanzados cuando se producen errores de ejecución, y pueden usarse como objeto base para excepciones definidas por el usuario.

3.4. Apuntes de manejo de errores

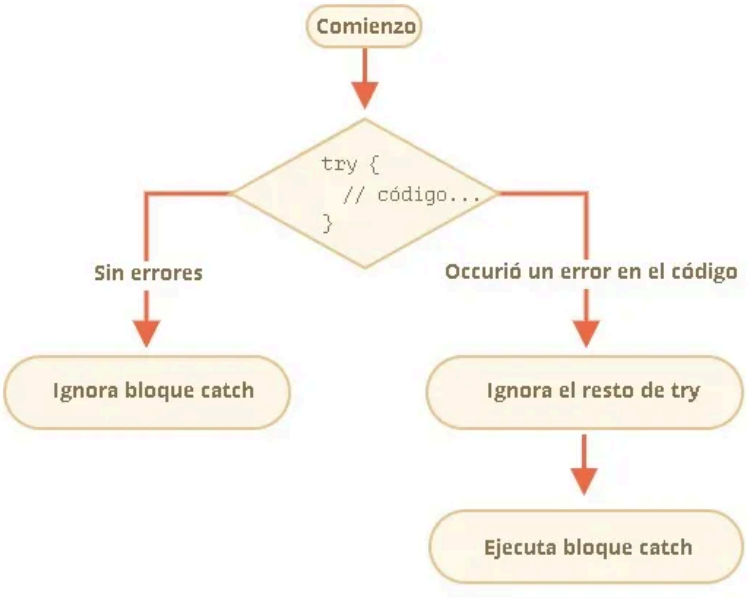
- Manejo de errores:

Los errores que se producen en un programa pueden ocurrir debido a nuestros descuidos, a una entrada inesperada del usuario, a una respuesta errónea del servidor, entre otras

razones. Por lo general, un script es interrumpido y se detiene cuando esto sucede. Pero podemos evitarlo con try...catch que nos permite “atrapar” errores para que el script pueda funcionar igualmente.

- La declaración try permite probar un bloque de código en busca de errores.
- La declaración catch permite manejar el error.
- La declaración throw permite crear errores personalizados.
- La declaración finally permite ejecutar código, después de intentar y capturar, independientemente del resultado.

- Flujo:



- Sintaxis:

```
try {
        Bloque de código del try.
} catch (error) {
        Bloque de código para manejar errores.
} finally {
        Bloque de código que se ejecuta independientemente del resultado del try/catch.
}
```

- Errores

Nombre del Error	Descripción
RangeError	Se ha producido un número “fuera de rango”.
ReferenceError	Ha ocurrido una referencia ilegal.
Error de sintaxis	Ha ocurrido un error de sintaxis.
Error de tecleado	Ha ocurrido un error de tipeo.
URIError	Se ha producido un error en un encodeURI().

4. Checkpoint de conocimientos
   - ¿Qué característica de las Promesas en JavaScript permite manejar múltiples operaciones asincrónicas?

Promise.all()

   - ¿Cuál es la principal ventaja de usar async/await en lugar de encadenamientos de promesas tradicionales?

async/await mejora la legibilidad y reduce la complejidad del código asincrónico.

   - ¿Cuál es el propósito principal de utilizar fetch en JavaScript?

Realizar solicitudes de red asincrónicas a recursos y APIs.

   - ¿Cómo se manejan los errores en una función asincrónica que utiliza async/await?

Utilizando un bloque try...catch para capturar los errores.