**Guía Avanzada: Creando Aplicaciones Web Dinámicas con APIs y JavaScript**

Esta guía completa sobre cómo conectar tus páginas web a fuentes de datos externas. Aprenderás a utilizar JavaScript para solicitar información a servidores remotos a través de APIs y a usar esa información para crear contenido dinámico y aplicaciones interactivas.

**1. Fundamentos: APIs y la Función fetch**

**¿Qué es una API?**

Una **API (Interfaz de Programación de Aplicaciones)** es un intermediario que permite que dos aplicaciones se comuniquen. En nuestro caso, nuestra aplicación de JavaScript (el *frontend*) le pedirá datos a un servidor (el *backend*) a través de su API. La respuesta más común de las APIs web viene en formato **JSON (JavaScript Object Notation)**, un formato de texto ligero que es muy fácil de procesar con JavaScript.

**La Herramienta Esencial: fetch**

fetch() es la función moderna de JavaScript para realizar peticiones de red. Su característica principal es que está **basada en Promesas**.

**Mini-Documentación: El Flujo de fetch**

1. **fetch('URL')**: Lanzas la petición. Esto devuelve inmediatamente una Promesa.
2. **Primer .then(respuesta => ...)**: La promesa se cumple cuando el servidor responde. El valor que recibes aquí no son los datos, sino un objeto Response. Este objeto contiene información sobre la respuesta (como el código de estado HTTP). Para obtener los datos, necesitas usar un método para parsear el cuerpo de la respuesta, como respuesta.json().
3. **respuesta.json()**: Este método también devuelve una Promesa.
4. **Segundo .then(datos => ...)**: Esta segunda promesa se cumple cuando el cuerpo de la respuesta ha sido completamente leído y parseado a JSON. ¡Aquí es donde finalmente tienes tus datos como un objeto o array de JavaScript!
5. **.catch(error => ...)**: Si en cualquier punto del proceso ocurre un error (problemas de red, URL incorrecta), la promesa se rechaza y el control pasa al bloque .catch.

**Ejemplo Básico Comentado:**

// app.js  
fetch('https://jsonplaceholder.typicode.com/todos/1') // URL de una API de prueba  
  .then(response => {  
    // Verificamos si la respuesta fue exitosa (ej. status 200 OK)  
    if (!response.ok) {  
      // Si no fue exitosa, lanzamos un error para que lo capture el .catch()  
      throw new Error('La respuesta de la red no fue OK');  
    }  
    // Si fue exitosa, parseamos el JSON. Esto devuelve otra promesa.  
    return response.json();  
  })  
  .then(data => {  
    // ¡Aquí tenemos los datos!  
    console.log("Datos recibidos:", data);  
    // data es un objeto como: { userId: 1, id: 1, title: 'delectus aut autem', completed: false }  
  })  
  .catch(error => {  
    // Capturamos cualquier error de red o el que lanzamos manualmente  
    console.error('Hubo un problema con la operación fetch:', error);  
  });

**2. Peticiones Básicas y Renderizado Simple**

**Objetivo:** Acostumbrarse al flujo de fetch y a mostrar un único dato en la página.

**Ejercicio 2.1: El Nombre de un Pokémon**

API: [PokéAPI](https://pokeapi.co/)

Tarea:

1. **HTML:** Crea un index.html con <h2 id="nombre-pokemon">Cargando...</h2>.
2. **JS:** Crea un app.js. Usa fetch para pedir datos del Pokémon "charmander" a la URL https://pokeapi.co/api/v2/pokemon/charmander.
3. Cuando obtengas los datos, selecciona el h2 y cambia su textContent por el nombre del Pokémon (que se encuentra en la propiedad name del objeto de datos).

**Ejercicio 2.2: Un Chiste Aleatorio**

API: [Chuck Norris API](https://api.chucknorris.io/)

Tarea:

1. **HTML:** Añade un <p id="chiste">Cargando chiste...</p>.
2. **JS:** Usa fetch para pedir un chiste aleatorio a https://api.chucknorris.io/jokes/random.
3. La respuesta JSON tiene una propiedad value que contiene el chiste. Muestra este chiste en el párrafo #chiste.

**Ejercicio 2.3: Un DATO de un País**

API: [REST Countries](https://restcountries.com/)

Tarea:

1. **HTML:** Añade un div con id="info-pais".
2. **JS:** Usa fetch para obtener información sobre Argentina: https://restcountries.com/v3.1/name/argentina.
3. La API devuelve un **array** con un solo objeto. Deberás acceder al primer elemento (datos[0]).
4. Muestra la capital del país (propiedad capital) dentro del div#info-pais.

**3. Renderizando Objetos Complejos**

**Objetivo:** Aprender a navegar por objetos JSON anidados y a construir componentes HTML más complejos dinámicamente.

**Ejercicio 3.1: Tarjeta de Pokémon Mejorada**

**Tarea:** Vamos a mejorar el ejercicio 2.1 creando una tarjeta más completa.

1. **HTML:** Crea un div vacío con id="pokemon-card".
2. **JS:** Haz la llamada a la API para obtener los datos de un Pokémon (ej: pikachu).
3. Cuando tengas los datos, limpia el contenido del div (innerHTML = '').
4. Crea dinámicamente y añade al div los siguientes elementos:

* Un h2 con el nombre (datos.name).
* Un img con la imagen principal. La URL está en datos.sprites.front\_default.
* Un p con el tipo principal del Pokémon. Los tipos están en un array: datos.types[0].type.name.
* Un p con una de sus habilidades: datos.abilities[0].ability.name.

**Ejemplo de acceso a datos anidados:**

// Suponiendo que 'datos' es el objeto Pokémon  
const nombre = datos.name; // 'pikachu'  
const urlImagen = datos.sprites.front\_default; // 'https://...'  
const primerTipo = datos.types[0].type.name; // 'electric'

**Ejercicio 3.2: Perfil de Personaje de Rick and Morty**

API: [Rick and Morty API](https://rickandmortyapi.com/)

Tarea: Crea una tarjeta de perfil para un personaje.

1. **HTML:** Crea un div vacío con id="rick-morty-card".
2. **JS:** Haz fetch a la URL de un personaje, por ejemplo: https://rickandmortyapi.com/api/character/2 (Morty Smith).
3. Renderiza una tarjeta que muestre:

* Nombre del personaje (name).
* Imagen (image).
* Especie (species).
* **(Desafío)** El nombre de su lugar de origen. Este dato está anidado: origin.name.

**4. async/await: La Sintaxis Moderna**

**Objetivo:** Entender y utilizar async/await para escribir código asíncrono más legible.

Mini-Documentación:

async/await es una forma de trabajar con promesas que hace que tu código parezca síncrono.

* Una función declarada con async siempre devuelve una promesa.
* Dentro de una función async, puedes usar await para pausar la ejecución hasta que una promesa se resuelva.
* El manejo de errores se hace con try...catch, lo cual es muy natural en JavaScript.

**Ejemplo de refactorización:**

// Con .then()  
function obtenerDatosConThen() {  
  fetch('url')  
    .then(res => res.json())  
    .then(data => console.log(data))  
    .catch(err => console.error(err));  
}  
  
// Con async/await  
async function obtenerDatosConAsync() {  
  try {  
    const respuesta = await fetch('url');  
    const datos = await respuesta.json();  
    console.log(datos);  
  } catch (error) {  
    console.error(error);  
  }  
}

**Ejercicio 4.1: Refactorizar la Tarjeta de Pokémon**

**Tarea:** Toma el código del ejercicio 3.1 y reescríbelo dentro de una función async llamada mostrarPokemon(nombrePokemon). La función debe recibir el nombre de un pokémon como parámetro y construir la tarjeta. Llama a la función con "gengar".

**Ejercicio 4.2: Obtener Datos del Clima (Ficticio)**

**Tarea:** Escribe una función async llamada consultarClima(ciudad).

1. Esta función simulará una llamada a una API de clima.
2. Dentro de un bloque try...catch:

* Imprime en consola Consultando clima para ${ciudad}....
* Crea una promesa que se resuelva después de 2 segundos con un objeto de clima: { temperatura: 25, condicion: "Soleado" }. Usa await en esta promesa.
* Si la promesa se resuelve, imprime La temperatura en ${ciudad} es de ${clima.temperatura}°C y está ${clima.condicion}..
* En el catch, maneja cualquier posible error.

1. Llama a consultarClima("Buenos Aires").

**5. Renderizando Listas de Datos (Arrays)**

**Objetivo:** Obtener un array de datos de una API y renderizar un elemento por cada ítem del array.

**Ejercicio 5.1: Lista de Tareas**

API: [JSONPlaceholder](https://jsonplaceholder.typicode.com/)

Tarea: Obtener y mostrar una lista de "todos" (tareas).

1. **HTML:** Crea un <ul> vacío con id="lista-tareas".
2. JS:  
   a. Crea una función async mostrarTareas().  
   b. Haz fetch a https://jsonplaceholder.typicode.com/todos?\_limit=10 para obtener 10 tareas.  
   c. Obtén el array de tareas de la respuesta.  
   d. Selecciona el <ul>.  
   e. Usa un bucle (forEach o for...of) para recorrer el array de tareas.  
   f. Por cada tarea, crea un <li> y asigna su textContent al título de la tarea (tarea.title).  
   g. (Desafío) Añade una clase CSS diferente al <li> si la tarea está completada (tarea.completed es true). Por ejemplo, una clase .completada que tache el texto.  
   h. Añade cada <li> al <ul>.

**Ejercicio 5.2: Galería de Fotos de Gatos**

API: [The Cat API](https://thecatapi.com/)

Tarea: Crear una galería de imágenes aleatorias de gatos.

1. **HTML:** Crea un div con id="galeria-gatos" y estilos de grid para que se vea bien.
2. JS:  
   a. Crea una función async cargarGatos().  
   b. Haz fetch a la URL: https://api.thecatapi.com/v1/images/search?limit=9. Esto te dará un array de 9 objetos, cada uno con una URL de imagen.  
   c. Itera sobre el array de respuesta.  
   d. Por cada objeto gato en el array, crea un elemento <img>, establece su src a gato.url y añádelo al div de la galería.

**6. Interactividad del Usuario con APIs**

**Objetivo:** Conectar las acciones del usuario (clics, envíos de formularios) con llamadas dinámicas a las APIs.

**Ejercicio 6.1: Buscador de Personajes de Rick and Morty**

**Tarea:**

1. **HTML:** Crea un <input type="text" id="nombre-personaje">, un <button id="buscar-btn">Buscar</button>, y un <div id="resultado-busqueda"></div>.
2. JS:  
   a. Selecciona los tres elementos.  
   b. Añade un addEventListener al click del botón.  
   c. Dentro del callback, obtén el valor del input.  
   d. Construye la URL de la API de Rick and Morty para buscar por nombre: https://rickandmortyapi.com/api/character/?name=NOMBRE\_BUSCADO.  
   e. Llama a una función async que haga el fetch.  
   f. La API devolverá una lista de resultados. Recorre esa lista (datos.results) y crea una pequeña tarjeta (nombre e imagen) para cada personaje encontrado y muéstrala en el div#resultado-busqueda.  
   g. (Desafío) Si la API devuelve un error (porque el personaje no existe), muestra un mensaje "No se encontraron personajes con ese nombre" en el div de resultados. Limpia los resultados anteriores antes de cada nueva búsqueda.

**Ejercicio 6.2: Generador de Consejos**

API: [Advice Slip API](https://api.adviceslip.com/)

Tarea:

1. **HTML:** Crea un <p id="consejo-texto">Haz clic para un nuevo consejo</p> y un <button id="nuevo-consejo-btn">Obtener Consejo</button>.
2. JS:  
   a. Crea una función async obtenerConsejo() que haga fetch a https://api.adviceslip.com/advice.  
   b. La respuesta JSON tiene un objeto slip que contiene el advice.  
   c. Actualiza el textContent del párrafo con el consejo obtenido.  
   d. Añade un addEventListener al click del botón para que llame a esta función.  
   e. Llama a la función una vez cuando la página cargue para mostrar un consejo inicial.

**7. Preguntas Teóricas y de Investigación**

1. **Pregunta Teórica:** Describe el flujo de una promesa devuelta por fetch. Explica qué representa el objeto resuelto en el primer .then() y qué representa en el segundo, después de invocar respuesta.json().
2. **Pregunta Teórica:** ¿Cuál es la principal ventaja de usar async/await en lugar de la sintaxis de .then().catch()? ¿Hay alguna situación en la que preferirías usar .then()?
3. **Pregunta de Investigación:** ¿Qué es **CORS (Cross-Origin Resource Sharing)**? ¿Por qué a veces una petición fetch falla en tu proyecto local con un "error de CORS" y cómo se soluciona (desde el lado del servidor)?
4. **Pregunta de Investigación:** Las peticiones que hemos hecho son de tipo GET. Investiga qué otros **métodos HTTP** existen (menciona al menos 3 más) y para qué se usa comúnmente cada uno en el contexto de una API REST.
5. **Pregunta de Investigación:** ¿Qué es un **"API Key" (Clave de API)**? ¿Por qué muchas APIs públicas (como las de clima o mapas) requieren que te registres para obtener una?