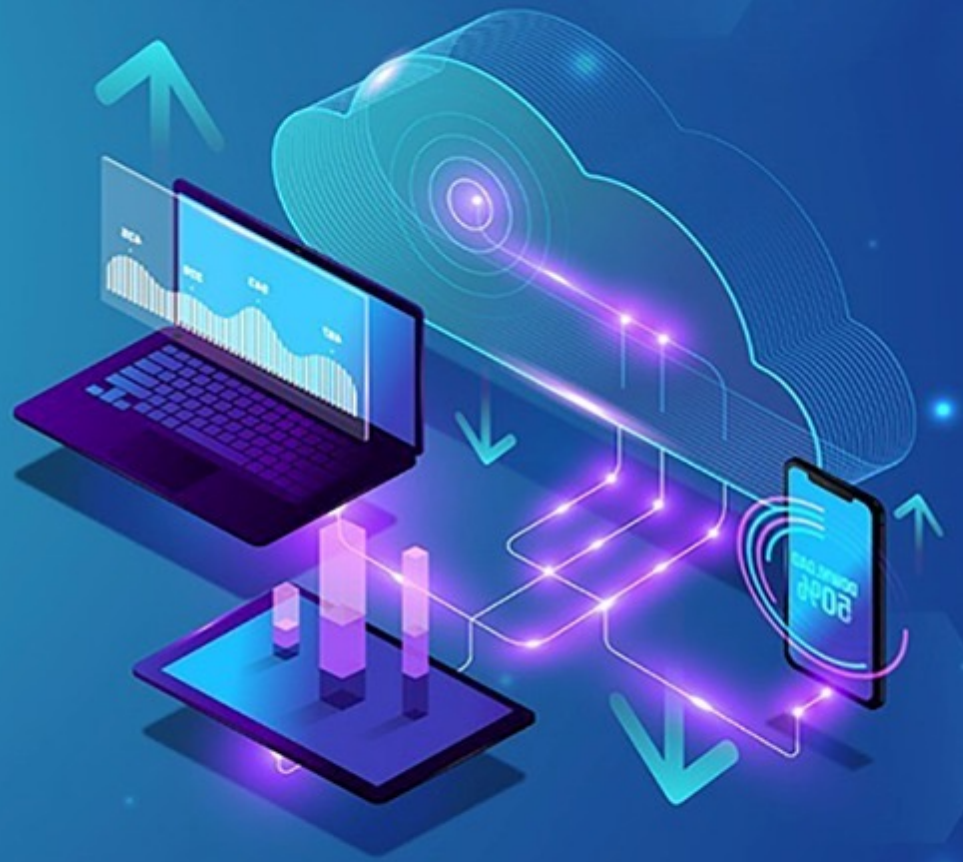
Proyecto BID-MINEC

PROYECTO UPSKILLING & RESKILLING

Orientado a la reconversión y mejora de habilidades digitales y tecnológicas



FORMÁS
CENTRO DE FORMACIÓN AVANZADO PARA PROFESIONALES

FEDISAL
Fundación para la Educación Integral Salvadoreña

BID

TECH SKILLS COUNCIL

GOBIERNO DE EL SALVADOR

MINISTERIO DE ECONOMÍA

FrontEnd básico-intermedio con JavaScript

Framework's web

Facilitador: Iván Alvarado

Contenido

1. JSON
2. LocalStorage
3. Peticiones
4. JQuery
5. JQueryUI

1. JSON.

- JavaScript Object Notation (JSON) es un formato basado en texto estándar para representar datos estructurados en la sintaxis de objetos de JavaScript.
- Es comúnmente utilizado para transmitir datos en aplicaciones web (por ejemplo: enviar algunos datos desde el servidor al cliente, así estos datos pueden ser mostrados en páginas web, o vice versa).
- JSON es un formato de datos basado en texto que sigue la sintaxis de objeto de JavaScript, popularizado por Douglas Crockford. Aunque es muy parecido a la sintaxis de objeto literal de JavaScript, puede ser utilizado independientemente de JavaScript, y muchos entornos de programación poseen la capacidad de leer (convertir; parsear) y generar JSON.

1. JSON.

- Los JSON son cadenas útiles cuando se quiere transmitir datos a través de una red.
- Debe ser convertido a un objeto nativo de JavaScript cuando se requiera acceder a sus datos.
- Ésto no es un problema, dado que JavaScript posee un objeto global JSON que tiene los métodos disponibles para convertir entre ellos.
- Un objeto JSON puede ser almacenado en su propio archivo, que es básicamente sólo un archivo de texto con una extension .json, y una MIME type (en-US) de application/json.

1. JSON.

Estructura del JSON

- Como se describió previamente, un JSON es una cadena cuyo formato recuerda al de los objetos literales JavaScript.
- Es posible incluir los mismos tipos de datos básicos dentro de un JSON que en un objeto estándar de JavaScript cadenas, números, arreglos, booleanos, y otros literales de objeto.
- Esto permite construir una jerarquía de datos, como la que se muestra en la siguiente diapositiva:

1. JSON.

app.json

```json
{
  "expo": {
    "name": "usolink_v4",
    "slug": "usolink_v4",
    "version": "1.0.0",
    "orientation": "portrait",
    "icon": "./assets/icon.png",
    "splash": {
      "image": "./assets/splash.png",
      "resizeMode": "contain",
      "backgroundColor": "#ffffff"
    },
    "updates": {
      "fallbackToCacheTimeout": 0
    },
    "assetBundlePatterns": [
      "**/*"
    ],
    "ios": {
      "supportsTablet": true,
      "bundleIdentifier": "edu.usonsonate.USOLinkV4",
      "buildNumber": "1.0.0"
    },
    "web": {
      "favicon": "./assets/favicon.png"
    }
  }
}
```

1. JSON.

Arreglos como JSON

- Anteriormente se mencionó que el texto JSON básicamente se parece a un objeto JavaScript, y esto es en gran parte cierto. La razón de esto es que un arreglo es también un JSON válido, por ejemplo:

```
[
  {
    "name": "Molecule Man",
    "age": 29,
    "secretIdentity": "Dan Jukes",
    "powers": [
      "Radiation resistance",
      "Turning tiny",
      "Radiation blast"
    ]
  },
  {
    "name": "Madame Uppercut",
    "age": 39,
    "secretIdentity": "Jane Wilson",
    "powers": [
      "Million tonne punch",
      "Damage resistance",
      "Superhuman reflexes"
    ]
  }
]
```

1. JSON.

- Array

Es un conjunto de valores ordenados. Un Array comienza con "[" y termina con "]" y los valores están separados por comas. Por ejemplo,

```
var userlist = [{"user":{"name":"Manas","gender":"Male","birthday":"1987-8-

{"user":{"name":"Mohapatra","Male":"Female","birthday":"1987-7-7"}}]
```

- String

Es cualquier cantidad de conjunto de caracteres Unicode que se incluye con comillas. Utiliza la barra invertida para identificar las comillas.

```
var userlist = "{\"ID\":1,\"Name\":\"Manas\",\"Address\":\"India\"}"
```

1. JSON.

Clase JSON en JavaScript

- Método stringify : transforma un objeto en una cadena con formato JSON.
- Método parse : transforma una cadena JSON en un objeto
- Ejemplos:

```
var arreglo = new Array(….);
var lst = JSON.stringify(arreglo);
….
var nuevoArreglo = JSON.parse(lst);
```

1. JSON.

Resumen JSON

- JSON es sólo un formato de datos — contiene sólo propiedades, no métodos.
- JSON requiere usar comillas dobles para las cadenas y los nombres de propiedades. Las comillas simples no son válidas.
- Una coma o dos puntos mal ubicados pueden producir que un archivo JSON no funcione.
- Se debe ser cuidadoso para validar cualquier dato que se quiera utilizar (aunque los JSON generados por computador tienen menos probabilidades de tener errores, mientras el programa generador trabaje adecuadamente). Es posible validar JSON utilizando una aplicación como JSONLint.

1. JSON.

Resumen JSON

- JSON puede tomar la forma de cualquier tipo de datos que sea válido para ser incluido en un JSON, no sólo arreglos u objetos.
- Así, por ejemplo, una cadena o un número único podrían ser objetos JSON válidos.
- A diferencia del código JavaScript en que las propiedades del objeto pueden no estar entre comillas, en JSON, sólo las cadenas entre comillas pueden ser utilizadas como propiedades.

2. LocalStorage

localStorage

```
function getData() {
 let data = localStorage.getItem("data");
 if (!data) {
   return fetch("/data").then(response => {
     data = response.data.data;
     localStorage.setItem("data", response.data.data);
     return data;
   });
 }
 return data;
}
```

2. LocalStorage

- La propiedad de sólo lectura localStorage permite acceder al objeto localStorage; los datos persisten almacenados entre de las diferentes sesiones de navegación.
- localStorage es similar a sessionStorage (en-US). 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 sesion de navegación lo cual ocurre cuando se cierra la página.
- Con sessionStorage (en-US) los datos persisten sólo en la ventana/tab que los creó, mientras que con localStorage los datos persisten entre ventanas/tabs con el mismo origen.
- Las claves y los valores son siempre cadenas de texto (ten en cuenta que, al igual que con los objetos, las claves de enteros se convertirán automáticamente en cadenas de texto).

2. LocalStorage

Sintaxis:

```
var almacenamiento = window.localStorage;
```

Valor

Un objeto Storage que se puede utilizar para acceder al espacio de almacenamiento local del origen actual.

Excepciones

SecurityError

La solicitud viola una decisión de política, o el origen no es una tupla válida de protocolo/host/puerto (esto puede suceder si el origen usa el protocolo file: o data:, por ejemplo). Por ejemplo, el usuario puede tener su navegador configurado a fin de denegar el permiso para conservar datos al origen especificado.

2. LocalStorage

Métodos:

- setItem(clave, valor) : Permite almacenar la clave en el localStorage asignando el valor especificado como parámetro
- getItem(clave) : devuelve el valor especificado en clave que anteriormente debe haberse guardado en el localStorage.
- removeItem(clave) : Elimina del localStorage la clave especificada como parámetro.
- clear() : Elimina todas las claves almacenadas en el localStorage
- key(indice) : Devuelve el nombre de la n-ésima clave, o nulo si n es mayor o igual que el número de pares clave/valor.

3. Peticiones asíncronas (Fetch y Ajax) y promesas

- Un navegador, durante la carga de una página, suele realizar múltiples peticiones HTTP a un servidor para solicitar los archivos que necesita renderizar en la página.
- Es el caso de, en primer lugar, el documento .html de la página (donde se hace referencia a múltiples archivos) y luego todos esos archivos relacionados: los ficheros de estilos .css, las imágenes .jpg, .png, .webp u otras, los scripts .js, las tipografías .ttf, .woff o .woff2, etc.

3. Peticiones asíncronas (Fetch y Ajax) y promesas

Que es una peticion HTTP:

- Una petición HTTP es como suele denominarse a la acción por parte del navegador de solicitar a un servidor web un documento o archivo, ya sea un fichero .html, una imagen, una tipografía, un archivo .js, etc.
- Gracias a dicha petición, el navegador puede descargar ese archivo, almacenarlo en un caché temporal de archivos del navegador y, finalmente, mostrarlo en la página actual que lo ha solicitado.

3. Peticiones asíncronas (Fetch y Ajax) y promesas

Peticiones HTTP mediante AJAX:

- Con el tiempo, aparece una nueva modalidad de realizar peticiones, denominada AJAX (Asynchronous Javascript and XML). Esta modalidad se basa en que la petición HTTP se realiza desde Javascript, de forma transparente al usuario, descargando la información y pudiendo tratarla sin necesidad de mostrarla directamente en la página.
- Esto produce un interesante cambio en el panorama que había entonces, puesto que podemos hacer actualizaciones de contenidos de forma parcial, de modo que se actualice una página «en vivo», sin necesidad de recargar toda la página, sino solamente actualizado una pequeña parte de ella, pudiendo utilizar Javascript para crear todo tipo de lógica de apoyo.

3. Peticiones asíncronas (Fetch y Ajax) y promesas

Originalmente, a este sistema de realización de peticiones HTTP se le llamó AJAX, donde la X significa XML, el formato ligero de datos que más se utilizaba en aquel entonces.

Actualmente, sobre todo en el mundo Javascript, se utiliza más el formato JSON, aunque por razones fonéticas evidentes (y probablemente evitar confundirlo con una risa) se sigue manteniendo el término AJAX.

Existen varias formas de realizar peticiones HTTP mediante AJAX, pero las principales suelen ser XMLHttpRequest y fetch (nativas, incluidas en el navegador por defecto), además de liberías como axios o superagent:

3. Peticiones asíncronas (Fetch y Ajax) y promesas

Método	Descripción
`XMLHttpRequest`	Se suele abreviar como `XHR`. El más antiguo, y también el más verbose. Nativo.
`fetch`	Nuevo sistema nativo de peticiones basado en promesas. Sin soporte en IE.
Axios	Librería basada en promesas para realizar peticiones en Node o navegadores.
superagent	Librería para realizar peticiones HTTP tanto en Node como en navegadores.
frisbee	Librería basada en fetch. Suele usarse junto a React Native.

3. Peticiones asíncronas (Fetch y Ajax) y promesas

XMLHttpRequest

```
const solicitud = new XMLHttpRequest();
solicitud.open('GET', 'https://mdn.github.io/learning-area/javascript/oojs/json/superheroes.json');
solicitud.responseType = 'json';
solicitud.send();
solicitud.onload = function(){
    cont = solicitud.response;
    console.log(cont)
}
```

Fetch

```
fetch('https://mdn.github.io/learning-area/javascript/oojs/json/superheroes.json')
.then( res => res.json())
.then( val => console.log(val));
```

Servicio público de información json

https://mdn.github.io/learning-area/javascript/oojs/json/superheroes.json

3. Peticiones asíncronas (Fetch y Ajax) y promesas

Que es una promesa

El objeto Promise (Promesa) representa un valor que puede estar disponible ahora, en el futuro o nunca.

Se utiliza cuando estamos frente a un código asíncrono. Ya que, valga la redundancia, este objeto "nos promete" que devolverá un valor en una línea de tiempo presente o futura y recuerda el contexto en donde se ejecuta, es decir, sabe con precisión en qué punto se ha de resolver un valor o lanzar un error.

Además, en contraposición a los callback, permite eliminar la dependencia entre funciones y ahorra el temido callback hell, funciones que dependen de funciones a través de callback hasta el punto de generar una cadena de dependencias en cascada difícil de seguir.

3. Peticiones asíncronas (Fetch y Ajax) y promesas

Que es una promesa

Siendo más concretos una promesa es un objeto que guarda una operación asíncrona.

Esta operación (una llamada AJAX, un evento del navegador como el click de un botón, una llamada a una función programada para el futuro) tiene una duración que desconocemos, pero al crear una promesa se nos genera un objeto con el que podemos trabajar

```
var promise = new Promise(function(resolve, reject) {

  function sayHello() {
    resolve('Hello World!')
  }

  setTimeout(sayHello, 10000)

})

console.log(promise)
```

3. Peticiones asíncronas (Fetch y Ajax) y promesas

En el ejemplo la variable promise es un objeto y se puede operar con él independientemente de que hasta dentro de 10 segundos no nos devuelva un valor y contenga el string 'Hello world!'.

Esto permite tener mayor control. Antes, la ejecución de nuestra app dependía de una llamada asíncrona que no sabíamos si iba o no iba a acabar.

Con las promesas seguimos trabajando independientemente de lo que pase con la llamada asíncrona. Además, contamos con la ventaja de que si el código lanza una excepción dentro de la Promesa, ésta la captura y la devuelve con la función 'catch'.

```
let promesa = new Promise((response, reject) => {
    if (true) {
      resolve('¡Ha funcionado!');
    } else {
      reject('Hay un error');
    }
});

promesa
    .then((response) => {
      console.log('Response:', response);
    })
    .catch((error) => {
      console.log('Error:', error);
    })
```

3. Peticiones asíncronas (Fetch y Ajax) y promesas

Estado de las promesas

A pesar de que se dice que las promesas tienen tres estados cuando se ejecutan, lo cierto es que en realidad tienen dos, uno inicial y uno final:

pending (pendiente)

fullfilled (resuelta exitosamente)

rejected (rechazada)

Inicialmente, una promesa tiene el estado pending, estado que tendrá hasta que la promesa se haya resuelto mediante resolve o haya ocurrido un error (reject).

Cuando una promesa alcanza uno de estos dos estados, ya no puede realizar transiciones a otro. Esto significa que no va a tener dos estados a la vez. No puede estar resuelta y rechazada al mismo tiempo. Una vez que una promesa tiene un estado, éste no cambia.

3. Peticiones asíncronas (Fetch y Ajax) y promesas

Estado de las promesas

En el ejemplo se crea dos setTimeOut, uno con 2 segundos en el que se ejecuta el resolve y otro con 3 segundos, en el que se llama a reject. Encima de los dos, se coloca un console log para observar el momento en el que la promesa está pendiente. En este caso, se va a ejecutar primero el setTimeOut de resolve y aunque reject no está en ninguna condición, no se ejecutará porque setTimeOut ya ha llamado a resolve.

```
let promesa = new Promise((response, reject) => {
    console.log('Promesa pendiente');
    setTimeout(() => {
      resolve('Promesa resuelta');
    }, 2000);

    setTimeout(() => {
      reject('Promesa rechazada');
    }, 3000);
});

promesa
    .then((response) => {
      console.log('Response:', response);
    })
    .catch((error) => {
      console.log('Error:', error);
    })
```

3. Peticiones asíncronas (Fetch y Ajax) y promesas

Promesa con parámetros

La promesa puede hacerse dinámica. En este caso se tiene que pasar argumentos para "personalizarla". Basta con crear una función que reciba esos parámetros, meter la promesa dentro y devolver su instancia.

```
function randomDelayed(max = 10, expected = 5, delay =  1000) {
  return new Promise((resolve, reject) => {
    const number = Math.floor(
      Math.random() * max)
    );

    setTimeout(
      () => number > expected
        ? resolve(number)
        : reject(new Error('número menor al esperado'));
      delay
    );
  });
}

randomDelayed(100, 75, 2500)
    .then(number => console.log(number))
    .catch(error => console.error(error));
```

3. Peticiones asíncronas (Fetch y Ajax) y promesas

Regresando al fetch

Una llamada al método fetch devuelve una Promise. Este objeto devuelto permite usar el método then. El método then, permite ejecutar un código cuando la promesa haya concluído.

Ejemplo de uso fetch:

```
fetch('https://mdn.github.io/learning-area/javascript/oojs/json/superheroes.json')
.then( res => res.json())
.then( val => console.log(val));
```

1. El código hace una petición a una API pública de superheroes.
2. Esta API devuelve una promesa. Cuando se haya cumplido se llamará al método json() que devuelve otra promesa que convierte la respuesta de la petición (texto plano) en un objeto de Javascript.
3. El segundo y último then procesa la promesa devuelta por el método .json() y hace un console.log para imprimirla en la consola.

3. Peticiones asíncronas (Fetch y Ajax) y promesas

Async y await

A continuación se realiza la misma petición del ejemplo anterior, pero ahora en lugar de utilizar la api de Promise (método *then*) para procesar la respuesta se utilizan los modificadores **async** y **await**.

```
<script>

    async function getPokemon() {
        const result = await fetch('https://pokeapi.co/api/v2/pokemon/ditto/');
        const res = await result.json();
        console.log(res);
    };
    getPokemon();

</script>
```

- El modificador **await** siempre debe estar dentro de una función con el modificador **async**, o daría error. Este modificador hace que hasta que no termine la promesa que le hemos pasado, las siguientes líneas de código no se ejecuten.
- El modificador **async** convierte la función en una Promise.

3. Peticiones asíncronas (Fetch y Ajax) y promesas

Async y await

```
<script>
    const b = () => new Promise(resolve => {
        setTimeout(() => {
            console.log('cosas');
            resolve();
        }, 3000)
    })

    async function a() {
        await b();
        console.log('llegamos');
    }

    a();
</script>
```

3. Peticiones asíncronas (Fetch y Ajax) y promesas

Async y await

A continuación se realiza la misma petición del ejemplo anterior, pero ahora en lugar de utilizar la api de Promise (método *then*) para procesar la respuesta se utilizan los modificadores **async** y **await**.

```
async function Procesar(){
    let resp = await fetch('https://mdn.github.io/learning-area/javascript/oojs/json/
        superheroes.json');
    return await resp.json();
}
Procesar().then(x => console.log(x));
```