1. Introducción a APIs
    1.1. APIs

- ¿Qué es una API?

Significa Interfaz de Programación de Aplicaciones (Application Programming Interface). Es un conjunto de reglas y definiciones que permite que diferentes aplicaciones se comuniquen entre sí.

- **Interactividad**: Define como las aplicaciones deben interactuar entre sí.
- **Comunicación**: Establece reglas comunes para transferencia de datos.
- **Acceso**: Actúa como intermediario, permitiendo acceder a funciones sin exponer detalles internos.
- **Tipos**: Web (HTTP) y Bibliotecas (funciones de programación).
- **Simplificación**: Facilita la integración y acelera el proceso de desarrollo de software.
- **Interoperabilidad**: Permite la colaboración eficiente entre diferentes aplicaciones y plataformas.

- ¿Qué es una Web API?

Proporciona un conjunto de reglas y definiciones que permiten que aplicaciones y servicios web se comuniquen entre sí de manera estandarizada.

- **Protocolo**: Utiliza métodos HTTP (GET, POST, PUT, DELETE) para solicitar o enviar datos.
- **Formatos**: Suelen trabajar con formatos JSON para el intercambio de información.
- **Acceso Remoto**: Permite a las aplicaciones acceder a funcionalidades o datos a través de la red, generalmente a través de URLs.
- Interoperabilidad: Facilita la interoperabilidad entre sistemas y plataformas heterogéneas.
- **Autenticacion y Autorizacion**: Mecanismos de seguridad para el acceso a recursos (Tokens).
- **RESTful o GraphQL**: Las Web APIs suelen seguir arquitecturas RESTful, o GraphQL para organizar las operaciones y recursos de manera eficiente.

Algunos ejemplos: API de servicios en la nube (como las de Google, Facebook o Twitter), las API de plataformas de pago en línea, y muchas otras que permiten la integración de servicios y la construcción de aplicaciones web dinámicas.

- ¿Qué es una Browser API?

Se refiere a las interfaces de APIs proporcionadas por los navegadores web. Estas APIs permiten que los desarrolladores web accedan y utilicen funciones específicas del navegador para crear aplicaciones web interactivas y dinámicas.

- **Geolocalización**: Proporciona la ubicación del usuario a través del navegador.
- **Cámara y Micrófono**: Permite el acceso a dispositivos multimedia (grabación, filmación).
- **LocalStorage y SessionStorage**: Ofrece almacenamiento local en el navegador.
- **Fetch**: Facilita realizar solicitudes HTTP y gestionar respuestas.

- **WebSockets**: Permite la comunicación bidireccional en tiempo real entre el navegador y un servidor.
- **Audio**: Ofrece capacidades para procesar y manipular audio en aplicaciones web.

1.2. Geolocalización

(Ver Carpeta)

navigator.geolocation

Propiedad que devuelve un objeto (objeto Geolocation) que proporciona acceso web a la ubicación de un dispositivo.

*Callback: función que se pasa como parámetro de otra función.

navigator.geolocation.getCurrentPosition(éxito, error, opciones)

Método usado para obtener la ubicación actual del dispositivo.

- **éxito**: función de callback que tiene como único parámetro de entrada un objeto de tipo *GeolocationPosition*.
- **error**: parámetro opcional, función de callback que tiene como parámetro de entrada un objeto de tipo GeolocationPositionError.
- **opciones**: parámetro opcional, es un objeto de opciones de configuración:
  - **maximumAge**: indica el tiempo máximo en milisegundos que una posible ubicaciones será almacenada en caché.
  - **timeout**: representa el tiempo máximo en milisegundos en que el dispositivo tiene permitido recuperar la ubicación.
  - **enableHighAccuracy**: booleano que indica que a la aplicación le gustaría obtener el resultado más preciso posible.

GeolocationPosition:

Es una interfaz devuelta en el método de callback de **getCurrentPosition()**. Representa la posición del dispositivo.

- **.coords**: devuelve un objeto que define la ubicación actual.
- **.coords.longitude**: propiedad que retorna un número que representa la longitud de una posición geográfica, especificada en grados decimales.
- **.coords.latitude**: propiedad que retorna un número que representa la latitud de una posición geográfica, especificada en grados decimales.

1.3. Form Validation

(Ver Carpeta)

Element.checkValidity()

Método que devuelve un booleano que indica si el elemento cumple con alguna regla de validación de restricciones que se le aplique.

Validación de Restricciones (Constraint validation):

En HTML las restricciones son declaradas de 2 formas:

- Eligiendo el calor semánticamente más apropiado para el atributo del elemento **<input>** que crea automáticamente una restricción que verifica si su valor es válido:
  - **type="tipo"**: email, text, number, url, button, checkbox, color, date, file, password, search, submit, etc.

- Estableciendo valores en atributos relacionados con la validación:
    - **pattern**: expresiones regulares.
    - **min** y **max**: número, fecha o número y fecha.
    - **required**: no tiene valor, indica que debe tener un valor.
    - **step**: se usa para especificar incrementos de valor (por ejemplo: step=”10” para un type number, solo acepta incrementos de 10 en 10).
    - **minLength** y **maxLength**: es un número que indica la cantidad mínima y máxima de caracteres.

Element.validationMessage
Propiedad que retorna un string representando un mensaje que describe las restricciones de validación que no se cumplen.

1.4. Mensaje de error personalizado

(Ver Carpeta)

1.5. History API

(Ver Carpeta)
window.history.go(valor)
Método que carga una página específica del historial de sesiones. Se puede usar para avanzar o retroceder en el historial según el valor del parámetro en relación con la página actual. Un valor negativo lo desplaza hacia atrás, un valor positivo lo desplaza hacia adelante. Si no se pasa un valor o el valor es 0, tiene el mismo resultado que llamar a **location.reload()**.

window.location.reload()
Método que carga de nuevo la URL actual, como lo hace el botón de Refresh de los navegadores.

2. Almacenamiento

2.1. LocalStorage

(Ver Carpeta)
window.localStorage
Propiedad que permite acceder al objeto **localStorage**; los datos persisten almacenados entre las diferentes sesiones de navegación. l**ocalStorage** es similar a **sessionStorage**. La única diferencia es que, mientras los datos almacenados en **localStorage** no tienen fecha de expiración, los datos almacenados en **sessionStorage** son eliminados cuando finaliza la sesión de navegación - lo cual ocurre cuando se cierra la página.

.localStorage.setItem(clave, valor)
Método que agrega, o actualiza si ya existe, un objeto con una **clave** y un **valor** al objeto **Storage**.

.localStorage.setItem(clave)
Método que devuelve el valor de la **clave** especificada en el parámetro.

.localStorage.removeItem(clave)
Método que elimina el ítem con la **clave** especificada en el parámetro.

.localStorage.clear()
Método que remueve todos los ítems de **localStorage**

2.2. SessionStorage

(Ver Carpeta)
window.sessionStorage
La propiedad **sessionStorage** permite acceder a un objeto ***Storage*** asociado a la sesión actual. Es similar a **localStorage**, la diferencia es que la información es eliminada al finalizar la sesión de la página. La sesión de la página perdura mientras el navegador se encuentra abierto, y se mantiene por sobre las recargas y reaperturas de la página. Abrir una página en una nueva pestaña o ventana iniciará una nueva sesión, lo que difiere en la forma en que trabajan las cookies de sesión. Funciona igual que **localStorage**.\

2.3. Borrar o limpiar almacenamiento

(Ver Carpeta)

3. Consola

3.1. Consola - Parte 1

(Ver Carpeta)
Console
El objeto console provee acceso a la consola de depuración de los navegadores. Los detalles de cómo funciona varían de navegador en navegador. Puede ser accedido desde cualquier objeto global **window** en el ámbito de navegación a través de la propiedad **console**. Está expuesto como **window.console**, y puede ser referenciado como **console**.

console.log(valor, …);
Método que muestra un mensaje por consola con el valor o los valores especificados en sus argumentos.

Sustituciones de cadenas:
Cuando se pasa una cadena a uno de los métodos del objeto console que la acepta, se puede usar las siguientes sustituciones de cadena:

- **%o** ó **%O**: Muestra un objeto JavaScript. Haciendo clic sobre el nombre del objeto abre más información acerca del mismo en el inspector.
- **%d** ó **%i**: Muestra un entero.
- **%s**: Muestra una cadena.
- **%f**: Muestra un valor de punto flotante.

Estilizando la salida por consola:
console.log(“%cmensaje”, estilos);

- **%c**: aplica estilos CSS a la salida de la consola, el argumento estilos es un string con el formato CSS clave/valor.

console.assert(afirmación, valores);
Método que muestra un mensaje de error por consola si el primer argumento es falso.

console.clear();
Método que limpia o despeja la consola de cualquier impresión previa.

console.count(mensaje);
Método que registra el número de veces que se llama a count(). Esta función toma como argumento opcional una etiqueta, por lo que devuelve la etiqueta y el número de veces que se llamó, si no se pasa ningún argumento, solo retorna el número de veces.

console.error(error);
Método que muestra un mensaje de error con la consola.

console.info(mensaje);
Método que emite un mensaje informativo a la Consola Web. En Firefox y Chrome, se muestra un pequeño ícono "i" junto a estos elementos en el registro de la Consola Web.

console.warn(mensaje);
Método que emite un mensaje de advertencia, sin que llegase a ser un error.

3.2. Consola - Parte 2

(Ver Carpeta)
console.group(etiqueta);
Método que crea un grupo en línea en el registro de la consola, lo que genera una sangría o indentación de nivel adicional a todos los mensajes posteriores hasta que se llame al método **console.groupEnd()**. El argumento es opcional y marca el grupo con dicho valor. Pueden anidarse.

console.groupEnd();
Método que finaliza el grupo inicializado con **console.group(etiqueta)**.

console.groupCollapsed(etiqueta);
Método que crea un nuevo grupo contraído en línea en la consola. El usuario deberá usar el botón *triángulo de expansión* que se encuentra junto a él para expandirlo y revelar las entradas creadas en el grupo. Se debe llamar a **console.groupEnd()** para volver al grupo principal.

console.table(datos, columnas);
Método que muestra datos tabulares como una tabla. Toma un argumento obligatorio: **datos**, que debe ser un *array* o un *objeto*, y un parámetro opcional: **columnas**.

Muestra el argumento **datos** como una tabla en la consola. Cada elemento en el array (o propiedad enumerable si data es un objeto) será una fila en la tabla.

La primera columna de la tabla se identificará como **index**. Si **datos** es un *array*, sus valores serán los índices del *array*. Si es un objeto, sus valores serán los nombres de las propiedades.

console.time(etiqueta);
Método que inicia un temporizador que se puede usar para realizar un seguimiento de la duración de una operación. Recibe como argumento un nombre único que, cuando se llame a **console.timeEnd(etiqueta)** con el mismo nombre, el navegador generará el tiempo, en milisegundos, transcurrido desde que se inició el temporizador.

console.timeEnd(etiqueta)
Método que detiene un temporizador que haya sido establecido previamente con **console.time(etiqueta)**.

console.trace()
Método que muestra una lista de seguimientos (stack trace), es decir, muestra la secuencia de funciones que llevaron a la ejecución del código en ese punto. Sirve para depurar código mostrando dónde y cómo se llamó una función, útil cuando quieres rastrear el origen de un error o entender el flujo de ejecución.

4. Timers
    4.1. Timers

(Ver Carpeta)
setTimeout(callback, milisegundos, …);
Método global que establece un temporizador que ejecuta una función o una pieza de código específica (**callback**) una vez que expira el temporizador (**milisegundos**). Puede recibir más parámetros opcionales (**...**) que se pasarán a la función de **callback**.

setInterval(callback, milisegundos, …);
Método que llama a una función o ejecuta un fragmento de código (**callback**) de forma reiterada, con un retardo de tiempo fijo (**milisegundos**) entre cada llamada. Puede recibir más parámetros opcionales (**...**) que se pasarán a la función de **callback**.

clearInterval(intervalID);
Método que cancela una acción reiterativa que se inició mediante una llamada a **setInterval()**, es decir, se creó un **setInterval()**, se lo guardó en una variable y luego se la pasa como argumento **intervalID**.

clearTimeout(timeoutID);
Método que borra el retraso que se inició mediante una llamada a **setTimeout()**, es decir, se creó un **setTimeout()**, se lo guardó en una variable y luego se la pasa como argumento **timeoutID**.

5. Checkpoint de conocimientos

- ¿Cuál es la principal diferencia entre localStorage y sessionStorage?

localStorage almacena datos sin fecha de expiración, mientras que sessionStorage los borra al cerrar la pestaña o navegador.

- ¿Qué método de la Web Storage API es más adecuado para almacenar datos que deben persistir entre sesiones de navegación?

localStorage.setItem()

- ¿Para qué se utiliza la Geolocalización API en el desarrollo web?

Para obtener la ubicación geográfica del usuario.

- ¿Qué objeto se utiliza en JavaScript para interactuar con el historial de navegación del navegador?

window.history

- ¿Cuál es el propósito de usar setTimeout() en JavaScript?

Ejecutar una función o un fragmento de código después de un retraso especificado.