Javascript

Semana 5.2

Índice

Resultado de aprendizaje, contenido y criterios de evaluación

Resultado de aprendizaje
RA2. Escribe sentencias simples, aplicando la sintaxis del lenguaje y verificando su ejecución sobre navegadores web.
Contenido
1 - Variables. 2 - Tipos de datos. Conversiones. 3 - Asignaciones. 4 - Operadores 5 - Comentarios al código. 6 - Sentencias. 7 - Decisiones. 8 - Bucles.
Criterios de evaluación
a) Se ha seleccionado un lenguaje de programación de clientes web en función de sus posibilidades b) Se han utilizado los distintos tipos de variables y operadores disponibles en el lenguaje. c) Se han identificado los ámbitos de utilización de las variables d) Se han reconocido y comprobado las peculiaridades del lenguaje respecto a las conversiones entre distintos tipos de datos e) Se han utilizado mecanismos de decisión en la creación de bloques de sentencias. f) Se han utilizado bucles y se ha verificado su funcionamiento g) Se han añadido comentarios al código

Resultado de aprendizaje
RA3. Escribe código, identificando y aplicando las funcionalidades aportadas por los objetos predefinidos del lenguaje.
Contenido
2 - Interacción con el navegador. Objetos predefinidos asociados 3 - Generación de texto y elementos HTML desde código 4 - Gestión de la apariencia de la ventana. 6 - Interacción con el usuario.
Criterios de evaluación
a) Se han identificado los objetos predefinidos del lenguaje d) Se han generado textos y etiquetas como resultado de la ejecución de código en el navegador. e) Se han escrito sentencias que utilicen los objetos predefinidos del lenguaje para interactuar con el usuario

Resultado de aprendizaje
R4. Programa código para clientes web analizando y utilizando estructuras definidas por el usuario
Contenido
2 - Llamadas a funciones. Definición de funciones 3 - Matrices (arrays). 4 - Operaciones agregadas: filtrado, reducción y recolección 5 - Creación de objetos. 6 - Definición de métodos y propiedades
Criterios de evaluación
b) Se han creado y utilizado funciones definidas por el usuario. c) Se han reconocido las características del lenguaje relativas a la creación y uso de matrices (arrays). d) Se han creado y utilizado matrices (arrays). e) Se han utilizado operaciones agregadas para el manejo de información almacenada en colecciones g) Se ha creado código para definir la estructura de objetos. h) Se han creado métodos y propiedades i) Se ha creado código que haga uso de objetos definidos por el usuario

Resultado de aprendizaje
RA5. Desarrolla aplicaciones web interactivas integrando mecanismos de manejo de eventos
Contenido
1 - Gestión de eventos. Tipos. 5 - Expresiones regulares.
Criterios de evaluación
b) Se han identificado las características del lenguaje de programación relativas a la gestión de los eventos d) Se ha creado un código que capture y utilice eventos f) Se han validado formularios web utilizando eventos g) Se han utilizado expresiones regulares para facilitar los procedimientos de validación

Resultado de aprendizaje
RA6. Desarrolla aplicaciones web analizando y aplicando las características del modelo de objetos del documento
Contenido
1 - El modelo de objetos del documento (DOM). 2 - Objetos del modelo. Propiedades y métodos 3 - Acceso al documento desde código 4 - Programación de eventos
Criterios de evaluación
a) Se ha reconocido el modelo de objetos del documento de una página web c) Se ha creado y verificado un código que acceda a la estructura del documento d) Se han creado nuevos elementos de la estructura y modificado elementos ya existentes. e) Se han asociado acciones a los eventos del modelo

Semana 6. Promesas y async/await

6.1. Ejemplo guiado

Comenzamos con un ejemplo guiado.

Ejemplo guiado

Crear una app que obtenga usuarios de una API pública y los muestre en el DOM. La API que usaremos será https://jsonplaceholder.typicode.com/users.

1. *Crearemos una estructura html añadiendo un div para mostrar los usarios y un botón que al pulsar lea la API y muestre los usuarios en el div.*

```
<!DOCTYPE html>
<html lang="es">
  <head>
    <title>Ejemplo guiado de API</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <script type="text/javascript" src="scripts/API.js" defer></script>
  </head>
  <body>
    <div id="usuarios"></div>
    <button id="cargar">Cargar usuarios</button>
  </body>
  <foot>
  </foot>
</html>
```

El script que vamos a desarrollar estará en un fichero extrerno y lo llamamos desde el index.html dentro de la etiqueta <head> con <script>. Los atributos que definimos son:

- *type: el tipo de fichero*
- *src: la ruta relativa de dónde se encuentra el fichero, en nuestro caso en la carpeta scripts*
- *defer: palabra reservada que le dice al navegador que cargue el fichero en backgroud y que esté disponible cuando el html esté completamente cargado para que el código esté disponible.*

2. *En el fichero API.js crearemos la función “obtenerUsuarios”:*

```
function obtenerUsuarios() {
  fetch("https://jsonplaceholder.typicode.com/users")
    .then(response => response.json())
    .then(data => mostrarUsuarios(data))
    .catch(error => console.error("Error:", error));
}
```

Comentemos que hace la función:

- *fetch: trae el contenido de la API y la retorna en la promesa “responde”.*
- *response => response.json(): en response se guarda la promesa en formato json.*
- *data => mostrarUsuarios(data): la promesa anterior del json llamará a la función mostrarUsuarios() pasándole la promesa.*
- *catch(): muestra el error que se pueda originar.*

Para comprender a fondo el uso de fetch() lee el anexo ANEXO VII. Asincronía, Promesas, async/await, Ajax. fetch() y Axios .

3. *En el fichero API.js crearemos la función “mostrarUsuarios”:*

```
function mostrarUsuarios(usuarios) {
  const contenedor = document.getElementById("usuarios");
  contenedor.innerHTML = "";

  usuarios.forEach(usuario => {
    const div = document.createElement("div");
    div.textContent = `${usuario.name} (${usuario.email})`;
    contenedor.appendChild(div);
  });
}
```

Comentemos qué hace la función:

- *const contenedor = document.getElementById("usuarios"): guardamos en la constante contenedor lo que tiene el elemento con id “usuarios”, que es el div.*
- *contenedor.innerHTML = "": creo un espacio dentro del div.*
- *usuarios.forEach(usuario => {… : recorro el array de usuarios*
- *const div = document.createElement("div"): crea una constante div que contiene un div.*
- *div.textContent = `${usuario.name} (${usuario.email})`: al div añadimos el contenido del array leído, recuperando lo atributos name y email.*

- *contenedor.appendChild(div): por último a la constante contenedor añadimos el nuevo elemento div creado.*

4. *Añadimos un escuchador de eventos al botón:*

```
document.getElementById("cargar").addEventListener("click", obtenerUsuarios);
```

5. *Finalmente abrimos el fichero index.html en un navegador, ¿se muestran los elemenos? Abre la consola del navegador para ver si hay mensajes de advertencias o errores.*

6.2. Actividades propuesto

En este tema vamos a tratar como JavaScript accede a datos exteriores usando promesas, then/catch, async/await y fetch().

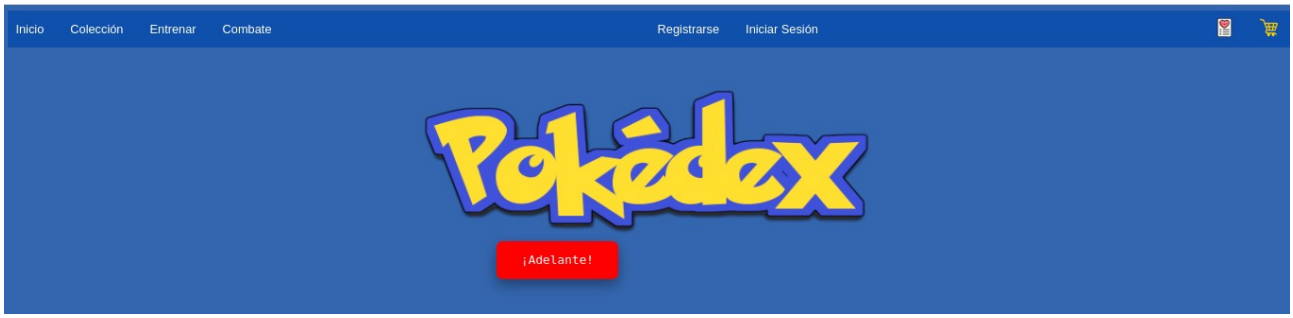
Antes de pasar a los ejercicios estudia la teoría en el ANEXO VII. Asincronía, Promesas, async/await, Ajax, fetch() y Axios.

Actividad 1
Solicita una lista de usuarios desde una API pública (JSONPlaceholder) y crea elementos HTML con document.createElement() para mostrar en el navegador su nombre y email.

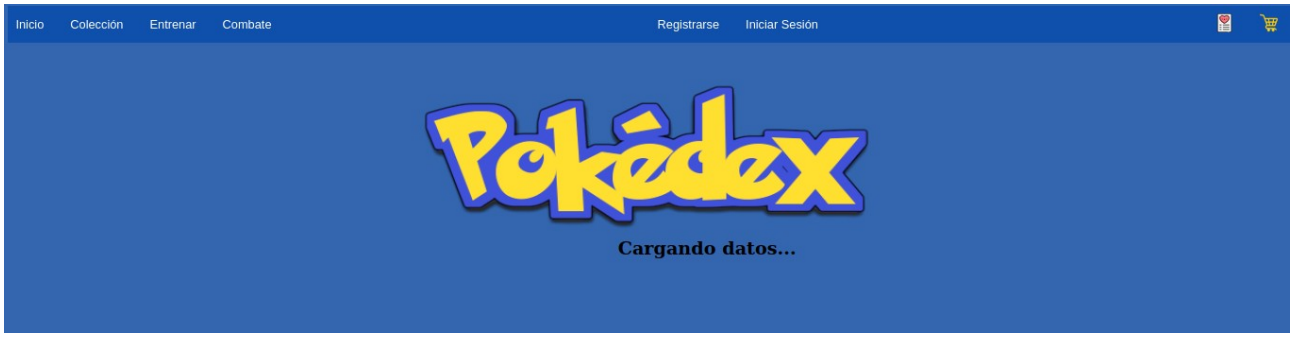
Actividad 2
Usando la actividad 1, añade un botón “Simular error” para volver a hacer la petición pero esta vez simulando un error en una petición (URL incorrecta o fallo intencional) y gestiona ese error mostrando un mensaje en el DOM con estilo visual (color, tamaño, etc.) de “Fallo en la petición”.
Para la petición normal: https://jsonplaceholder.typicode.com/users
Para simular error: https://jsonplaceholder.typicode.com/users-invalid-url

Actividad 3
Realiza una solicitud HTTP con fetch() a una API pública (por ejemplo, PokéAPI o CommunityDragon (de League of Legend)) y muestra por consola uno o varios datos concretos obtenidos de la respuesta.

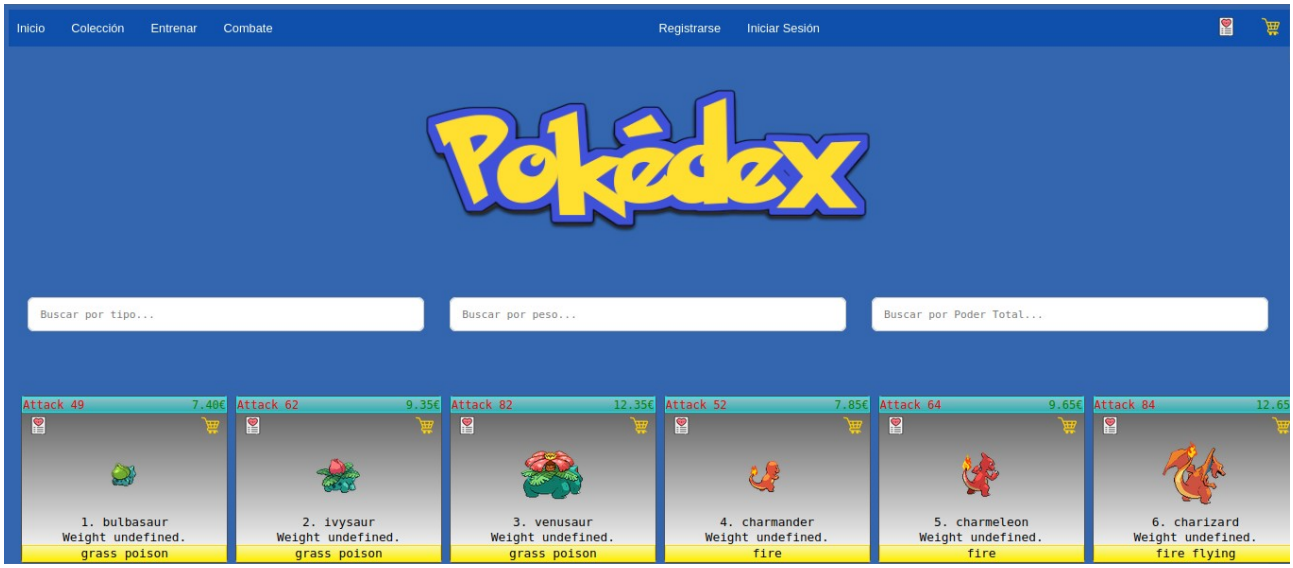
Véase un ejemplo de una Pokedex:



Tras pulsar:



Y cuando termina



ANEXO VII. Asincronía, Promesas, async/await, Ajax, fetch() y axios.

AVII.1. JavaScript Asíncrono

Hasta ahora, las cosas que hemos visto se desarrollan de forma secuencial. Pero podemos hacer, con las nuevas funcionalidades de javascript, acciones asíncronas, es decir, hacer unas cosas en background mientras se hacen otras funcionalidades en la aplicación.

Por ejemplo en el código:

```
const segundo = () => {
    setTimeout(() => {
        console.log('Llamada a segudo');
    }, 2000);
}

const primero = () => {
    console.log('Inicio primero');
    segundo();
    console.log('Fin primero');
}

primero();
```

Devuelve:

```
Inicio primero                                  app.js:8
Fin primero                                     app.js:10
Llamada a segudo                                app.js:3
```

¿Qué ha ocurrido? Pues en primero se ha seguido el orden secuencial, pero segundo() tiene un retardo de 2 segundos, tiempo que primero() aprovecha para continuar su secuencialidad, no se paraliza.

Un uso real de setTimeout() es cuando leemos dato de otra fuente externa que tardará en traerlos y no sabemos cuanto.

Vamos a simular un ejemplo:

```
const getUsuarios = () => {
    setTimeout(() => {
        const userIDs = [101,102,103,104];
        console.log(userIDs);
    }, 1500);
}
getUsuarios();
```

Esto simplemente se demora 1,5s y luego “trae” los userIDs.

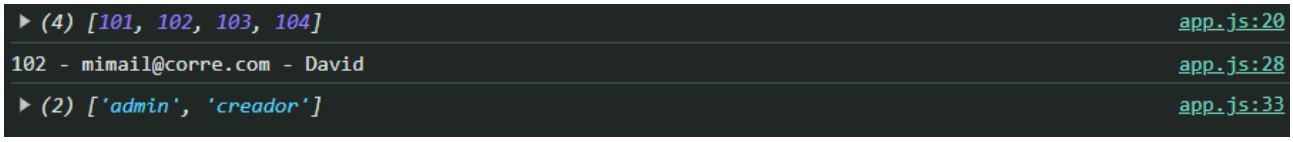
Ahora quiero mostrar datos de un usuario y “traer” sus permisos:

```
const getUsuarios = () => {
    setTimeout(() => {
        const userIDs = [101,102,103,104];
        console.log(userIDs);

        // Simulando que traigo los usuarios
        setTimeout(id => {
            const usuario = {
                email: 'mimail@corre.com',
                nombre: 'David'
            }
            console.log(`${id} - ${usuario.email} - ${usuario.nombre}`);

            // Simulando que traigo sus permisos
            setTimeout(idPermiso => {
                const permisos = ['admin', 'creador'];
                console.log(permisos);
            }, 1500, userIDs[2]);
        }, 1500, userIDs[1]);
    }, 1500);
}

getUsuarios();
```



Entiéndase de que cada Timeout es una simulación de acceso a datos.

Pero la anidación se puede complicar y el mantenimiento puede ser tedioso. Para solventarlo se hace unos de “promesas” (Promise). Véamosla en el siguiente punto.

AVII.2. Promesas

La promesa es un objeto, con la cual tendremos que instanciarla. Representa un valor que puede estar disponible ahora, en el futuro o nunca, en otras palabras, representa la terminación o el fracaso.

Su sintaxis es:

```
new Promise(function(resolver, rechazar){…..});
```

Veamos un ejemplo:

Primeramente yo declaro mi promesa:

```
let promesa = new Promise((resolve, reject) => {
    // El true o false del if es hardcodeado por ahora
    if (false) {
        resolve('La operación ha sido exitosa');
    }else{
        reject('Hubo un error');
    }
});
```

Si es verdadero se llama al método resolver() al que se le pasa el mensaje 'La operación ha sido exitosa'. En caso contrario se llama al método reject() al que se le pasa el mensaje 'Hubo un error’.

Ahora hagamos uso de la constante “promesa”:

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

Con el Then aplicamos que tiene que hacer:

Si response (todo ha ido bien) realiza el console.log. Nota que se pasa en la variable resolve el texto 'La operación ha sido exitosa'.

¿Y cuando hay error? Este se captura en el .catch, e igualmente, en la variable error va el mensaje 'Hubo un error.

Vamos a ver otro ejemplo donde podemos reescribir el código más conciso:

```
var miPromesa = Promise.resolve('Comida');
miPromesa.then(resp => console.log(resp));
```

En este caso declaro resolver directamente.

Veamos otro ejemplo:

```
let miPromesa3 = new Promise((resolve, reject) => {
    setTimeout(() => resolve(4), 2000);
});

miPromesa3.then(resp => {
    resp += 5;
    console.log(resp);
})
```

En este caso solo declaro el resolve dentro de un Timeout, que tras 2s envía el número 4 en el resolve. Y al hacer uso de miPromesa3 en el then recupero el valor 4 y le sumo 5.

Ahora vamos a resolver el problema de los callbacks anidados que teníamos:

```
const getUsuarios = () => {
    setTimeout(() => {
        const userIDs = [101,102,103,104];
        console.log(userIDs);

        // Simulando que traigo los usuarios
        setTimeout(id => {
            const usuario = {
                email: 'mimail@corre.com',
                nombre: 'David'
            }
            console.log(`${id} - ${usuario.email} - ${usuario.nombre}`);

            // Simulando que traigo sus permisos
            setTimeout(idPermiso => {
                const permisos = ['admin', 'creador'];
                console.log(permisos);
            }, 1500, userIDs[2]);
        }, 1500, userIDs[1]);
    }, 1500);
}

getUsuarios();
```

Resolvemos primero la parte de simular el que se trae los Ids:

```
const getUsuariosIDs = new Promise((response, reject) => {
    setTimeout(() => {
        response([101,102,103,104]); // Simulo que consigue los IDs
    }, 1500);
});

getUsuariosIDs.then(IDs => {
    console.log(IDs);
});
```

Continuamos con el siguiente timeout que simula que se trae al usuario con un ID:

Creo una nueva función que retornará una promesa. Dicha promesa será el Timeout para traer un usuario dado un id. Y en el resolve devuelve el string :${id} - ${usuario.email} - ${usuario.nombre}`.

```
    const getUsuario = userID => {
    return new Promise((resolve, reject) => {
        // Simulo que consigue los IDs
        setTimeout(id =>{
            const usuario = {
                email: 'mimail@correo.com',
                nombre: 'David'
            }
            resolve(`${id} - ${usuario.email} - ${usuario.nombre}`);
        }, 1500, userID);
    })
}
```

Toma nota que a la función setTimeout le pasamos el parámetro UserId que recuperamos con id. Seguidamente añado la nueva promesa al control de flujo:

```
getUsuariosIDs
.then(IDs => {
    console.log(IDs);
    return getUsuario(IDs[3])
})
.then(usuario => {
    console.log(usuario);
})
.catch(() => {
    console.log('Error');
});
```

Fíjese que en el primer Then, como ha ido todo bien se llama a la función promesa getUsuario(Id) cuya promesa se controla en el siguiente Then.

Finalmente vamos a traer los permisos, creamos la función promesa:

```
const getPermisos = id => {
    return new Promise((resolve, reject) => {
        // Simulo que consigue los permisos
        setTimeout(id=> {
            const permiso = ['admin', 'creador'];
            resolve(permiso);
        }, 1500, id);
    })
};
```

Y finalmente lo agregamos al control de flujo:

```
getUsuariosIDs
.then(IDs => {
    console.log(IDs);
    return getUsuario(IDs[3])
})
.then(usuario => {
    console.log(usuario);
    return getPermisos(usuario.id);
}).then(permisos => {
    console.log(permisos);
})
.catch(() => {
    console.log('Error');
});
```

La forma de tratar y declarar las promesas de esta manera puede ser un tanto engorrosa, por ello JavaScript aporta Async y Await para abordar el tema.

AVII.3. Asyn / Await

Async crea una función asíncron que se ejecuta en background.

El consumo de las promesas sería haciendo uso de async sería:

```
async function getUsiariosAW(){
    const IDs = await getUsuariosIDs();
    console.log(IDs);
    const usuario = await getUsuario(IDs(3));
    console.log(usuario);
    const permisos = await getPermisos(usuario.id);
    console.log(permisos);
}
```

Vemos que el código es más resumido y limpio. Además, con async indicamos que todo lo que opera dentro de la función es asíncrono y con await trata de recuperar los datos y no ejecuta lo siguiente hasta obtenerlo.

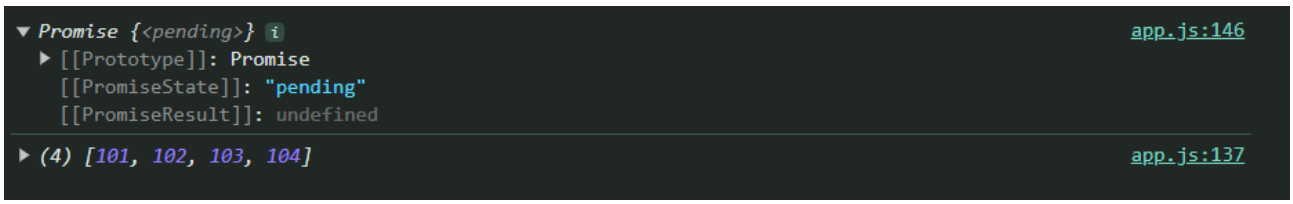
La función async devuelve una promesa que yo puedo manipular. Veamos como:

Vamos hacer que la función devuelva usuario:

```
async function getUsiariosAW(){
    const IDs = await getUsuariosIDs();
    console.log(IDs);
    const usuario = await getUsuario(IDs(3));
    console.log(usuario);
    const permisos = await getPermisos(usuario.id);
    console.log(permisos);
    return usuario;
}

const usuario = getUsuariosAW();
console.log(usuario);
```

Se puede observar que mientras se ejecutan las promesas usuario tan solo es una Promesa y está pendiente.



Cuando acaba el proceso se encuentra terminada:

```
▼ Promise {<pending>} i                                                    app.js:146
  ▼ [[Prototype]]: Promise
    ▶ catch: ƒ catch()
    ▶ constructor: ƒ Promise()
    ▶ finally: ƒ finally()
    ▶ then: ƒ then()
      Symbol(Symbol.toStringTag): "Promise"
    ▶ [[Prototype]]: Object
    [[PromiseState]]: "fulfilled"
    [[PromiseResult]]: "104 - mimail@correo.com - David"
▶ (4) [101, 102, 103, 104]                                                 app.js:137
104 - mimail@correo.com - David                                            app.js:139
▶ (2) ['admin', 'creador']                                                 app.js:141
```

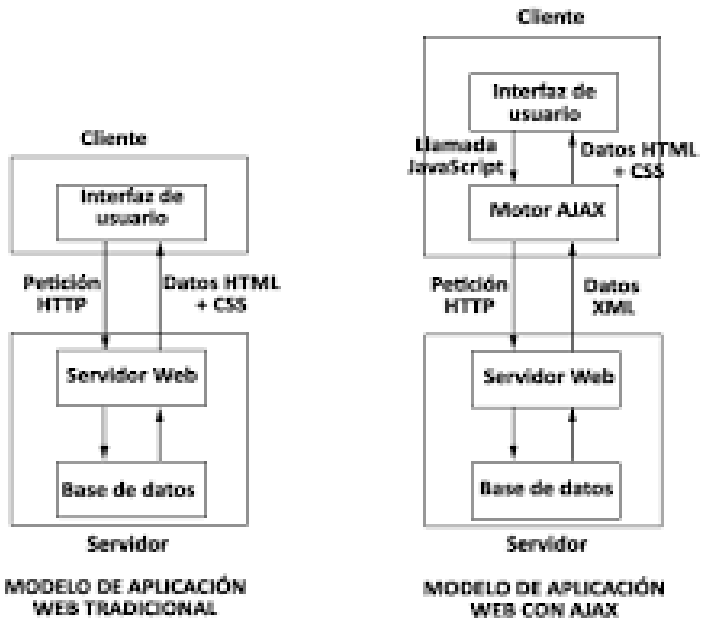
Podemos aprovechar que se comporta como una promesa para ver su flujo:

```
getUsuariosAW().then(resultado => console.log(`${resultado}, es un usuario`));
```

Mostramos el resultado para que se vea el uso de then.

```
▶ (4) [101, 102, 103, 104]                                                 app.js:137
104 - mimail@correo.com - David                                            app.js:139
▶ (2) ['admin', 'creador']                                                 app.js:141
104 - mimail@correo.com - David, es un usuario                             app.js:150
```

Cuando la comunicación asincrona se realiza contra un servidor. Se han implementado una serie de tecnologías para facilitar la comunicación y de aquí nace AJAX (Asynchronous Javascript and XML).



Donde el motor Ajax hace uso de XMLHttpRequest.

Bien, vamos a ver cómo quedaría el ejemplo haciendo uso de AJAX en el siguiente punto.

AVII.4. AJAX con HTTPRequest

Antes que nada accedamos a la web https://jsonplaceholder.typicode.com/. Es una página de prueba APIRest donde yo puedo hacer peticiones y probar mi aplicación.

Instancio la clase XMLHttpRequest para poder hacer las peticiones:

```
const request = new XMLHttpRequest();
```

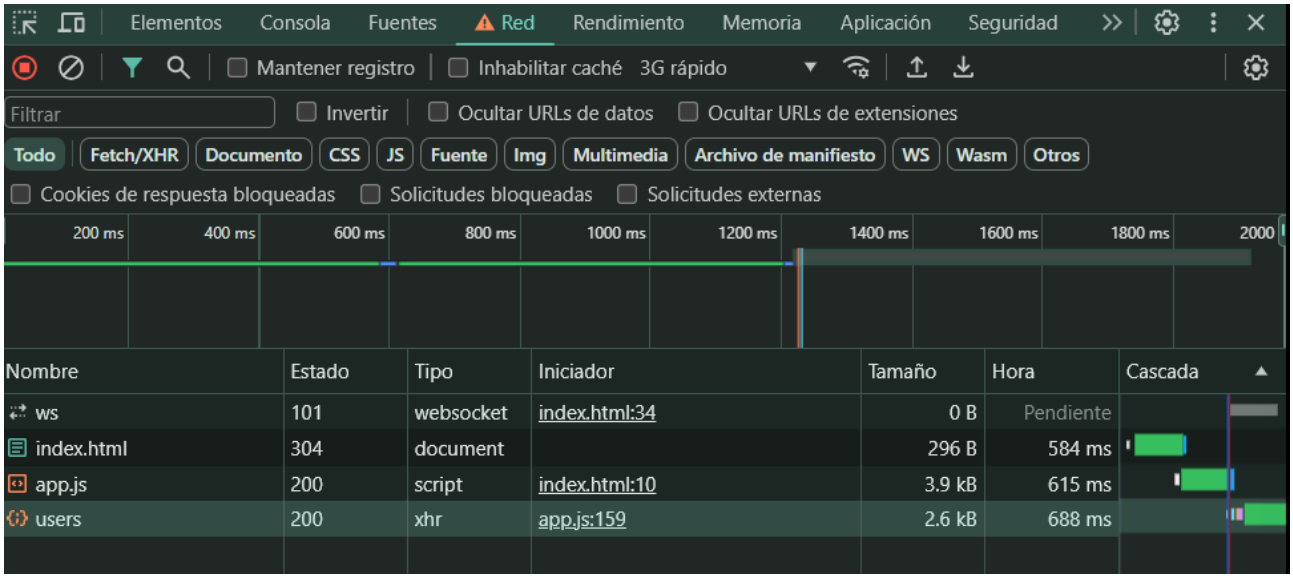
Abro el canal GET a la URL de la Web anterior para traerme unos usuarios de prueba.

```
const request = new XMLHttpRequest();
request.open('GET', 'https://jsonplaceholder.typicode.com/users');
```

Envío la petición:

```
const request = new XMLHttpRequest();
request.open('GET', 'https://jsonplaceholder.typicode.com/users');
request.send();
```

En el navegador podré ver en Network (Red) la comunicación con el servidor:



Nótese que se ha recibido un “users”. Si clicamos en el:

Vemos la información de la comunicación Http. Y en la pestaña Respuesta:

Están los datos que nos hemos traído.

Vamos a visualizarlo. Pero hemos de tener cuidado. Debemos asegurarnos de que la información ha terminado de descargarse. Lo realizamos con:

```
const request = new XMLHttpRequest();

request.addEventListener('readystatechange', (e) => {
    if(e.target.readyState === 4){
        const datos = JSON.parse(e.target.responseText);
        console.log(datos);
    }
})
request.open('GET', 'https://jsonplaceholder.typicode.com/users');
request.send();
```

Hemos creado un EventListener para escuchar el evento HTTPRequest readystatechange, para que, cuando tenga el valor 4 (devolución de “Done”, indicando que se ha realizado con éxito la comunicación y devolución de datos).

Obtenemos los datos:

Javascript tiene una librería con una función nativa pata hacer peticiones HTTP de forma más amigables, es Fetche(). Véase el siguiente Anexo VII.5.

AVII.5. Fetch()

Como decía anteriormente, fetch() es una función nativa que sustituye a XMLHttpRequest y se basa en promesas.

Su sintaxis es:

```
fetch(url, options)
  .then(response => {
    // manejar la respuesta
  })
  .catch(error => {
    // manejar errores de red
  });
```

fetch recibe una url a la que se hace una petición con unas opciones, opcionales, para configurar: el tipo de acceso (GET, POST, PUT, DELETE, PATCH), cabecera, cuerpo, etc.

Un ejemplo envío de datos podría ser:

```
const nuevoPost = {
  title: 'Nuevo título',
  body: 'Contenido del post',
  userId: 1
};

fetch('https://webejemplo.com/posts', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify(nuevoPost)
})
  .then(response => response.json())
  .then(data => {
    console.log('Respuesta del servidor:', data);
  })
  .catch(error => {
    console.error('Error al enviar:', error);
  });
```

Hay aún una manera más limpia y concisa de escribir el código haciendo uso de async/await. Véase el siguiente anexo AVII.6. Async/await.

AVII.6. Fetch con async/await

Veamos un ejemplo de uso:

```
async function cargarDatos() {
  try {
    const response = await fetch('https://jsonplaceholder.typicode.com/users');
    if (!response.ok) {
      throw new Error('Fallo en la carga');
    }
    const datos = await response.json();
    console.log(datos);
  } catch (error) {
    console.error('Error:', error);
  }
}

cargarDatos();
```

Se ha de tener ciertas consideraciones:

- Que el servidor no admita peticiones de otros dominios (CORS) lo que tendrás que modificar la configuración del servidor.
- Intenta siempre usar URLs seguras (https).
- Estar atento en enviar los datos correctos que se han de enviar en la cabecera.

AVII.7. Axios

Axios es una librería HTTP basada en Promesas para navegador y Node.js. En el lado del servidor usa el modulo nativo http de node.js, mientras que en el lado del cliente (navegador) usa XMLHttpRequest.

Su instalación en proyectos se realiza con: npm install axios

Siguiendo el ejemplo anterior de https://jsonplaceholder.typicode.com

Usando un entorno **ESM y Node ≥18:**

```
import axios from "axios";

const api = axios.create({ baseURL: "https://jsonplaceholder.typicode.com", timeout: 5000 });

async function main() {
  try {
    const { data: users } = await api.get("/users");
    console.log(users.map(u => `${u.id} - ${u.name}`));
  } catch (err) {
    if (axios.isAxiosError(err)) {
      console.error("HTTP error:", err.response?.status, err.message);
    } else {
      console.error("Unexpected:", err);
    }
  }
}
main();
```

Usando un entorno **CommonJS:**

```
const axios = require("axios");

(async () => {
  try {
    const { data } = await axios.get("https://jsonplaceholder.typicode.com/users");
    console.log(data);
  } catch (e) {
    console.error(e);
  }
})();
```

Cuándo elegir fetch o Axios

Fetch es usado apps modernas con necesidades simples HTTP y un control relativamente fino de cabecera para GET, POST, etc.

Axios e smás bien usado en proyectos con muchos endpoints, auth complejo, interceptores, métricas y manejo de errores uniforme.