APIs públicas y aplicaciones híbridas

En el desarrollo web actual, muy pocas aplicaciones viven aisladas. Lo habitual es que una aplicación consuma información de terceros: catálogos de productos, datos meteorológicos, información geográfica, contenidos multimedia, etc. Por ese motivo, comprender qué es una API pública, cómo se usa correctamente y qué limitaciones tiene es una competencia clave.

¿Qué es una API?

Una API (Application Programming Interface) es un mecanismo que permite que dos aplicaciones se comuniquen entre sí de forma controlada. En lugar de permitir el acceso directo a una base de datos o a los sistemas internos de una aplicación, se expone una interfaz intermedia que define qué se puede pedir, cómo se debe pedir y qué tipo de respuesta se obtendrá.

Desde el punto de vista del desarrollo web, una API actúa como un contrato. El servidor se compromete a responder a determinadas peticiones con datos estructurados, normalmente en formato JSON, y el cliente se compromete a realizar esas peticiones siguiendo unas reglas concretas.

Una analogía habitual es la del restaurante. El cliente no entra en la cocina ni decide cómo se preparan los platos. El cliente consulta una carta, hace un pedido al camarero y recibe el plato ya preparado. En este ejemplo, la carta sería la documentación de la API, el camarero la propia API y la cocina el sistema interno del servidor.

¿Qué es una API pública?

Una API pública es una API accesible desde Internet, pensada para ser utilizada por desarrolladores externos. No está limitada a una única aplicación ni a una red privada, sino que cualquiera puede consumirla siempre que respete las condiciones establecidas por el proveedor.

Las APIs públicas suelen utilizar el protocolo HTTP o HTTPS y se accede a ellas mediante URLs. Al realizar una petición a una de estas URLs, el servidor responde con datos estructurados que pueden ser procesados fácilmente por un programa.

En el ámbito educativo, las APIs públicas son especialmente útiles porque permiten trabajar con datos realistas sin necesidad de crear y mantener una base de datos propia desde el primer momento.

APIs gratuitas, freemium y de pago

No todas las APIs públicas funcionan del mismo modo ni ofrecen las mismas condiciones de uso.

Existen APIs completamente gratuitas, que no requieren registro ni autenticación. Suelen estar orientadas al aprendizaje, a pruebas o a prototipos. Ofrecen datos limitados y no garantizan estabilidad a largo plazo. La FakeStore API es un ejemplo claro de este tipo.

Otras APIs funcionan con un modelo freemium. Permiten un uso gratuito inicial, pero requieren registrarse y obtener una clave de acceso, conocida como API Key. Esta clave identifica a la aplicación que consume la API y permite al proveedor controlar el número de peticiones realizadas.

Por último, existen APIs exclusivamente de pago, orientadas a entornos profesionales y empresariales, donde se paga en función del número de peticiones o del volumen de datos consumidos.

Cómo saber si una API es gratuita

La única forma fiable de saber si una API es gratuita, de pago o tiene limitaciones es consultar su documentación oficial. En ella se especifican aspectos como la necesidad de autenticación, los límites de uso, las condiciones legales y el propósito de la API.

En el caso de FakeStore API, su documentación indica claramente que se trata de una API gratuita pensada para pruebas y aprendizaje, sin necesidad de API Key. Esto la convierte en una opción ideal para el aula.

EJERCICIO: Investiga al menos 5 APIs que provean datos de algún tema que te interese. Explícalas en clase, diferenciando si son gratuitas, freemium o de pago.

Uso correcto de APIs en aplicaciones híbridas

Un error frecuente en proyectos iniciales es consumir directamente una API pública desde el frontend. Aunque técnicamente es posible, no es una práctica recomendable.

La arquitectura correcta consiste en situar el backend como intermediario entre el frontend y la API externa. De este modo, el frontend se comunica únicamente con nuestro servidor, y es el servidor quien se encarga de hablar con la API pública.

Este enfoque mejora la seguridad, permite controlar errores, adaptar los datos al formato que necesita la aplicación y evita exponer claves o detalles internos.

Axios como herramienta para consumir APIs

Para realizar peticiones HTTP desde Node.js se utilizan librerías específicas. En este curso se emplea axios, una librería muy extendida que simplifica enormemente el trabajo con peticiones asíncronas.

Axios permite realizar peticiones GET, POST y otros métodos HTTP y devuelve siempre los datos en una estructura clara y predecible.

El patrón DAO (Data Access Object) se utiliza para separar la lógica de acceso a los datos del resto de la aplicación. Esto significa que el controlador o las rutas no necesitan saber si los datos proceden de una base de datos, de un fichero o de una API pública.

Aplicar este patrón incluso cuando se trabaja con APIs externas es una buena práctica, ya que facilita cambios futuros y mejora la mantenibilidad del código.

Ejemplo práctico completo

El siguiente ejemplo muestra cómo consumir una API pública de productos utilizando Node.js, axios y el patrón DAO.

```
const axios = require("axios");

class ProductoDAO {

  static async buscarPorTexto(texto) {

    const response = await axios.get("https://fakestoreapi.com/products");

    const productos = response.data;


    const filtrados = productos.filter(p =>

      p.title.toLowerCase().includes(texto.toLowerCase())

    );


    return filtrados.map(p => ({

      id: p.id,

      nombre: p.title,

      precio: p.price

    }));  } }

module.exports = ProductoDAO;
```

Análisis fragmento a fragmento:

```
const axios = require("axios");

class ProductoDAO {

  static async buscarPorTexto(texto) {
```

En primer lugar se importa la librería axios, que será la encargada de realizar la petición HTTP. A continuación se define la clase ProductoDAO, que encapsula todo el acceso a los datos de productos.

El método buscarPorTexto es estático y asíncrono, lo que permite llamarlo sin instanciar la clase y utilizar await para esperar la respuesta de la API.

```
const response = await axios.get("https://fakestoreapi.com/products");

    const productos = response.data;
```

La llamada a la API se realiza mediante una petición GET a la URL de FakeStore. La respuesta contiene los datos en formato JSON, que se almacenan en la variable productos.

```
const filtrados = productos.filter(p =>
    p.title.toLowerCase().includes(texto.toLowerCase())
);
```

Dado que esta API no ofrece búsquedas por texto, el filtrado se realiza de forma local en el backend. Este proceso es equivalente a una cláusula LIKE en SQL, donde se buscan coincidencias parciales dentro del título del producto.

```
return filtrados.map(p => ({
  id: p.id,
  nombre: p.title,
  precio: p.price
}));
```

Finalmente, los datos se transforman para adaptarlos al frontend. Se seleccionan únicamente los campos necesarios y se renombran para mantener coherencia con el idioma y el diseño de la aplicación.

Limitaciones y problemas reales

Trabajar con APIs públicas implica aceptar ciertas limitaciones. El servicio puede dejar de estar disponible, cambiar el formato de los datos o imponer límites de uso. Por este motivo, nunca se debe depender completamente de una API externa sin mecanismos de control.

En aplicaciones reales es fundamental gestionar errores, prever tiempos de espera elevados y, cuando sea posible, almacenar datos en caché.

Consumo avanzado de APIs públicas en Node.js

Cuando trabajamos con APIs públicas en proyectos reales, además de realizar peticiones y filtrar datos, hay algunos aspectos que todo desarrollador debe conocer:

- **Autenticación con API Key**

Muchas APIs requieren una clave de acceso (API Key) que identifica a la aplicación. Esta clave nunca debe exponerse en el frontend; siempre se envía desde el backend.

El fichero .env es un archivo de texto que contiene variables de entorno de nuestra aplicación.

Sirve para guardar API Keys, contraseñas o configuraciones sensibles

Nunca debe subirse a GitHub ni enviarse al frontend

Ejemplo de contenido de un .env:

OPENWEATHER_API_KEY=123456abcdef

PORT=3000

Cada línea es una variable con estructura NOMBRE=VALOR

En Node.js podemos leer estas variables con:

process.env.OPENWEATHER_API_KEY

Eso permite cambiar la clave sin tocar el código, tener claves distintas en cada ordenador, y proteger datos sensibles

Vamos a crear este fichero .env siempre en la raíz del proyecto, al mismo nivel que app.js.

Ejemplo de un DAO usando la API de OpenWeatherMap:

```
const axios = require("axios");

class ClimaDAO {

  static async obtenerClima(ciudad) {

    const apiKey = process.env.OPENWEATHER_API_KEY; // guardada en variables de entorno

    const url = `https://api.openweathermap.org/data/2.5/weather?q=${ciudad}&appid=${apiKey}&units=metric`;

    try {

      const response = await axios.get(url);

      const data = response.data;

      return {

        ciudad: data.name,

        temperatura: data.main.temp,

        descripcion: data.weather[0].description   };

    } catch (error) {

      // Manejo de errores HTTP

      if (error.response) {

        // La API respondió con un código de error

        throw new Error(`Error API: ${error.response.status} ${error.response.statusText}`);

      } else {

        // Error de red o de Axios

        throw new Error(`Error de conexión: ${error.message}`); }   }  }}

module.exports = ClimaDAO;
```

Análisis de los puntos más importantes:

const apiKey = process.env.OPENWEATHER_API_KEY;

Como ya hemos dicho, la API Key no está escrita en el código. Se obtiene desde una variable de entorno, normalmente definida en un archivo .env o en el sistema.

Esto evita que cualquiera pueda usar nuestra cuota de peticiones.

const url = `https://api.openweathermap.org/data/2.5/weather?q=${ciudad}&appid=${apiKey}&units=metric`;

- Si leemos la documentación de la API, veremos que se construye dinámicamente la URL:
    - q → ciudad buscada
    - appid → API Key
    - units=metric → temperatura en grados Celsius
- Muchas APIs funcionan exactamente así: endpoint + parámetros.

const response = await axios.get(url);

const data = response.data;

- Axios devuelve un objeto complejo con:
    - status
    - headers
    - data → lo que nos interesa (JSON de la API)
- await detiene la ejecución hasta recibir respuesta.

return {

ciudad: data.name,

temperatura: data.main.temp,

descripcion: data.weather[0].description

};

- Nunca se devuelve el JSON completo de la API.
- Se seleccionan solo los datos necesarios y se adaptan al frontend.
- Esto desacopla nuestra app de posibles cambios en la API externa.

catch (error) {

```
    if (error.response) {
        throw new Error(`Error API: ${error.response.status} ${error.response.statusText}`);
    } else {
        throw new Error(`Error de conexión: ${error.message}`);
    }
}
```

Cuando se consume una API pública desde el backend, es imprescindible gestionar los errores de forma controlada. Las peticiones HTTP pueden fallar por múltiples motivos: la API puede estar caída, la clave puede ser incorrecta o se puede haber superado el límite de peticiones.

La variable error es proporcionada por axios cuando una petición falla. Axios distingue entre errores de respuesta y errores de conexión.

La propiedad error.response existe cuando el servidor de la API sí respondió, pero lo hizo con un código de error HTTP, como por ejemplo:

- 401 (API Key incorrecta o no autorizada)
- 404 (recurso no encontrado)
- 429 (demasiadas peticiones)
- 500 (error interno del servidor de la API)

En estos casos, el problema no es de red, sino que la API ha rechazado la petición o no ha podido procesarla correctamente.

Si error.response no existe, normalmente significa que la petición no llegó a completarse. Las causas más habituales son problemas de red, timeout de la petición, el servidor de la API no está accesible, …

Por ese motivo se pueden diferenciar ambos casos y se generan mensajes de error distintos.

Qué significa throw (teníamos pendiente profundizar en ello)

La palabra clave throw se utiliza en JavaScript para lanzar un error. Cuando se ejecuta un throw, el flujo normal del programa se detiene inmediatamente y se pasa el control al siguiente catch disponible en la cadena de llamadas, en este caso en el controlador.

Al hacer:

```
throw new Error("mensaje de error");
```

ocurre lo siguiente:

- Se crea un objeto de tipo Error
- Se interrumpe la ejecución del método
- El error se propaga hasta el controlador que llamó al DAO

Esto permite separar responsabilidades: el DAO detecta el problema, pero no decide cómo se comunica al usuario final.

Diferencia entre throw y return

Aunque ambos finalizan la ejecución de una función, **no hacen lo mismo**.

return se utiliza para devolver un valor cuando todo ha ido bien. El flujo del programa continúa de forma normal en la función que llamó al método.

return datos;

Significa: *"la operación ha sido correcta y devuelvo este resultado"*.

En cambio, throw se utiliza cuando ocurre un problema y no se puede continuar con la operación.

throw new Error("Algo ha fallado");

Significa: *"la operación no se ha podido completar y notifico un error"*.

Diferencias clave:

- return devuelve un valor y la ejecución continúa normalmente.
- throw lanza un error y obliga a que alguien lo capture con try/catch.
- Un throw no devuelve datos, solo interrumpe el flujo.
- Un return no indica error, indica éxito.

- **Consumo de APIs públicas con paginación**

Muchas APIs públicas no devuelven todos los datos de una sola vez. En lugar de eso, utilizan un sistema llamado paginación, mediante el cual los resultados se dividen en bloques o páginas. Esto se hace para evitar respuestas demasiado grandes, reducir el consumo de red y mejorar el rendimiento.

La PokéAPI es un ejemplo muy claro y didáctico de este tipo de APIs, ya que obliga a trabajar con paginación desde el principio.

Funcionamiento de la paginación en la PokéAPI

La PokéAPI utiliza dos parámetros principales para controlar la paginación:

- **limit**: indica el número de elementos que se van a devolver.
- **offset**: indica desde qué posición del listado se empieza a devolver datos.

Ejemplos prácticos:

- limit=10&offset=0 → devuelve los primeros 10 Pokémon.
- limit=10&offset=10 → devuelve los Pokémon del 11 al 20.
- limit=10&offset=20 → devuelve los Pokémon del 21 al 30.

Gracias a estos parámetros, el cliente puede avanzar o retroceder por el listado sin necesidad de descargar todos los datos.

DAO con paginación

```
const axios = require("axios");


class PokemonDAO {
  static async listarPokemons(limit = 10, offset = 0) {
    const url = `https://pokeapi.co/api/v2/pokemon?limit=${limit}&offset=${offset}`;
    try {
      const response = await axios.get(url);


      return response.data.results.map(p => ({
        nombre: p.name,
        url: p.url
      }));
    } catch (error) {
      throw new Error("No se pudieron obtener los Pokémon");    } }


module.exports = PokemonDAO;
```

Explicación clave del DAO

- El DAO no sabe nada del frontend ni de HTTP.
- Solo recibe parámetros (limit y offset) y se encarga de obtener datos.
- Transforma la respuesta de la API externa a un formato más simple.
- Si ocurre un error, lanza una excepción, pero no responde HTTP.
- Su única responsabilidad es el acceso a datos.

En el aula virtual veremos completa la aplicación, llamada pokemon_paginacion.zip

- **Caché simple en el backend**

Cuando una aplicación consume una API pública, cada petición implica tiempo de espera, dependencia de un servicio externo y, en muchos casos, límites de uso impuestos por dicha API. Si varios usuarios realizan la misma petición en un intervalo corto de tiempo, repetir constantemente la llamada a la API externa resulta ineficiente.
Para evitar este problema, es habitual implementar un sistema de caché en el backend.

La idea principal de la caché es guardar temporalmente en memoria los resultados de ciertas peticiones y reutilizarlos mientras sigan siendo válidos. De esta forma se mejora el rendimiento general de la aplicación y se reduce el número de llamadas a la API externa, sin modificar el comportamiento visible para el usuario.

Caché en memoria

Una caché en memoria es un almacenamiento temporal que reside directamente en la memoria RAM del servidor mientras la aplicación está en ejecución. Su principal ventaja es la velocidad, ya que el acceso a memoria es mucho más rápido que una petición HTTP a un servicio externo. Como contrapartida, los datos se pierden cuando el servidor se reinicia, lo cual la hace ideal para ejemplos educativos, pruebas o aplicaciones sencillas.

Uso de NodeCache

Para implementar este tipo de caché se utiliza la librería **NodeCache**, que permite guardar datos en memoria junto con un tiempo de vida asociado. Este tiempo de vida, conocido como TTL, indica cuánto tiempo se considera válido un dato antes de ser eliminado automáticamente.

```
const NodeCache = require("node-cache");

const cache = new NodeCache({ stdTTL: 60 }); // caché de 60 segundos
```

En este caso, los datos se mantendrán en caché durante 60 segundos. Si durante ese intervalo se repite una petición idéntica, el sistema reutilizará los datos almacenados sin volver a llamar a la API externa.

DAO con caché

La caché se implementa dentro del DAO porque esta capa es la responsable del acceso a los datos. El controlador no debería saber si los datos provienen directamente de una API externa o de una caché intermedia. De este modo se mantiene una arquitectura limpia y bien separada por responsabilidades.

```
const axios = require("axios");

const NodeCache = require("node-cache");


const cache = new NodeCache({ stdTTL: 60 });


class PokemonDAOConCache {

  static async listarPokemons(limit = 10, offset = 0) {
```

```
        const key = `pokemons-${limit}-${offset}`;

        if (cache.has(key)) {
            return cache.get(key);
        }

        const url = `https://pokeapi.co/api/v2/pokemon?limit=${limit}&offset=${offset}`;

        try {
            const response = await axios.get(url);

            const resultado = response.data.results.map(p => ({
                nombre: p.name,
                url: p.url
            }));

            cache.set(key, resultado);
            return resultado;

        } catch (error) {
            throw new Error("No se pudieron obtener los Pokémon");
        }
    }
}

module.exports = PokemonDAOConCache;
```

Explicación paso a paso de los fragmentos relativos a la caché

const key = `pokemons-${limit}-${offset}`;

- Cada combinación de limit y offset genera una clave única.
- Ejemplos:
    - pokemons-10-0
    - pokemons-10-10
    - pokemons-20-0

Esto permite cachear cada página por separado.

```
if (cache.has(key)) {
   return cache.get(key);
}
```

- Si los datos ya están en caché:
    - No se llama a la API externa.
    - Se devuelve el resultado inmediatamente.
- Esto hace la respuesta **mucho más rápida**.

const response = await axios.get(url);

- Solo se ejecuta si los datos no están cacheados o han expirado.
- Se reduce drásticamente el número de peticiones externas.

cache.set(key, resultado);

- El resultado se guarda en memoria.
- Permanecerá disponible durante 60 segundos.
- Las siguientes peticiones reutilizarán estos datos.

Para que todo funcione correctamente, únicamente es necesario sustituir el DAO original por el DAO con caché. No es necesario modificar ni las rutas, ni el frontend, ni los parámetros de paginación. Desde el punto de vista del cliente, la aplicación se comporta exactamente igual, aunque internamente sea mucho más eficiente.