Documentacion Reto

Gestion de Datos en el Cliente:

En el cliente se usa el localStorage y cookies para almacenar datos en local del usuario.

El localStorage se usa para almacenar la información de las empresas seleccionadas y saber en todo momento cual es la que desea operar; con la función siguiente se almacenan las empresas seleccionadas en el localStorage:

```
function extraerEmpresas() {

  const empresas = document.querySelectorAll('#dRecive .envia');

  if (empresas.length !== 0) {

    let array = [];

    for (let i = 0; i < empresas.length; i++) {

      array.push(empresas[i].id);

      }

    localStorage.setItem('empresas', array);

    router('operar')

    crearEmpresas();

  } else {

    alert('Debe seleccionar alguna empresa para poder operar')

  }}
```

Y en la siguiente funcion extraemos esas empresas y generamos el html y hacemos la llamada a la api para obtener los datos:

```
function crearEmpresas() {

  const empresas = localStorage.getItem('empresas').split(',');

  const contenedor = document.getElementById('listaEmpresas');



  const fecha = new Date();

  const fechaFormateada = fecha.toISOString().slice(0,
10).replace(/-/g, "-");



  empresas.map(function (empresa) {

    let div = document.createElement('div');

    card = `<div class="flex justify-center">

        <div class="rounded-lg shadow-lg bg-white max-w-sm">

          <a href="#!" data-mdb-ripple="true"
data-mdb-ripple-color="light">

            <img class="rounded-t-lg"
src="../../imagenes/${empresa}.png" alt="${empresa}"/>

          </a>

          <div class="p-6">

            <h5 class="text-sky-900 text-xl font-medium
mb-2">${empresa.toUpperCase()}</h5>

            <p class="text-gray-700 text-base mb-4">

              Precio: <span class="text-rose-800"
id="precio${empresa}">${datos}>0</span>

            </p>

            <a href="#!"
onclick="router('grafico','${empresa}');pararSetInterval()">

            <button id="${empresa}"  type="button" class=" inline-block
px-6 py-2.5 bg-sky-600 text-white font-medium text-xl leading-tight
uppercase rounded shadow-md hover:bg-sky-700 hover:shadow-lg
focus:bg-sky-700 focus:shadow-lg focus:outline-none focus:ring-0
```

```
active:bg-sky-800 active:shadow-lg transition duration-150
ease-in-out">Grafico</button>

            </a>

          </div>

        </div>

      </div>`

    div.innerHTML = card;

    contenedor.appendChild(div);

  })

  console.log(empresas, fechaFormateada);

  pedir(empresas, fechaFormateada, 'a')

}
```

De igual manera utilizamos las cookies para almacenar el token que permitirá hacer la petición de la API; donde tenemos un archivo llamado cookies.js que tiene todas las funciones que manejan esta cookie.

Gestion de Panel Drag & Drop:

La Gestión de Drag & Drop la realizo con JQuery de la siguiente manera:

```
$(function () {

        $(".envia").draggable({

            helper: "clone",

            appendTo: "body",

            start: function (event, ui) {

                $(ui.helper).addClass("dragging");

            },
```

```
            stop: function (event, ui) {
                $(ui.helper).removeClass("dragging");
            }
        });
        $("#dRecive,#dEnvia").droppable({
            accept: ".envia",
            drop: function (event, ui) {
                const dragged = $(ui.draggable);
                $(this).append(dragged);
            }
        });
    });
```

Gestión de Promesas:

Para las promesas he utilizado funciones asincronas para poder gestionar mejor el tiempo de respuesta de la API al realizar el fetch, ejemplo:

```
async function apiLogin() {
    try {
        var pass = document.querySelector('#pass').value;
        var email = document.querySelector('#email').value;
        var myHeaders = new Headers();
        myHeaders.append("Content-Type",
"application/x-www-form-urlencoded");
        var urlencoded = new URLSearchParams();
        urlencoded.append("email", email);
        urlencoded.append("password", pass);
        var requestOptions = {
```

```
            method: 'POST',
            headers: myHeaders,
            body: urlencoded,
            signal: AbortSignal.timeout(4000)
        };
        const response = await fetch("http://localhost/api/login",
requestOptions)
        if (response.status === 200) {
            const json = await response.json();
            setCookie('token', json.token, 20);
            location.reload()
        } else {
            throw new Error(`Error en la respuesta del servidor:
${response.status}`);
        }
    } catch (error) {
        console.error(error);
        alert('Error, verifica tus datos e intenta nuevamente');
    }
}
```

También he usado try catch para que la gestión de errores no me falle todo el programa.

Tengo un AbortController que en caso de que la petición tarde más de 4s se anula completamente.

En el intervalo siguiente gestiono la respuesta de la API y en caso de que suba o baje el precio cambio los colores; esta petición se realiza cada 10s:

```
interval = setInterval(async () => {
```

```
    datos = await apiPeticiones(empresas, fecha, hora);

    datos.Empresas.map((empresa) => {

      let preioViejo =
Number(document.getElementById(`precio${empresa.empresa}`).text);



      if (Number(empresa.cierre) >= preioViejo)

        document.getElementById(`precio${empresa.empresa}`).style.color
= 'green';

      else

        document.getElementById(`precio${empresa.empresa}`).style.color
= 'red';

      document.getElementById(`precio${empresa.empresa}`).innerHTML =
`${empresa.cierre}`;

    });

  }, 10000);
```

Archivo routes.js

Este archivo es el encargado de definir las rutas de la aplicación web. Esto se hace mediante la creación de objetos con los siguientes atributos:

- _path_: representa la ruta de la página.
- _template_: representa el HTML que se mostrará en la página correspondiente a la ruta.

Se crean 5 objetos: home, login, registro, operar y grafico. Cada objeto representa una página en la aplicación.

Además, se encuentra la función getHTML, la cual se encarga de obtener el HTML de cada página. La función recibe dos argumentos:

- _file_: representa la ruta del archivo HTML a obtener.

- _callback_: representa una función que se ejecutará una vez que se haya obtenido el HTML.

La función utiliza el objeto **XMLHttpRequest** para realizar una petición **GET** al archivo HTML correspondiente a cada objeto. Si la petición es exitosa, se asigna el HTML obtenido al atributo **template** del objeto correspondiente.

Archivo validaciones.js

Este archivo es el encargado de realizar las validaciones de los formularios de la aplicación web. Se encuentran dos funciones:

Función validacionLogin

- Esta función realiza la validación del formulario de inicio de sesión. Para esto, se utiliza el plugin jQuery Validate.
- El formulario se asocia a la función validate mediante su ID #loginForm. Luego, se definen las reglas y mensajes de error para los campos de email y password.
- Una vez realizadas las validaciones, se verifica si el formulario es válido y, en caso afirmativo, se ejecuta la función apiLogin.

Función validacionRegistro

- Esta función realiza la validación del formulario de registro de usuarios. Para esto, también se utiliza el plugin jQuery Validate.
- El formulario se asocia a la función validate mediante su ID #registrationForm. Luego, se definen las reglas y mensajes de error para los campos de nombre, email, password y password_confirmation.
- Una vez realizadas las validaciones, se verifica si el formulario es válido y, en caso afirmativo, se ejecuta la función apiRegistro.

Es importante destacar que antes de ejecutar la función validate, se detiene el comportamiento por defecto del formulario al hacer submit mediante la función preventDefault().

Archivo api.js

Este archivo api.js contiene cuatro funciones asíncronas que se conectan a un API. Las funciones son apiRegistro(), apiLogin(), apiLogout() y apiPeticiones().

función apiRegistro()

- Envía una solicitud POST al API para registrar un nuevo usuario. Recopila los datos de registro, incluyendo el nombre, correo electrónico, contraseña y confirmación de la contraseña, de los elementos del formulario con querySelector(). Luego, agrega estos datos a un objeto URLSearchParams y establece la cabecera Content-Type para "application/x-www-form-urlencoded". La solicitud POST se envía a la dirección http://localhost/api/register con una señal de aborto que tiene un tiempo de espera de 4000 milisegundos. Si la respuesta del servidor es un código de estado 200, muestra una alerta de que el usuario se ha creado correctamente y redirige a la página de inicio de sesión. De lo contrario, muestra un error en la consola y en una alerta.

función apiLogin()

- Envía una solicitud POST al API para iniciar sesión con un usuario existente. Recopila los datos de inicio de sesión, incluyendo el correo electrónico y la contraseña, de los elementos del formulario con querySelector(). Luego, agrega estos datos a un objeto URLSearchParams y establece la cabecera Content-Type para "application/x-www-form-urlencoded". La solicitud POST se envía a la

dirección http://localhost/api/login con una señal de aborto que tiene un tiempo de espera de 4000 milisegundos. Si la respuesta del servidor es un código de estado 200, se guarda el token devuelto en una cookie y se actualiza la página. De lo contrario, muestra un error en la consola y en una alerta.

función apiLogout()

- Envía una solicitud POST al API para cerrar la sesión de un usuario. Establece una cabecera Authorization que contiene el token guardado en una cookie con getCookie('token'). La solicitud POST se envía a la dirección http://localhost/api/logout con una señal de aborto que tiene un tiempo de espera de 4000 milisegundos. Si la respuesta del servidor es un código de estado 200, se borra el token de la cookie y se muestra un mensaje de éxito en la consola y se actualiza la página. De lo contrario, muestra un error en la consola.

función apiPeticiones()

- Realiza peticiones GET y POST al API y se puede utilizar para cualquier otro fin que requiera acceder a los datos almacenados en el servidor. Esta función acepta una URL, un método y un objeto con datos opcional como parámetros. Si se proporciona un objeto con datos, se agrega a un objeto URLSearchParams y se establece la cabecera Content-Type para "application/x-www-form-urlencoded". La solicitud se envía a la dirección proporcionada con el método especificado y una señal de aborto con un tiempo de espera de 4000 milisegundos. Si la respuesta del servidor es un código de estado 200, se devuelve la respuesta como un objeto JSON. De lo contrario, muestra un error en la consola y en una alerta.

En resumen, este archivo api.js ofrece una manera sencilla de realizar solicitudes a un API con una estructura común y un manejo de errores claro. Con estas funciones, es posible realizar tareas como registrar y autenticar usuarios, realizar peticiones al servidor y cerrar sesión.

Archivo graficos.js

Este archivo se llama grafico.js y es un script que se utiliza para crear un gráfico de precios de una empresa en función del tiempo.

El archivo define varias variables, como precios, datosGrafico, empresas y labels, que se utilizan en diferentes partes del código. Además, define una variable chartLine, que es un objeto de Chart.js y se utiliza para representar el gráfico.

función obtenerFecha()

- Se utiliza para determinar una fecha en función de un intervalo de tiempo específico (1 día, 1 semana, 1 mes o 1 año).

función crearGrafico()

Es la función principal y se encarga de realizar la petición API para obtener los datos de precios de la empresa y crear el gráfico.

función escribirGrafico()

Se encarga de crear el gráfico de línea utilizando Chart.js y se utiliza la variable configLineChart para definir las opciones y configuraciones del gráfico.

En resumen, este archivo se utiliza para obtener los datos de precios de una empresa y representarlos en un gráfico de línea interactivo utilizando Chart.js.

Archivo operar.js

Este archivo JavaScript, llamado operar.js, es un script que muestra una lista de empresas seleccionadas por el usuario y actualiza los precios de las acciones de estas empresas en tiempo real.

función extraerEmpresas()

Se ejecuta al presionar un botón y recoge todas las empresas seleccionadas por el usuario. Luego, estas empresas se guardan en el almacenamiento local del navegador y se redirige a la página 'operar'.

función crearEmpresas()

Crea una lista de las empresas seleccionadas y muestra sus logos, nombres, precios actuales y un botón para visualizar un gráfico.

función pedir()

Hace una petición a una API para obtener los precios de cierre de las acciones de cada empresa en una fecha y hora determinadas. Estos precios se actualizan cada 8 segundos gracias a la función setInterval().

función pararSetInterval()

Detiene la actualización continua de los precios.