Tecnologías Informáticas B

Tutoría Frontend Refactoring 01

El frontend de este ejemplo está conformado por los archivos de tipo html, css y js. Que coinciden respectivamente con los lenguajes de HTML, CSS y Javascript. En Tecnologías B trabajaron con los lenguajes HTML y CSS, es por eso que en esta tutoría nos limitaremos a explicar el archivo que participa como una especie de controlador del Front End y su interacción con el archivo html.

Explicación de frontDispatcher.js

1. Antes que nada, qué es Javascript:
    a. JavaScript es un lenguaje de programación o de secuencias de comandos que te permite implementar funciones complejas en páginas web, cada vez que una página web hace algo más que sentarse allí y mostrar información estática para que la veas, muestra oportunas actualizaciones de contenido, mapas interactivos, animación de Gráficos 2D/3D, desplazamiento de máquinas reproductoras de vídeo, etc., puedes apostar que probablemente JavaScript está involucrado. Es la tercera capa del pastel de las tecnologías web estándar, dos de las cuales (HTML y CSS) hemos cubierto con mucho más detalle en otras partes del Área de aprendizaje. (MDN Web Docs https://developer.mozilla.org/es/docs/Learn_web_development/Core/Scripting/What_is_JavaScript)

2. Se intentará explicar a continuación características modernas que se utilizan en la capa que permite separar el maquetado HTML de la lógica de Backend, osea el frontDispatcher.js, este archivo está incluído al final del index.html cómo pueden verificar y puede acceder a modificar su comportamiento.

Explicación de conceptos clave de JavaScript:

async y await:

Estas dos palabras clave permiten trabajar con **promesas** (acciones que toman tiempo, como consultas a servidores) de forma más sencilla.

```
async function fetchData() {
    const res = await fetch('url');
    const data = await res.json();
}
```

- **async** marca una función como asincrónica. Esto significa que puede usar **await** adentro.

- **await** pausa la ejecución hasta que la promesa se resuelve. Hace que el código asincrónico se lea de forma *sincrónica* (ordenada).

✔️ **Ventaja**: más legible que usar **.then().catch()**

Funciones flecha (=>)

Son una forma más moderna y corta de escribir funciones.

```
async function fetchData() {
    const res = await fetch('url');
    const data = await res.json();
}
```

En **frontDispatcher.js** usamos esto por ejemplo en:

```
document.addEventListener('DOMContentLoaded', () => { ... });
```

Acá la función flecha es **anónima** (no tiene nombre).

Se ejecuta automáticamente cuando la página terminó de cargarse.

DOMContentLoaded

Es un evento que indica que todo el HTML fue cargado y procesado por el navegador.

```
document.addEventListener('DOMContentLoaded', () => {
   // Aquí va el código que usa elementos del DOM
});
```

Esto evita errores como `document.getElementById(...)` is `null` porque garantiza que los elementos del HTML existen antes de usarlos.

JSON.stringify() y JSON.parse()

Cuando enviamos o recibimos datos por la red, usamos el **formato JSON** (texto con estructura de objetos).

- **JSON.stringify(objeto)** convierte un objeto JS en texto JSON
- **JSON.parse(textoJSON)** convierte un texto JSON en un objeto JS

Ejemplo:

```
const alumno = { nombre: "Juan", edad: 22 };
const json = JSON.stringify(alumno); //
'{"nombre":"Juan","edad":22}'
const nuevoAlumno = JSON.parse(json); // objeto JS otra vez
```

En el código:

```
body: JSON.stringify(formData)
```

Estamos enviando los datos del formulario al backend como texto JSON.

Funciones anónimas

Son funciones sin nombre, y se usan mucho cuando solo necesitás ejecutar algo **una sola vez** o como **callback**.

Ejemplo:

```
button.addEventListener('click', function() {
  alert("Hiciste clic");
});
```

O con flecha:

```
button.addEventListener('click', () => {
  alert("Hiciste clic");
});
```

Manipulación del DOM sin innerHTML

Es preferible **crear elementos con createElement y appendChild** en lugar de insertar HTML como texto, ya que:

- Es **más seguro** (evita ataques XSS).
- Es **más controlado** y menos propenso a errores.
- No borra y recrea todo el contenido, lo cual ayuda al rendimiento.

Ejemplo en el código:

```
const tdName = document.createElement('td');
tdName.textContent = student.fullname;
```

Confirmación con confirm()

```
if (!confirm("¿Seguro que querés borrar este estudiante?"))
return;
```

- Muestra un cuadro de diálogo con **Aceptar** y **Cancelar**.
- Si el usuario cancela, la función termina (**return**).

Explicación de funcionalidad del código:

frontDispatcher.js

Este script maneja toda la lógica de interacción entre el frontend y el backend utilizando JavaScript puro (vanilla JS), fetch, async/await y evitando el uso de innerHTML por seguridad (XSS).

```
const API_URL = '../backend/server.php';
```

- Define la URL del backend que se utilizará en todas las llamadas `fetch`.
- Usar una constante al principio del archivo permite modificar la ruta fácilmente si cambia la estructura del proyecto.

```
document.addEventListener('DOMContentLoaded', () => {
```

Espera a que el contenido HTML esté completamente cargado antes de ejecutar el resto del código JavaScript.

Referencias a los elementos del DOM:

```
const studentForm = document.getElementById('studentForm');
conststudentTableBody=document.getElementById('studentTableBody');
const fullnameInput = document.getElementById('fullname');
const emailInput = document.getElementById('email');
const ageInput = document.getElementById('age');
const studentIdInput = document.getElementById('studentId');
```

Se almacenan referencias a los elementos del formulario y la tabla para poder leer y actualizar sus valores.

```
fetchStudents();
```

Llama a la función que carga todos los estudiantes desde el backend y los muestra en la tabla al iniciar la página.

```
studentForm.addEventListener('submit', async (e) => {
        e.preventDefault();
```

Cancela el envío tradicional del formulario (que recargaría la página).

```
  const formData = {
      fullname: fullnameInput.value,
      email: emailInput.value,
      age: ageInput.value,
  };
```

Crea un objeto con los valores del formulario.

```
const id = studentIdInput.value;
const method = id ? 'PUT' : 'POST';
if (id) formData.id = id;
```

Si hay un **id**, significa que se está editando un estudiante (método PUT). Si no hay, se está creando uno nuevo (método POST).

```
try {
    const response = await fetch(API_URL, {
        method,
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify(formData),
    });
```

Envía los datos al backend usando el método adecuado y con el cuerpo en formato JSON.

```
if (response.ok) {
    studentForm.reset();
    studentIdInput.value = '';
    await fetchStudents();
```

Si la operación fue exitosa:

- Se resetea el formulario.
- Se borra el **id** oculto.

  Se recarga la lista de estudiantes.

```
        } else {
            alert("Error al guardar");
        }
    } catch (err) {
        console.error(err);
    }
```

Muestra errores si algo sale mal con la conexión. Utilizando manejo de excepciones (try y catch), característica que permite atrapar un comportamiento “excepcional” para lo cual no fue pensado, ni preparado el código.

Función fetchStudents()

```
async function fetchStudents() {
      try {
          const res = await fetch(API_URL);
          const students = await res.json();
```

Hace una solicitud GET al backend y convierte la respuesta JSON en un array de estudiantes.

```
studentTableBody.replaceChildren();
```

Borra todas las filas de la tabla (de forma segura, sin usar innerHTML).

```
students.forEach(student => {
    const tr = document.createElement('tr');
```

Itera sobre cada estudiante y crea una nueva fila de tabla por cada uno.

```
const tdName = document.createElement('td');
    tdName.textContent = student.fullname;
```

Crea una celda y le asigna el nombre del estudiante. Lo mismo se hace para email y age.

```
const tdActions = document.createElement('td');
```

Crea la celda para botones "Editar" y "Borrar".

```
const editBtn = document.createElement('button');
editBtn.textContent = 'Editar';
editBtn.classList.add('w3-button', 'w3-blue', 'w3-small',
'w3-margin-right');
editBtn.onclick = () => {
                    fullnameInput.value = student.fullname;
                    emailInput.value = student.email;
                    ageInput.value = student.age;
                    studentIdInput.value = student.id;
                };
```

Botón para editar con estilos de la w3schools: carga los datos del estudiante al formulario y guarda su **id**.

```
const deleteBtn = document.createElement('button');
deleteBtn.textContent = 'Borrar';
deleteBtn.classList.add('w3-button', 'w3-red', 'w3-small');
deleteBtn.onclick = () => deleteStudent(student.id);
```

Botón para borrar: llama a deleteStudent() con el id.

```
tdActions.appendChild(editBtn);
tdActions.appendChild(deleteBtn);
```

Agrega ambos botones a la celda de acciones.

```
tr.appendChild(tdName);
tr.appendChild(tdEmail);
tr.appendChild(tdAge);
tr.appendChild(tdActions);
studentTableBody.appendChild(tr);
```

Ensambla toda la fila y la agrega al cuerpo de la tabla.

Función deleteStudent(id)

```
async function deleteStudent(id) {
 if (!confirm("¿Seguro que querés borrar este estudiante?"))
return;
```

Pide confirmación antes de eliminar.

```
 try {
       const response = await fetch(API_URL, {
         method: 'DELETE',
         headers: { 'Content-Type': 'application/json' },
         body: JSON.stringify({ id }),
     });
```

Envia una solicitud DELETE con el `id` en el cuerpo JSON.

```
    if (response.ok) {
       await fetchStudents();
    } else {
         alert("Error al borrar");
    }
    } catch (err) {
           console.error(err);
    }
  }
});
```

Si la eliminación fue exitosa, recarga la tabla.

FIN DE LA EXPLICACIÓN DEL SCRIPT