«Talento Tech»

Front-End JS

Clase 14

Buenos Aires Ciudad

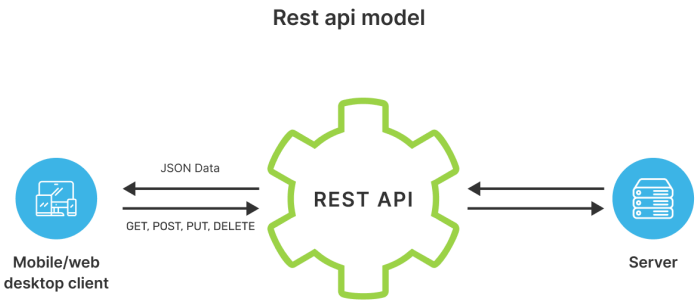
Clase 14: Asincronía e Introducción a API

Índice

Objetivo de la clase:

En esta clase vas a comprender qué es una API REST y por qué se utiliza en el desarrollo web moderno, explorando ejemplos reales de su aplicación. Aprenderás el concepto de asincronía y cómo JavaScript maneja tareas asincrónicas para mejorar la experiencia de usuario. También conocerás el uso de `fetch()` para consumir datos de una API, interpretar la respuesta en formato JSON y gestionar posibles errores. Finalmente, aplicarás estos conocimientos para renderizar dinámicamente productos en el HTML, conectando los datos obtenidos con la estructura visual de tu sitio web.

1. Introducción al Consumo de API REST



1.1. **¿Qué es una API REST?**

Podés pensar en una API REST como un mozo que lleva tu pedido a la cocina de un restaurante y después te trae la comida. Vos no necesitás saber cómo cocinan, solo que tu pedido llegue correctamente. La API hace de intermediario entre tu aplicación y un servidor.

1.2. **Ejemplos de Uso en Empresas de Tecnología**

- **Redes Sociales**: Facebook, Twitter e Instagram ofrecen APIs que permiten acceder a perfiles, publicaciones y seguidores.
- **E-commerce**: Plataformas como Amazon y eBay usan APIs para facilitar la integración de productos, precios y disponibilidad en tiendas online externas.
- **Mapas y Geolocalización**: Google Maps y OpenStreetMap tienen APIs que permiten a las aplicaciones obtener información geográfica, rutas y mapas en tiempo real.
- **Pago y Autenticación**: PayPal, Stripe, y MercadoPago ofrecen APIs para gestionar pagos seguros y verificar identidades.
- **Noticias y Entretenimiento**: APIs como las de Spotify y YouTube permiten que las aplicaciones integren contenido musical o audiovisual.

🛑Las APIs REST son fundamentales para la conectividad y el intercambio de datos entre sistemas de diversas industrias, ofreciendo flexibilidad y potencia para crear experiencias digitales modernas.

1.3. **¿Por qué se usa JSON en lugar de XML?**

- **Ligereza**: JSON (JavaScript Object Notation) es más ligero y fácil de leer que XML. Esto permite transferir datos de manera más rápida, reduciendo el tiempo de carga en la aplicación.
- **Compatibilidad**: JSON es más compatible con JavaScript, ya que se puede convertir directamente en objetos JavaScript. Con XML, era necesario un procesamiento adicional para extraer los datos.
- **Legibilidad y Sencillez**: JSON es menos verboso, haciendo que el código sea más claro y fácil de interpretar.

Ejemplo de JSON vs. XML:

XML

```
<empinfo>
   <employees>
      <employee>
         <name>James Kirk</name>
         <age>40></age>
      </employee>
      <employee>
         <name>Jean-Luc Picard</name>
         <age>45</age>
      </employee>
      <employee>
         <name>Wesley Crusher</name>
         <age>27</age>
      </employee>
   </employees>
</empinfo>
```

JSON

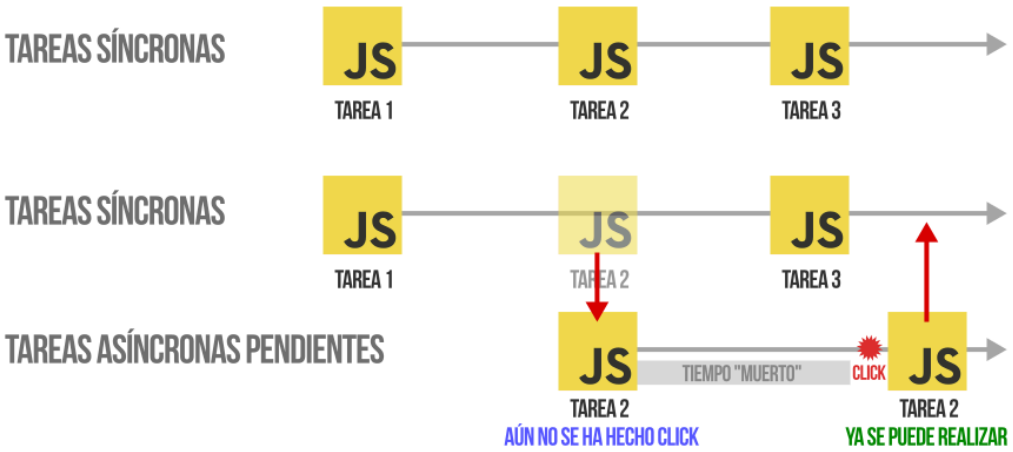
```
{  "empinfo" :
      {
            "employees" :  [
            {
                  "name" : "James Kirk",
                  "age" : 40,
            },
            {
                  "name" : "Jean-Luc Picard",
                  "age" : 45,
            },
            {
                  "name" : "Wesley Crusher",
                  "age" : 27,
            }
                              ]
      }
}
```

2. Introducción a la Asincronía

La asincronía significa que no tenemos que esperar a que una tarea termine para seguir con otras, como poner la pava para calentar el agua y mientras tanto preparar las tostadas.

¿Por qué la web necesita asincronía?
Porque no queremos que el sitio se “congele” mientras espera una respuesta del servidor o un archivo pesado. La web tiene que seguir funcionando aunque todavía no hayan llegado los datos.

Asincronía en JavaScript: Básicamente, el código sigue ejecutándose sin esperar a que la tarea asíncrona (como una petición a una API) se complete. Imaginá que hacés un pedido de comida con delivery: mientras el repartidor va hacia tu casa, vos seguís viendo tele. Ese sería el comportamiento asincrónico: la tarea de traer la comida no frena tus otras actividades.



3. Consumo de APIs con `fetch()`

3.1. **Uso de `fetch()` para Hacer Solicitudes HTTP**

- `fetch()` permite realizar solicitudes HTTP a una API de forma asíncrona. Esto nos permite pedir datos desde un servidor de forma sencilla, similar a enviar un mensaje y esperar la respuesta.

Ejemplo básico de fetch:

```
fetch('https://fakestoreapi.com/products')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error('Error al obtener datos:', error));
```

3.2. **Procesamiento de la Respuesta en JSON**

- Usamos `response.json()` para convertir los datos en un objeto JSON, lo cual facilita el procesamiento en JavaScript.

Ejemplo:

```
fetch('https://fakestoreapi.com/products')
  .then(response => response.json())
  .then(data => {
      // Aquí se procesan los datos obtenidos y se integran en el DOM
  });
```

¿Qué pasa en este código?

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

3.3. **Manejo de Errores con `.catch()` y Códigos 400/500**

- Cuando una solicitud a la API falla, es esencial manejar el error para que el usuario reciba un mensaje claro.
- **Errores Comunes**:
    - **Errores 400**: Problemas con la solicitud, como parámetros incorrectos o falta de permisos. Ejemplo: 404 (No Encontrado), 401 (No Autorizado).
    - **Errores 500**: Problemas en el servidor. Estos errores son más difíciles de predecir, ya que ocurren en el lado del servidor.

4XX Client Error	
400	Bad Request
401	Unauthorized
402	Payment Required
403	Forbidden
404	Not Found
405	Method Not Allowed
406	Not Acceptable
407	Proxy Authentication Required
408	Request Timeout

5XX Server Error	
500	Internal Server Error
501	Not Implemented
502	Bad Gateway
503	Service Unavailable
504	Gateway Timeout
505	HTTP Version Not Supported
506	Variant Also Negotiates
507	Insufficient Storage
508	Loop Detected
510	Not Extended
511	Network Authentication Required
599	Network Connect Timeout Error

Manejo de Errores:

🛑Cuando hacés un `fetch` para traer datos, la respuesta llega igual aunque el servidor diga “no encontré nada” (por ejemplo, un 404) o “me rompí” (un 500).

Por eso **no alcanza** con usar solo `.catch`, porque ese solo atrapa errores de red (como que no tengas conexión).

```
fetch('https://fakestoreapi.com/products')
  .then(response => {
      if (!response.ok) {
          throw new Error(`HTTP error! Status: ${response.status}`);
      }
      return response.json();
  })
  .then(data => {
      // Procesamiento de datos
  })
  .catch(error => {
      console.error('Error en la comunicación con la API:', error);
      // Aquí podrías mostrar un mensaje de error al usuario
  });
```

Explicación:

- Primero hacemos la petición con `fetch`.
- Después, chequeamos `response.ok`: esto vale *false* si el servidor devuelve un status 400 o 500, aunque la respuesta haya llegado.
- Si no está OK, lanzamos un error manual con `throw new Error(...)` para que el flujo pase al `.catch`.
- Si está OK, transformamos la respuesta con `response.json()`.
- Finalmente, en el `.catch`, capturamos cualquier problema: ya sea errores de red o errores que nosotros lanzamos con `throw`.

4. Renderizado Dinámico de Productos

Para esta práctica utilizaremos un sitio el cual simula diferentes API Rest para implementarlo en el proyecto.
https://fakestoreapi.com/

4.1. **Creación de Contenedor en HTML para Productos**

En el HTML podemos crear un `div` con id `productos-container` que va a ser el espacio donde se insertarán las tarjetas de producto.

```
<div id="productos-container"></div>
```

4.2. **Procesamiento y Estructura de Datos en JavaScript**

- Una vez que tenemos los datos de la API transformados en JSON, usamos `forEach` para recorrerlos y preparar el contenido que queremos mostrar: imagen, título, precio.

4.3. **Integración de Productos en el DOM Usando JavaScript**

Con `innerHTML` podemos ir agregando cada tarjeta de producto dentro del contenedor que creamos, para que el usuario vea el catálogo dinámicamente sin recargar la página.

```
fetch('https://fakestoreapi.com/products')
 .then(response => response.json())
 .then(data => {
     const contenedor =
document.getElementById("productos-container");
     data.forEach(producto => {
         const productoCard = `
             <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>
         `;
         contenedor.innerHTML += productoCard;
     });
 });
```

💭**Storytelling**

Consumir APIs y Carrito Dinámico con Fetch y LocalStorage

Tomás (Desarrollador Senior)

¡Vamos por más! Ahora que ya tenés una buena base en JavaScript, es momento de dar un gran paso hacia la programación moderna: trabajar con datos externos y manejar operaciones asíncronas. Esto es clave para crear aplicaciones reales que se conectan a servicios en la nube.

Ejercicio práctico #1:

Consumir una API REST con fetch

Lucía (Product Owner)

Queremos que tu tienda online muestre productos reales obtenidos desde una API pública. Para eso, usarás la función `fetch()` para traer los datos, mostrarlos en pantalla y manejar cualquier error que pueda aparecer.

Pasos a seguir:

1. Elegí una API pública para probar, por ejemplo:
   `https://fakestoreapi.com/products`

2. En tu archivo HTML, creá un contenedor donde se mostrarán los productos, por ejemplo:
   `<div id="productos-container"></div>`

3. En un archivo JavaScript separado o dentro de un `<script>`:

- Usá `fetch()` para obtener los productos desde la API.
- Convertí la respuesta a JSON para poder trabajar con los datos.
- Recorré la lista de productos con un bucle `.forEach()` y generá las tarjetas para mostrarlos en el contenedor del HTML.
- Agregá manejo de errores con `.catch()`, mostrando un mensaje claro si algo falla (alert o texto en la página).

4. Probá recargar la página para verificar que los productos se carguen correctamente.

Tips de Tomás:

Usá un diseño simple para las tarjetas, con imagen, título y precio.

Prepará la estructura para añadir botones de “Añadir al carrito” después.

Mantener el JavaScript separado en archivos externos mejora la organización y mantenimiento.

Ejercicio práctico #2:

Carrito dinámico con productos de la API

Lucía (Product Owner)

Ahora vamos a hacer que tu tienda permita agregar productos al carrito, y que este se mantenga guardado aunque recargues la página. Para esto, combinaremos `fetch()`, la manipulación del DOM y `localStorage`.

Pasos a seguir:

1. Partí del ejercicio anterior donde ya tienes los productos cargados con fetch.

2. Añadí un botón “Añadir al carrito” en cada tarjeta de producto.

3. Implementá funciones para:

- Recuperar el carrito guardado en `localStorage` o crear uno vacío si no existe.
- Añadir productos nuevos al carrito (podés guardar solo el ID o el objeto completo).
- Guardar el carrito actualizado en `localStorage`.
- Mostrar la cantidad total de productos en el carrito en un elemento visible de la página (por ejemplo, un `<div id="contador-carrito"></div>`).
- Actualizar ese contador cada vez que se agregue un producto.
- Al cargar la página, inicializar el contador leyendo el carrito guardado en `localStorage`.

4. Confirmá cada agregado al carrito con un mensaje breve (por ejemplo un `alert()`).

Tips de Tomás:

Usá `innerHTML` o `createElement` para generar dinámicamente las tarjetas y botones.

Recordá siempre convertir los objetos a texto con `JSON.stringify()` para guardarlos, y `JSON.parse()` para leerlos.

🛑No rehagas la carga de productos del ejercicio 1, construí sobre esa base para mantener orden y claridad.

Resolución de los ejercicios:

```
<!DOCTYPE html>

<html lang="es">

<head>

  <meta charset="UTF-8" />

  <meta name="viewport" content="width=device-width,
initial-scale=1.0"/>

  <title>Carrito con API</title>

  <style>

    body {

      font-family: Arial, sans-serif;

    }

    .producto-card {
```

```
    border: 1px solid #ddd;
    margin: 10px;
    padding: 10px;
    width: 200px;
    display: inline-block;
    vertical-align: top;
}
.producto-card img {
    max-width: 100%;
    height: 150px;
    object-fit: contain;
}
#contador-carrito {
    margin: 20px;
    font-weight: bold;
}
#vaciar-carrito {
    margin: 10px 20px;
    padding: 8px 15px;
    background-color: red;
    color: white;
    border: none;
    cursor: pointer;
}
#vaciar-carrito:hover {
```

```
      background-color: darkred;
    }
    /* mensaje de carga */
    #mensaje-cargando {
      font-style: italic;
      color: #555;
      margin: 20px;
    }
  </style>
</head>
<body>
  <h1>Tienda Online</h1>
  <!-- CONTADOR DE PRODUCTOS EN EL CARRITO -->
  <div id="contador-carrito">Productos en carrito: 0</div>
  <!-- BOTÓN PARA VACIAR EL CARRITO -->
  <button id="vaciar-carrito">Vaciar Carrito</button>
  <!-- MENSAJE DE CARGA -->
  <div id="mensaje-cargando">Cargando productos, por favor
espere...</div>
  <!-- CONTENEDOR DE LOS PRODUCTOS -->
  <div id="productos-container"></div>
  <script>
    // ============================================
    // 1 OBTENER PRODUCTOS DESDE API
    // ============================================
```

```
    fetch('https://fakestoreapi.com/products')
      .then(response => response.json())
      .then(productos => {
        const contenedor =
document.getElementById('productos-container');
        const mensajeCargando =
document.getElementById('mensaje-cargando');
        // Ocultamos el mensaje de carga porque ya llegaron los
productos
        mensajeCargando.style.display = "none";
        productos.forEach(producto => {
          // CREAR TARJETA DEL PRODUCTO
          const card = document.createElement('div');
          card.className = 'producto-card';
          card.innerHTML = `
            <img src="${producto.image}" alt="${producto.title}">
            <h3>${producto.title}</h3>
            <p>$${producto.price}</p>
            <button data-id="${producto.id}">Añadir al carrito</button>
          `;
          contenedor.appendChild(card);
        });
        // Activar eventos
        cargarEventosAgregar();
        actualizarContador();
      })
```

```
      .catch(error => {
        // Mostrar mensaje de error
        document.getElementById('mensaje-cargando').textContent =
          "Ocurrió un error al obtener los productos: " +
error.message;
      });
    // ==========================================
    // 2 FUNCIÓN PARA OBTENER EL CARRITO ACTUAL
    // ==========================================
    function obtenerCarrito() {
      return JSON.parse(localStorage.getItem('carrito')) || [];
    }
    // ================================================
    // 3 FUNCIÓN PARA GUARDAR EL CARRITO EN STORAGE
    // ================================================
    function guardarCarrito(carrito) {
      localStorage.setItem('carrito', JSON.stringify(carrito));
    }
    // ========================================
    // 4 FUNCIÓN PARA AGREGAR PRODUCTO AL CARRITO
    // ========================================
    function agregarAlCarrito(id) {
      let carrito = obtenerCarrito();
      carrito.push(id);  // Podrías guardar el objeto completo si
quisieras
      guardarCarrito(carrito);
```

```
      actualizarContador();

      alert('Producto agregado al carrito');

    }

    // ========================================================

    // 5 FUNCIÓN PARA ACTUALIZAR EL CONTADOR EN PANTALLA

    // ========================================================

    const contadorCarrito =
document.getElementById('contador-carrito');

    function actualizarContador() {

      const carrito = obtenerCarrito();

      contadorCarrito.textContent = `Productos en carrito:
${carrito.length}`;

    }

    // ============================================================

    // 6 FUNCIÓN PARA ACTIVAR EL EVENTO DE AÑADIR A TODOS LOS BOTONES

    // ============================================================

    function cargarEventosAgregar() {

      const botones = document.querySelectorAll('#productos-container
button');

      botones.forEach(boton => {

        boton.addEventListener('click', () => {

          agregarAlCarrito(boton.getAttribute('data-id'));

        });

      });

    }

    // ========================================================
```

```
    // 7 FUNCIÓN PARA VACIAR EL CARRITO Y RESETEAR CONTADOR
    // ======================================================
    document.getElementById('vaciar-carrito').addEventListener('click',
() => {
      localStorage.removeItem('carrito');
      actualizarContador();
      alert('Carrito vaciado correctamente');
    });
  </script>
</body>
</html>
```

Buenos Aires aprende

Agencia de Habilidades para el Futuro

Buenos Aires Ciudad