«Talento Tech»

Front-End JS

Clase 14

Buenos Aires Ciudad

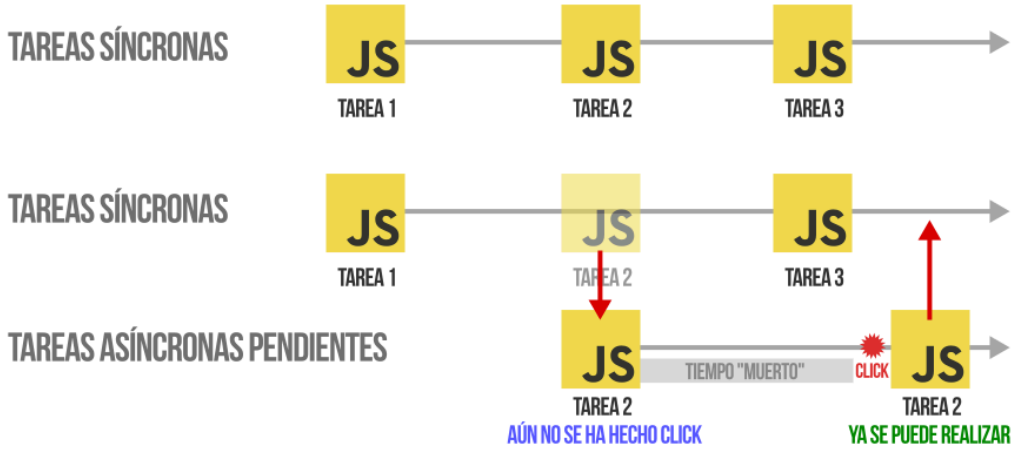
Clase 14: Asincronía

1. Introducción a la Asincronía
2. Fetch: Consumir una API REST
3. Promesas: ¿Qué es una promesa en JavaScript?
4. Manejo de Datos con Fetch y Promesas
5. Manejo de Errores con Fetch
6. Async/Await: Otra Forma de Esperar
7. Ejemplo Práctico: Consumir API Externa y Mostrar Datos en HTML
8. Ejemplo Práctico: Crear un Carrito de Compras con Productos de una API

1. Introducción a la Asincronía

Imaginemos que estás cocinando una pizza. Ponés la masa en el horno, pero no te quedás mirándola todo el tiempo, ¿no? Mientras se cocina podés hacer otras cosas: cortar algunos ingredientes, preparar la salsa, etc. Esto es lo que denominamos, en programación, asincronía. En JavaScript, cuando realizás una tarea que va a demorar cierto tiempo (como pedir datos a una API) podés hacer otras cosas mientras esperás la respuesta.

Asincronía en JavaScript: Básicamente, el código sigue ejecutándose sin esperar a que la tarea asíncrona (como una petición a una API) se complete. Esto es muy útil en aplicaciones web porque permite que la página siga funcionando mientras se procesan otras tareas, como cargar productos desde una base de datos externa.



2. Fetch: Consumir una API REST

Pero, ¿cómo pedimos esos datos que no están en nuestra página? Acá es donde entra **fetch()**. `fetch()` nos permite hacer peticiones a servidores externos para obtener datos.

¿Qué es `fetch`?
Es una función que nos permite hacer una solicitud HTTP para obtener datos. Lo genial es que lo hace de manera asíncrona, es decir, no interrumpe el funcionamiento de la página.

Código básico con `fetch()`:

```
fetch('https://fakestoreapi.com/products')
    .then(response => response.json())  // Convertimos la
respuesta a JSON
    .then(data => {
        console.log(data);  // Mostramos los datos obtenidos
    })
    .catch(error => {
        console.error('Hubo un error al obtener los
productos:', error);
    });
```

¿Qué pasa en este código?

1. **fetch()**: Hace la petición a la API.
2. **.then(response => response.json())**: Cuando la API responde, convertimos los datos a JSON.
3. **.then(data => ...)**: Utilizamos esos datos (en este caso, los mostramos en la consola).
4. **.catch(error => ...)**: Si hay un error (por ejemplo, si la API no responde), lo manejamos y mostramos un mensaje de error.

3. Promesas: ¿Qué es una promesa en JavaScript?

Cuando usás `fetch`, te devuelve una promesa: funciona como lo indica la palabra, es un compromiso del programa que tiene tres estados factibles. Pensalo como cuando quedás con tu amiga en cenar el sábado próximo, pero tenés que confirmar si podés más cerca de la fecha. En nuestra programación, esta promesa, como decíamos, puede tener tres estados: **pendiente, resuelta o rechazada.**

Estados de una promesa:

- **Pendiente (pending)**: La promesa todavía está esperando una respuesta (no sabés si vas a ir a cenar o no).
- **Resuelta (resolved)**: Todo salió bien (confirmaste que vas a cenar).
- **Rechazada (rejected)**: Algo salió mal (tu amiga canceló la cena).

4. Manejo de datos con Fetch y promesas

Ahora que sabemos pedir datos, ¿qué hacemos con ellos? Supongamos que estamos creando una tienda online y queremos mostrar productos de una API externa.

Mostrar los productos en nuestra página:

```
fetch("https://fakestoreapi.com/products")
 .then((response) => response.json())
 .then((data) => {
   const contenedor =
document.getElementById("productos-container");
   data.forEach((producto) => {
     contenedor.innerHTML += `
               <div class="card">
                   <img src="${producto.image}"
alt="${producto.title}">
                   <h3>${producto.title}</h3>
                   <p>Precio: $${producto.price}</p>
                   <button
onclick="agregarAlCarrito(${producto.id})">Añadir al
carrito</button>
               </div>
           `;
```

```
    });
  })
  .catch((error) => console.error("Error al obtener
productos:", error));
```

Explicación:

1. **Recorremos** los datos de los productos que nos devolvió la API.
2. Por cada producto, creamos una tarjeta (`card`) con la imagen, el nombre, el precio y un botón para agregar al carrito.
3. El botón llama a la función `agregarAlCarrito`, que es la que nos va a permitir agregarlo al carrito.

5. Manejo de errores con Fetch.

Como en la vida misma, en ocasiones las cosas no salen como esperamos. Las promesas pueden fallar y es importante dar gestión a dichos errores.

Manejo de errores:

```
fetch("https://fakestoreapi.com/products")
  .then((response) => response.json())
  .then((data) => {
    // Mostrar productos...
  })
  .catch((error) => {
    console.error("Error al obtener productos:", error);
    alert("Hubo un problema al cargar los productos.");
  });
```

6. Async/Await: Otra forma de esperar

Hay otra forma de manejar la asincronía que es más sencilla y clara: **async/await**. Con `async` le decimos a la función que va a manejar tareas asíncronas y con `await` le decimos que espere a que termine antes de seguir.

Código usando async/await:

```
async function obtenerProductos() {
 try {
   const response = await
fetch("https://fakestoreapi.com/products");
   const data = await response.json();
   console.log(data);
 } catch (error) {
   console.error("Error al obtener productos:", error);
 }
}
```

¿Por qué usar async/await?

Es más fácil de leer que las promesas anidadas con `.then()` y hace que el código sea más claro.

7. Ejemplo práctico: Consumir API externa y mostrar datos en HTML

Vamos a implementar un ejemplo completo que consuma una API y muestre productos en la pantalla de un e-commerce.

HTML:

```
<!DOCTYPE html>
<html lang="es">
 <head>
   <meta charset="UTF-8" />
   <meta name="viewport" content="width=device-width,
initial-scale=1.0" />
   <title>Tienda Online</title>
   <link rel="stylesheet" href="styles.css" />
 </head>
 <body>
   <h1>Productos Disponibles</h1>
   <div id="productos-container"></div>

   <script src="script.js"></script>
 </body>
</html>
```

JavaScript (script.js):

```
document.addEventListener("DOMContentLoaded", () => {
 fetch("https://fakestoreapi.com/products")
   .then((response) => response.json())
   .then((data) => {
     const contenedor =
document.getElementById("productos-container");
     data.forEach((producto) => {
       contenedor.innerHTML += `
                 <div class="card">
                     <img src="${producto.image}"
alt="${producto.title}">
                     <h3>${producto.title}</h3>
                     <p>Precio: $${producto.price}</p>
                     <button
onclick="agregarAlCarrito(${producto.id})">Añadir al
carrito</button>
                 </div>
             `;
     });
   })
   .catch((error) => {
     console.error("Error al obtener productos:", error);
     contenedor.innerHTML = "<p>Hubo un problema al cargar los
productos.</p>";
   });
});

function agregarAlCarrito(idProducto) {
 let carrito = JSON.parse(localStorage.getItem("carrito")) ||
[];
 carrito.push(idProducto);
 localStorage.setItem("carrito", JSON.stringify(carrito));
 alert("Producto añadido al carrito");
}
```

8. Ejemplo práctico: Crear un carrito de compras con productos de una API

Ahora vamos a integrar todo esto en un carrito de compras básico. Vamos a guardar los productos en el carrito utilizando **LocalStorage** para que, aunque se cierre la página, el carrito siga estando ahí.

Pasos:

1. Mostrar los productos obtenidos desde la API.
2. Cuando se haga clic en "Añadir al carrito", se debe guardar ese producto en LocalStorage.
3. Al abrir la página, mostrar cuántos productos tiene el carrito.

Código extendido del carrito:

JavaScript (agregando la función para mostrar el carrito):

```
function actualizarCarrito() {
 const carrito = JSON.parse(localStorage.getItem("carrito"))
|| [];
 const carritoCounter =
document.getElementById("cart-counter");
 carritoCounter.textContent = carrito.length;
}

document.addEventListener("DOMContentLoaded", () => {
 fetch("https://fakestoreapi.com/products")
   .then((response) => response.json())
   .then((data) => {
     const contenedor =
document.getElementById("productos-container");
     data.forEach((producto) => {
       contenedor.innerHTML += `
                 <div class="card">
                     <img src="${producto.image}"
alt="${producto.title}">
                     <h3>${producto.title}</h3>
```

```
                    <p>Precio: $${producto.price}</p>
                    <button
onclick="agregarAlCarrito(${producto.id})">Añadir al
carrito</button>
                </div>
            `;
    });
    actualizarCarrito();
  })
  .catch((error) => {
    console.error("Error al obtener productos:", error);
    contenedor.innerHTML = "<p>Hubo un problema al cargar los
productos.</p>";
  });
});

function agregarAlCarrito(idProducto) {
 let carrito = JSON.parse(localStorage.getItem("carrito")) ||
[];
 carrito.push(idProducto);
 localStorage.setItem("carrito", JSON.stringify(carrito));
 alert("Producto añadido al carrito");
 actualizarCarrito();
}
```

Ejercicio práctico #1:

Aplicar el consumo de API Fetch en tu proyecto personal

Enunciado:
En este ejercicio, vas a integrar el consumo de una API REST utilizando `fetch()` en tu proyecto personal de e-commerce o cualquier otro proyecto que estés desarrollando. Los pasos a seguir son:

1. **Elige una API pública** (como Fake Store API) que te proporcione datos de productos, usuarios y usuarias o cualquier otro recurso que quieras mostrar en tu proyecto.
2. Usa **fetch()** para hacer una solicitud a la API y obtener los datos.
3. Muestra los datos obtenidos en tu proyecto, ya sea en forma de lista de productos, usuarias o usuarios o lo que elijas.
4. Asegúrate de manejar los posibles errores utilizando `.catch()` y mostrá un mensaje si algo falla.
5. Opcional: Integra los datos obtenidos con alguna funcionalidad de tu proyecto, como un carrito de compras o una lista de productos favoritos.

Tips:

- **Selección de API**: Busca una API pública que tenga los datos que querés incluir en tu proyecto. Algunas opciones son APIs de productos, películas, personas, etc.
- **Manejo de errores**: Usa `.catch()` para capturar cualquier error en la solicitud y mostrar un mensaje de error en la página.
- **Integración con el proyecto**: Pensá cómo podés integrar los datos obtenidos con otras funcionalidades de tu proyecto.

Ejercicio práctico #2:

Crear un carrito de compras dinámico con productos de una API

Enunciado:
Vas a crear un carrito de compras dinámico que permite agregar productos al carrito utilizando datos obtenidos de una API externa. Los pasos específicos son:

1. Utilizá **fetch()** para obtener una lista de productos desde una API (puede ser la misma API de productos del Ejercicio 1).
2. Mostrá los productos en la página en forma de tarjetas o lista.
3. Agregá un botón "Añadir al carrito" para cada producto. Al hacer clic en el botón, el producto debe añadirse al carrito.
4. Usá **LocalStorage** para almacenar los productos que se agreguen al carrito, de manera que si recarga la página, los productos sigan allí.
5. Mostrá la cantidad de productos que hay en el carrito en todo momento, actualizándose cada vez que se añada un nuevo producto.

Tips:

- **Manipulación del DOM**: Usá métodos como `createElement()` y `appendChild()` para crear dinámicamente las tarjetas de productos y los botones.
- **LocalStorage**: Almacená los productos agregados en LocalStorage utilizando `JSON.stringify()` y `JSON.parse()` para convertir los objetos a formato JSON y viceversa.
- **Actualización del carrito**: Cada vez que un producto sea añadido al carrito, asegurate de actualizar el contador del carrito para poder ver cuántos productos tiene.


Buenos Aires
aprende
Buenos
Aires
Ciudad