Desarrollo de aplicaciones JS

Luis Valencia Cabrera (lvalencia@us.es)

Research Group on Natural Computing
Departamento de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla

13-04-2026, Bases de Datos

Índice

1. Introducción
2. Asincronía
3. Comunicación con servidor
4. Servidor

Contexto: De Almacén de Datos a Aplicaciones Reales

En el primer cuatrimestre (Bases de Datos):

- Nos centramos en el *corazón* del sistema: el **Dato**.
- Diseñamos estructuras robustas, relaciones y reglas de integridad.
- El dato existía de forma estática, consultado por nosotros (los mismos diseñadores de la BD).

En el segundo (Desarrollo de Aplicaciones Web):

- Elementos constituyentes básicos (CSS, HTML, JS)
- Vamos a construir el *cuerpo* que rodea a ese corazón: la **Aplicación**.
- El dato ahora debe fluir hacia un usuario ajeno a SQL o MongoDB.
- Necesitamos "puentes" que conecten nuestra base de datos con la pantalla del navegador.

Las tres capas de la arquitectura

Para que el dato llegue del disco a la pantalla, pasamos tres niveles:

1. **Persistencia (Memoria):** Es la BD (MySQL/Mongo). El dato "persiste" aunque apaguemos el programa.
2. **Negocio o Aplicación (Cerebro, *Backend*):** El servidor (Node.js). Procesa la lógica, aplica seguridad y consulta la base de datos.
3. **Presentación (Cara, *Frontend*):** El navegador. La interfaz HTML+CSS+JS con la que el usuario interactúa.

Pregunta clave

Hemos visto la capa de persistencia y la de presentación. Ahora bien: **¿Qué nos falta para conectar estas capas de forma eficiente y segura?** Necesitamos ***dotar del backend***, que ***interactuará con la BD***, y de un ***mecanismo para comunicar*** **Frontend** con **Backend**.

El Modelo Cliente-Servidor

Es la arquitectura fundamental para conectar nuestras capas:

- **El Cliente:** Genéricamente, un programa que inicia la comunicación solicitando un servicio o dato (en nuestro caso, la Capa de Presentación).
- **El Servidor:** Genéricamente, un programa "siempre activo" que provee servicio (en nuestro caso, custodia los recursos, la Capa de Negocio + Persistencia).

Regla de Oro

El servidor nunca inicia la conversación; siempre reacciona a una petición del cliente.

Vuestra experiencia previa como “Clientes”

Ya habéis trabajado así en el contexto de las bases de datos:

Cliente	**Protocolo/Puerto**	**Servidor**
MySQL Workbench/phpMyAdmin	TCP / 3306	MySQL Server
MongoDB Compass/NoSQL Booster	TCP / 27017	MongoDB Instance

Vuestra experiencia previa como “Clientes”

Ya habéis trabajado así en el contexto de las bases de datos:

Cliente	Protocolo/Puerto	Servidor
MySQL Workbench/phpMyAdmin	TCP / 3306	MySQL Server
MongoDB Compass/NoSQL Booster	TCP / 27017	MongoDB Instance

¿Qué cambiará en el ámbito web?

- El *navegador*, con su frontend ya cargado, actuará como **cliente**.
- Un programa (en *Node.js*, PHP, Python, R, etc.) actuará como **servidor**.

Vuestra experiencia previa como "Clientes"

Ya habéis trabajado así en el contexto de las bases de datos:

Cliente	Protocolo/Puerto	Servidor
MySQL Workbench/phpMyAdmin	TCP / 3306	MySQL Server
MongoDB Compass/NoSQL Booster	TCP / 27017	MongoDB Instance

¿Qué cambiará en el ámbito web?

- El *navegador*, con su frontend ya cargado, actuará como **cliente**.
- Un programa (en *Node.js*, PHP, Python, R, etc.) actuará como **servidor**.

Contexto	Cliente	Protocolo/Puerto	Servidor
Bases de Datos	MySQL Workbench	TCP / 3306	MySQL Server
Bases de Datos	MongoDB Compass	TCP / 27017	MongoDB Instance
Web Apps	**Navegador (JS)**	**HTTP / 80 - 443**	**Web API (Node.js)**

Vuestra experiencia previa como "Clientes"

Ya habéis trabajado así en el contexto de las bases de datos:

Cliente	Protocolo/Puerto	Servidor
MySQL Workbench/phpMyAdmin	TCP / 3306	MySQL Server
MongoDB Compass/NoSQL Booster	TCP / 27017	MongoDB Instance

¿Qué cambiará en el ámbito web?

- El *navegador*, con su frontend ya cargado, actuará como **cliente**.
- Un programa (en *Node.js*, PHP, Python, R, etc.) actuará como **servidor**.

Contexto	Cliente	Protocolo/Puerto	Servidor
Bases de Datos	MySQL Workbench	TCP / 3306	MySQL Server
Bases de Datos	MongoDB Compass	TCP / 27017	MongoDB Instance
Web Apps	**Navegador (JS)**	**HTTP / 80 - 443**	**Web API (Node.js)**

Conclusión: El navegador es simplemente otro tipo de cliente que habla HTTP y recibe JSON. Es más: el propio servidor Node.js actuará como cliente sobre el servidor de BD.

Comunicación Frontend-Backend

Comunicación por paso de mensajes

En la web, la interacción nace de una **petición asíncrona**[1]:

- **Petición (Request):** El ***cliente*** (vuestro JS en el navegador) solicita mediante una URL devolver un recurso o realizar una acción al servidor.
- **Respuesta (Response):** El ***servidor*** procesa la lógica, accede en su caso a bases de datos o ficheros, y devuelve un resultado (normalmente en formato **JSON**).

[1] El navegador no se congela; sigue funcionando mientras los datos viajan por la red.

Separación de Responsabilidades

¿Por qué no conectar el Navegador directamente a la Base de Datos?

Roles bien diferenciados

Cliente (Navegador)

- Ve lo que el servidor le envía.
- No puede leer archivos del servidor.
- Misión: **presentar** los datos, **enviarlos** al servidor para guardar.

Servidor (Node/PHP/Python/R)

- Tiene las "llaves" de la BD.
- Realiza procesamiento.
- Misión: **servir** datos.

Reto

Ver **cómo se realiza la petición del cliente al servidor** mediante la **API Fetch**, lo que requiere entender bien antes la ***asincronía***.

Índice

1 Introducción

2 Asincronía

3 Comunicación con servidor

4 Servidor

El concepto de Asincronía

En un entorno síncrono, cada instrucción espera a que la anterior finalice. En la web, esto es inviable:

- Si una petición de datos tarda 2 segundos, no podemos "congelar" la interfaz del usuario.
- **Asincronía:** Capacidad de iniciar una tarea y permitir que el programa continúe con otras instrucciones hasta que el resultado esté listo.

Analogía del Restaurante

- **Síncrono:** El camarero espera en la cocina a que se cocine tu plato antes de atender a otra mesa.
- **Asíncrono:** El camarero deja la comanda, atiende a otros clientes y vuelve cuando el plato está listo.

Manejo de la Asincronía

¿Cómo tratamos la asincronía?

Vamos a ver dos tipos de soluciones tradicionalmente empleadas en `JavaScript` para manejar la asincronía:

- Funciones *callback*.
- Promesas.

Para entender algunos de estos conceptos, debemos repasar conceptos como las *funciones de orden superior*, que conectaremos con el concepto de *callback*.

Orden Superior y Callbacks

En JavaScript, las funciones son objetos. Podemos pasar una función como argumento a otra (*orden superior*). Es lo que hacíamos, por ejemplo, en `map`, `filter` o `forEach`, pasábamos una función, que era llamada por estas funciones. Un *callback* es una función que se pasa a otra para ser ejecutada más tarde.

```
// Función de Orden Superior (la que recibe un callback)
function operar(a, b, callback) {
  return callback(a, b);
}

// Llamada a la función pasándole otra función (anónima)
const suma = operar(5, 3, (x, y) => x + y);
const mult = operar(5, 3, (x, y) => x * y);

console.log(suma); // 8
```

Primera solución para la Asincronía: Callbacks

Para gestionar la asincronía, podemos usar **callbacks**: funciones pasadas como argumento para ser ejecutadas más tarde, sin bloquear el flujo de ejecución (ej: la última instrucción termina antes de `operacion(5,f)`).

```
function operacion(dato, callback) {
  console.log("Procesando...");
  // Simulamos un retraso antes de ejecutar el callback
  setTimeout(() => {
    callback(dato);
  }, 2000)
}
operacion(5, (res) => console.log("Resultado:", res));
console.log("Ya se ha lanzado la operación")
```

Problema:

El encadenamiento de múltiples procesos genera el *"Callback Hell"*.

¿Por qué evitar los Callbacks? (*Callback Hell*)

Cuando un proceso asíncrono depende del resultado de otro, el código comienza a crecer horizontalmente (la “Pirámide de la Perdición”):

Código difícil de mantener y depurar

```
obtenerUsuario(id, (usuario) => {
  obtenerPedidos(usuario.id, (pedidos) => {
    obtenerDetalle(pedidos[0].id, (detalle) => {
      enviarCorreo(detalle.email, (resultado) => {
        console.log("Proceso finalizado");
        // ¿Qué pasa si aquí falla algo?
      });
    });
  });
});
```

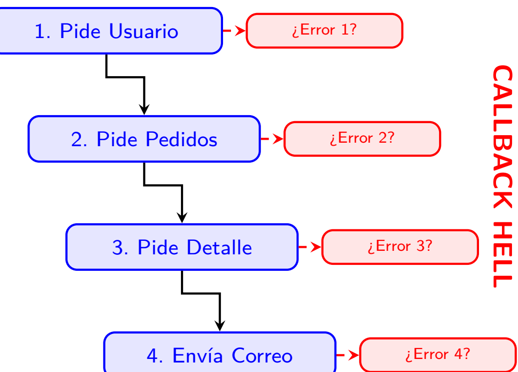
- **Legibilidad:** El flujo lógico se pierde en la indentación.
- **Gestión de Errores:** Habría que manejar el error en cada nivel.

El problema de la anidación (Callback Hell)

Cuando encadenamos múltiples operaciones asíncronas dependientes, el código crece horizontalmente y el flujo de control se vuelve caótico.

Pirámide de la Perdición

```
obtenerUsuario(id, (usr) => {
  // Procesar usuario
  obtenerPedidos(usr.id, (peds) => {
    // Procesar pedidos
    obtenerDetalle(peds[0].id, (det) => {
      // Procesar detalle
      enviarCorreo(det.email, (res) => {
        // ¿Y si falla algo aqui?
        console.log("Finalizado");
      });
    });
  });
});
```



La solución moderna: Promesas

Una **Promesa** es un objeto que representa el resultado de una operación asíncrona. Puede encontrarse en los siguientes **estados**:

- **Pending (Pendiente):** Estado inicial, la operación sigue en curso.
- **Fulfilled (Resuelta):** La operación terminó con éxito.
- **Rejected (Rechazada):** La operación falló por algún motivo.

Concepto clave

La promesa nos permite separar la **petición** de los datos del **procesamiento** de los mismos.

La solución moderna: Promesas

¿Cómo se define una promesa?

```
const miPromesa = new Promise((resolve, reject) => {
  setTimeout(() => {
    if (true) resolve(42); // El "valor" futuro
    else reject("Error");
  }, 2000);
});
```

Cuestión: ¿cómo tratamos la promesa?¿cómo accedemos a ese 42?
Para acceder al resultado devuelto, concluida la promesa, encadenamos funciones `then` (tratar el resultado al alcanzar estado *fulfilled*) o `catch` (si queremos tratar el error en caso de estado *rejected*).

```
miPromesa
  .then(valor => console.log("Resultado:", valor))
  .catch(err => console.error(err)
```

Consumo de Promesas: .then() y .catch()

Habiendo visto los *callbacks*, podemos entender el tratamiento de la promesa. Vemos que `then`[2] es una función de orden superior que espera un callback[3] para ejecutarlo una vez se cumpla la promesa:

```
miPromesa
  .then(datos => {
    console.log("Éxito:", datos);
  })
  .catch(error => {
    console.error("Error:", error);
  });
```

- .then(f) → Ejecuta *f* si hay éxito (*resolve*)
- .catch(g) → Ejecuta *g* si hay error (*reject*)

- Si ha terminado bien alcanzará el estado *resuelta*, y habrá devuelto un resultado, que será el que tome la función que le indiquemos (en este caso, muestra "Éxito", junto al resultado recibido).
- En caso contrario (estado *rechazada*), se captura el error que haya devuelto, y también se imprime por consola.

[2] Análogamente con `catch`
[3] En el ejemplo, una función anónima

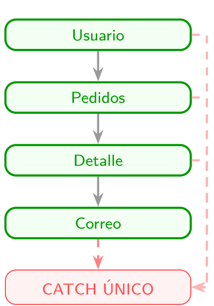
Alternativa a Pirámide de la Perdición:

Encadenamiento (Chaining)

Como `.then()` **devuelve** siempre **una nueva Promesa**, podemos ***encadenar procesos*** de forma lineal en lugar de anidarlos.

Flujo Lineal (Promesas)

```
obtenerUsuario(id)
  .then(usr => obtenerPedidos(usr.id))
  .then(peds => obtenerDetalle(peds[0].id))
  .then(det => enviarCorreo(det.email))
  .then(() => console.log("Finalizado"))
  .catch(err => console.error("Error único:", err));
console.log("Procesamiento fuera de then-catch")
```



- **Código plano:** Se lee de arriba a abajo.
- **Un solo catch:** Cualquier error en la cadena cae aquí.
- El mensaje tras el bloque se ejecutará antes de terminar la cadena.

Sintaxis moderna: Async / Await

Es “azúcar sintáctico” sobre Promesas. No cambia funcionamiento, pero permite escribir código asíncrono y leerlo como secuencial.

- **`async`**: Palabra clave que indica que una función siempre devolverá una promesa.
- **`await`**: Solo se usa dentro de funciones `async`. Pausa la ejecución de *esa* función hasta que la promesa se resuelva.

Diferencia clave con `.then()`

En un bloque `.then()`, el código que pongas fuera, justo después de la cadena de then-catch, se ejecuta **antes** de recibir la respuesta. Con `await`, el código que hay debajo **espera** obligatoriamente.

Evolución final del ejemplo con async/await

```
async function procesoEnvio(id) {
  try {
    const usr = await obtenerUsuario(id);
    const peds = await obtenerPedidos(usr.id);
    const det = await obtenerDetalle(peds[0].id);
    await enviarCorreo(det.email);
    console.log("Finalizado correctamente");
  } catch (err) {
    console.error("Error en cualquier paso:", err);
  }
}
```

- El mensaje tras los `await` no se ejecuta hasta el final.
- **Legibilidad:** Es casi idéntico al código síncrono tradicional.
- **Control:** El bloque `try/catch` sustituye al `.catch()` anterior.

Índice

1 Introducción

2 Asincronía

3 Comunicación con servidor

4 Servidor

El nexo de unión entre *frontend* y *backend*: API Fetch

La **API Fetch** es la interfaz moderna de JS para realizar peticiones HTTP. Con ella nuestro **Cliente** habla con el **Servidor**.

- **¿Qué devuelve fetch?**: Siempre devuelve una **Promesa**.
- **¿Es asíncrona?**: Sí, por naturaleza. La red es incierta y el tiempo de respuesta varía.
- **Estandarización**: Sustituye al antiguo objeto `XMLHttpRequest`, con una sintaxis mucho más limpia, basada en lo que hemos aprendido.

La firma de la función

$$\text{fetch(URL)} \xrightarrow{\text{devuelve}} \text{Promise}\langle\text{Response}\rangle$$

Dato importante

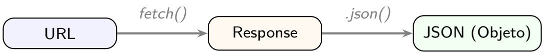
La respuesta de `fetch` es un flujo de datos. Para trabajar con ella como un objeto JS, debemos transformarla (normalmente a **JSON**), lo cual es, a su vez, **otro proceso asíncrono**.

Estructura típica de Fetch

Para consumir una API necesitamos dos pasos:

1. Realizar la petición (`fetch`)
2. Recuperar la respuesta como JSON o texto

Ambos procesos son asíncronos.



Opción A: con `.then()`

```
fetch('https://api.ejemplo.com/datos') // 1. Petición
  .then(respuesta => {
    // 2. Verificar conexión, recuperar resultado
    return respuesta.json();
  })
  .then(datos => {
    // 3. Procesar resultado
    console.log(datos);
  })
  .catch(error => {
    console.error("Fallo:", error);
  });
```

Opción B: con `async/await`

```
async function pedirDatos() {
  try {
    // Paso 1: Esperar respuesta
    const res = await fetch('url');
    // Paso 2: Esperar al JSON
    const datos = await res.json();
    console.log(datos);
  } catch (error) {
    console.error("Fallo:", error);
  }
}
```

Ejemplo Real: Consumiendo la PokéAPI

Vamos a obtener los datos de **Pikachu** y mostrarlos en consola y salida.

```
async function obtenerPokemon(nombre) {
  try {
    const respuesta = await fetch(`https://pokeapi.co/api/v2/pokemon/${nombre}`);

    // Se comprueba si la respuesta del servidor llegó bien (status 200-299)
    if (!respuesta.ok) throw new Error("Pokémon no encontrado");
    const datos = await respuesta.json();

    console.log(`Nombre: ${datos.name}`);
    console.log(`Imagen: ${datos.sprites.front_default}`);

    document.body.innerHTML = `
      <h1>${datos.name.toUpperCase()}</h1>
      <img src="${datos.sprites.front_default}" alt="${datos.name}">
    `;
  } catch (error) {
    console.error("Error en la petición:", error);
  }
}

obtenerPokemon('pikachu');
```

Índice

1 Introducción
2 Asincronía
3 Comunicación con servidor
4 Servidor

¿Qué ocurre cuando haces un fetch?

Hasta ahora hemos sido el **Cliente**. Pero para que el mundo funcione, necesitamos un **Servidor**[4].

- **El Cliente (Front-end):** El navegador (Chrome, Safari). Pide recursos y acciones al servidor.
- **El Servidor (Back-end):** Una computadora que "escucha" peticiones 24/7 y decide qué responder.

La Analogía del Restaurante

- **Cliente:** Comensal que lee la carta (HTML) y pide comida (fetch).
- **Servidor:** La cocina que procesa el pedido y devuelve el plato (JSON/Datos).

[4] El servidor puede ser propio o de terceras partes, y puede escribirse en lenguaje PHP, Python, R, Java, C#, JavaScript, etc.

Las 3 misiones críticas del Servidor

¿Por qué no guardar todo en el navegador del usuario?

1. **Seguridad y Control:** El cliente es público (cualquiera puede ver el código JS). El servidor es privado; ahí guardamos contraseñas y lógica de negocio secreta.
2. **Persistencia Real:** El servidor conecta con la **Base de Dato**. Si cierras el navegador, los datos siguen vivos en el servidor.
3. **Procesamiento Pesado:** Operaciones matemáticas complejas o manejo de archivos grandes se delegan al servidor para no colapsar el móvil/PC del usuario.

"El servidor es la fuente de la verdad; el cliente es solo una ventana para ver esa verdad."

¿Cómo pasaremos de ser cliente a servidor?

JavaScript sale del navegador

Antiguamente, JavaScript solo vivía dentro del navegador. En 2009, todo cambió con **Node.js**.

- **¿Qué es?** Un entorno que permite ejecutar JavaScript directamente en el sistema operativo.
- **¿Para qué sirve?** Para crear servidores ultrarrápidos, APIs, y herramientas de automatización.
- **La gran ventaja:** ¡Usamos el mismo lenguaje en ambos lados! (*Full-Stack JavaScript*).

Próximamente...

Aprenderemos a configurar nuestro servidor, crear APIs con las que conectar con nuestra BD y acercar nuestra lógica al mundo real.

Agradecimientos: `Gemini`, que asistió en la creación de esta presentación.