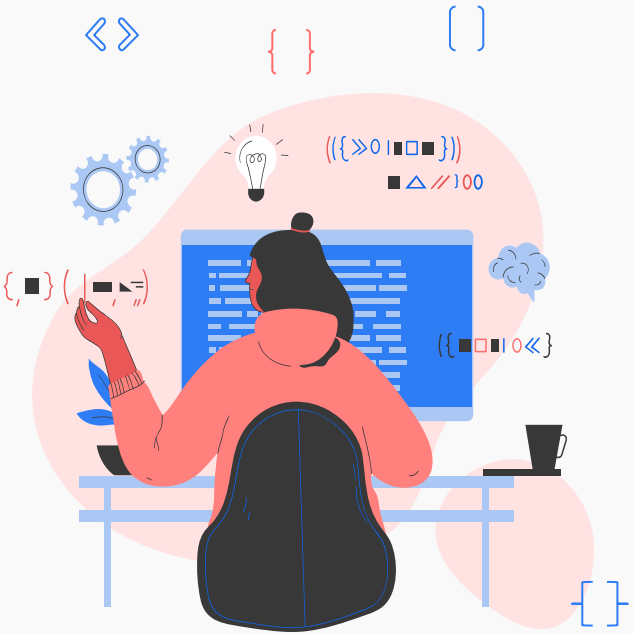
Desarrollo Web en Entorno Cliente

Tema 5 – Mecanismos de comunicación asíncrona



Marina Hurtado Rosales
marina.hurtado@escuelaartegranada.com

Indice de contenidos

Aplicaciones web y organización de código

Aplicaciones web modernas

Las aplicaciones web modernas han evolucionado desde páginas estáticas hacia sistemas interactivos que permiten al usuario trabajar con datos en tiempo real.

Una aplicación web moderna:

- Muestra información dinámica
- Permite modificar datos
- Mantiene su estado
- Ofrece una experiencia de usuario fluida

Aplicaciones web modernas

A medida que una aplicación web crece:

- Aumenta la cantidad de código
- Se gestionan más datos
- Aparecen más interacciones

Sin una buena organización:

- El código se vuelve más difícil de mantener
- Aparecen errores con más frecuencia

Patrones de desarrollo web

Para organizar aplicaciones complejas se utilizan patrones de desarrollo.

Un patrón de desarrollo:

- Es una solución general a un problema común
- No depende de un lenguaje concreto
- Sirve como guía para estructurar aplicaciones

Un **patrón de diseño** es una solución habitual a un problema común en el diseño de software. Cada patrón es como un plano que se puede personalizar para resolver un problema de diseño particular de tu código

¿En qué consiste un patrón de diseño?

La mayoría de los patrones se describen con mucha formalidad para que la gente pueda reproducirlos en muchos contextos diferentes. Las secciones que suelen usarse para describirlos son:

- **Propósito**: explica brevemente el problema y la solución
- **Motivación**: explica en más detalle el problema y la solución que brinda el patrón.
- **Estructura**: muestra cada una de las partes del patrón y el modo en que se relacionan
- **Ejemplo de código**: se muestra un ejemplo en uno de los lenguajes de programación populares.

Principales patrones de desarrollo web

En el desarrollo web se utilizan distintos patrones y enfoques, entre ellos:

- **MVC** (Model–View–Controller)
- **MVP** (Model–View–Presenter)
- **MVVM** (Model–View–ViewModel)
- **Arquitectura basada en componentes**
- **Arquitectura reactiva** (patrón observador)

Cada uno responde a necesidades distintas.

MVC – Model View Controller

Se trata de un patrón de diseño de software comúnmente utilizado para implementar interfaces de usuario, datos y lógica de control.

Enfatiza la separación entre la lógica de negocios y su visualización. Busca separar responsabilidades y evitar mezclar conceptos.

Este patrón divide el código en tres partes principales:

- **Modelo**: Maneja datos y la lógica de negocio. Contiene toda la lógica de datos.
- **Vista**: se encarga del diseño y presentación.
- **Controlador**: enruta comandos a los modelos y vistas. Es el “cerebro” de la aplicación, que controla cómo se muestran los datos y comunica la vista con el modelo.

MVC – Model

El **Modelo** es la capa de comunicación de la base de datos y las capas de redes con el resto de la aplicación. Sus principales responsabilidades son:

- El manejo de los datos de la aplicación
- La gestión de la lógica asociada a esos datos
- La encapsulación de los datos de la aplicación y gestión de las reglas de acceso a estos.
- Realizar operaciones CRUD (Create, Read, Update and Delete) en los datos.

En el **entorno cliente** puede incluir

- Objetos JavaScript
- Datos en formato JSON
- Información almacenada en *localStorage*.

El trabajo del modelo es simplemente administrar los datos. Ya sea que los datos provengan de una base de datos, una API o un objeto JSON, el modelo es responsable de administrarlos.

MVC – View

La **Vista** es la parte visible de la aplicación. Se encarga del diseño y de la presentación.

Se trata básicamente del **frontend** de la aplicación, que el usuario puede ver e interactuar. Se le conoce también como **Interfaz de Usuario (UI).**

Sus responsabilidades incluyen:

- Manejar la lógica que no es de negocio, la que sólo se relaciona con la presentación.
- Mostrar los datos proporcionados por las otras capas al usuario
- Representar la información
- Permitir la interacción, recibiendo las entradas del usuario y derivándolas a otras capas (captura los eventos de usuario).

Está formada por el código HTML y CSS y por el DOM.

El trabajo de la Vista es decidir qué verá el usuario en su pantalla y cómo.

MVC – Controller

El **Controlador** actúa como intermediario entre **Modelo** y **Vista**. Su responsabilidad es extraer, modificar y proporcionar datos al usuario.

A través de las funciones getter y setter, el controlador extrae datos del modelo e inicializa las vistas.

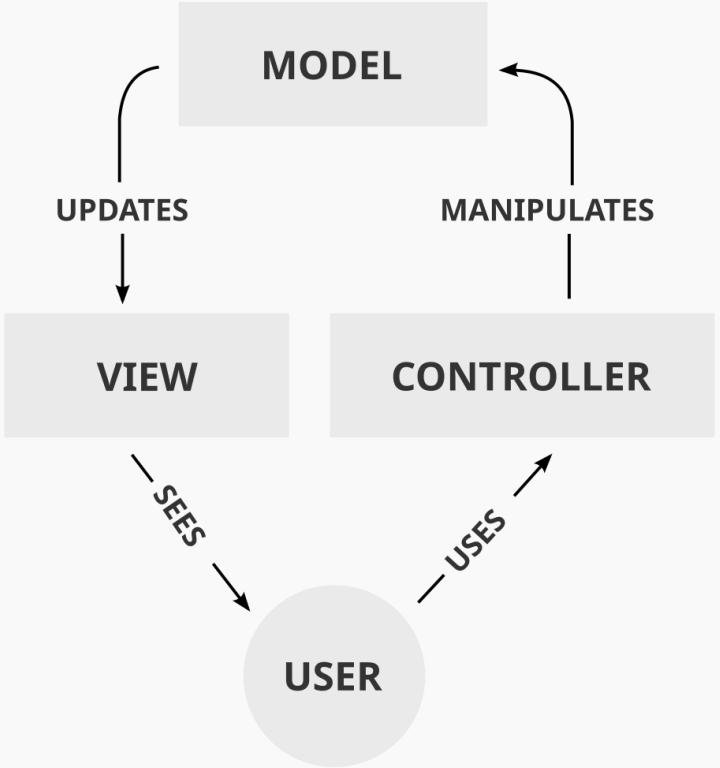
Sus principales responsabilidades son:

- Capturar eventos del usuario
- Decidir qué acciones realizar
- Coordinar datos y vista
- Manipular los datos a través de la capa del Modelo
- Actualizar la Vista con los cambios definidos en la lógica de control

El **Controlador** contiene una lógica que actualiza el Modelo y/o Vista en respuesta a las entradas de los usuarios de la aplicación.

MVC – Flujo de funcionamiento

- El usuario interactúa con la **Vista**
- El **Controlador** captura el evento
- El **Controlador** consulta o modifica el **Modelo**
- La **Vista** se actualiza con los nuevos datos



Otros patrones de desarrollo web

Además de MVC, en el desarrollo web existen otros patrones y enfoques que permiten organizar aplicaciones según sus necesidades.

Estos patrones:

- No sustituyen a MVC
- Resuelven problemas distintos
- Suelen aparecer en frameworks modernos

Es importante conocerlos, aunque no se implementen directamente en este módulo

MVC – Model View Presenter

El patrón **MVP** es una evolución de MVC. Se utiliza principalmente para construir interfaces de usuario. En este patrón, el Presentador adopta la funcionalidad de “middle man”.

Se caracteriza porque:

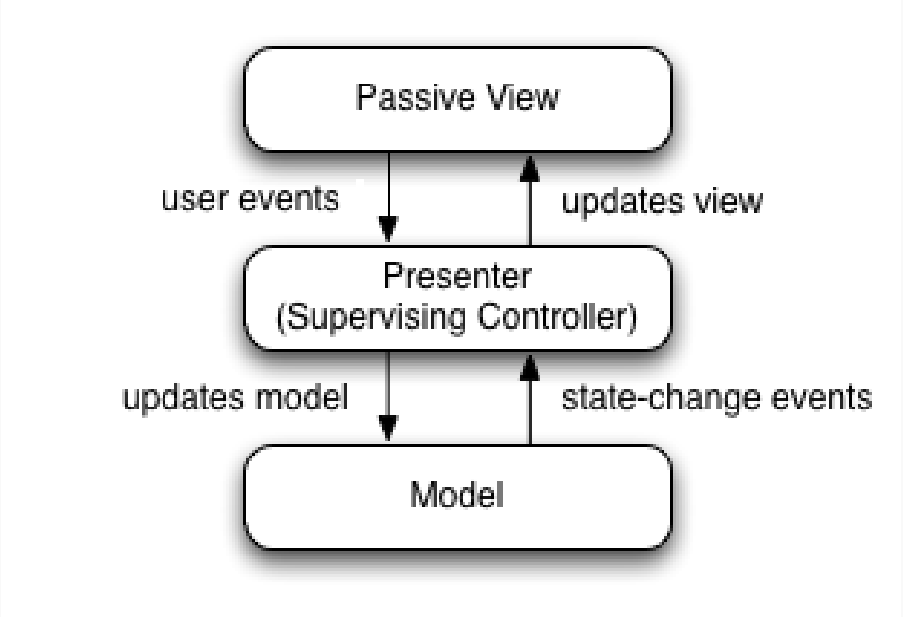
- La Vista es más pasiva
- El Presenter contiene gran parte de la lógica
- Se reduce la dependencia directa entre Vista y Modelo

Se utiliza cuando se necesita un mayor control de la lógica de presentación.
En este modelo las partes son:

- **Modelo**: al igual que en MVC, el modelo se encarga del manejo de los datos de la aplicación.
- **Vista**: se encarga **únicamente** de mostrar lo que le dicen. **NO** captura eventos, sino que los lanza.
- **Presentador**: contiene **toda la lógica**. Decide qué mostrar y cómo. No deja a la vista tomar decisiones.

MVP – Flujo de funcionamiento

- El usuario interactúa con la **Vista**
- La **Vista** no decide nada
- La **Vista** avisa al **Presentador**
- El **Presentador**:
  - Decide qué hacer
  - Consulta o modifica el **Modelo**
  - Prepara los datos
- El **Presentador** ordena a la **Vista** qué mostrar



Comparación MVC y MVP

Aspecto	MVC	MVP
Papel de la vista	Activo	Pasivo
Lógica de presentación	Repartida	Centralizada
Comunicación Vista-Modelo	Puede existir	No existe
Intermediario	Controlador	Presentador
Complejidad	Media	Mayor
Uso típico	Apps pequeñas/medias	Interfaces complejas

MVVM – Model View ViewModel

El patrón **MVVM** es muy utilizado en frameworks modernos.

Se caracteriza porque:

- El **ViewModel** actúa como intermediario
- La vista se actualiza automáticamente cuando cambian los datos
- Se reduce la manipulación directa del DOM

Este patrón divide la aplicación en tres partes, cada uno con una responsabilidad clara y definida:

- **Modelo**: gestiona los datos de la aplicación, contiene la lógica de negocio y valida y procesa la información. **NO** conoce la vista ni el VIewModel
- **Vista**: muestra la interfaz al usuario, define la estructura visual, contiene muy poca lógica. No toma decisiones, no gestiona datos y se limita únicamente a mostrar información.
- **ViewModel**: prepara los datos para la vista, mantiene el estado de la interfaz, reacciona antes las acciones del usuario y se comunica con el modelo. **NO** conoce la Vista directamente.

MVVM – Model View ViewModel

Una característica fundamental del MVVM es el **enlace de datos** (data binding).

Gracias a este mecanismo:

- Cuando cambian los datos, la vista se actualiza automáticamente.
- No es necesario manipular el DOM manualmente.
- Se reduce el código repetitivo.

Flujo de funcionamiento en MVVM

- El usuario interactúa con la **Vista**
- La **Vista** notifica el cambio al **ViewModel**
- El **ViewModel** actualiza el modelo si es necesario
- El **ViewModel** actualiza su estado
- La **Vista** se actualiza automáticamente mediante data binding

Diferencias MVVM vs MVC

En comparación con MVC:

- En **MVC**, la vista participa activamente.
- En **MVVM**, la vista es más declarativa.
- **MVC** requiere actualizar la vista manualmente.
- **MVVM** actualiza la vista automáticamente mediante data binding.
- **MVVM** reduce la manipulación directa del DOM.

Diferencias MVVM vs MVP

En comparación con MVP:

- En **MVP**, el Presentador controla la vista.
- En **MVVM**, la vista se actualiza sola.
- **MVP** requiere llamadas explícitas para actualizar la vista.
- **MVVM** se basa en enlaces de datos automáticos.
- **MVVM** reduce la cantidad de código de presentación.

Comparativa global

- **MVC**: sencillo, vista activa, control manual. La vista participa activamente.
- **MVP**: vista pasiva, lógica centralizada. La vista es controlada por el presentador.
- **MVVM**: vista declarativa, actualización automática.

Cada patrón responde a necesidades distintas.

Arquitectura basada en componentes

Una arquitectura basada en componentes divide la aplicación en **pequeñas piezas independientes,** llamadas **componentes.**
Cada componente puede desarrollarse y mantenerse de forma independiente. De esta manera, cada componente

- Representa una parte independiente y concreta de la interfaz
- Tiene una responsabilidad clara y definida, no intenta hacer demasiadas cosas
- Puede reutilizarse en distintos lugares

La aplicación web se construye combinando componentes entre sí. Esto facilita:

- La reutilización de código
- La claridad estructural
- El mantenimiento
- La escalabilidad de la aplicación
- El trabajo en equipo

Arquitectura basada en componentes

Un **componente** suele:

- Mostrar una parte de la interfaz
- Gestionar su propio estado
- Responder a eventos del usuario
- Comunicarse con otros componentes

Los componentes permiten que el código esté:

- Mejor organizado
- Más aislado
- Mas fácil de mantener

Esta arquitectura no elimina los patrones clásicos, sino que los reorganiza. Por ejemplo:

- Un componente puede actuar como vista
- Otro puede encargarse de la lógica
- Otro de la gestión del estado.

Arquitectura reactiva y patrón observador

En una arquitectura reactiva, la aplicación **reacciona automáticamente** a los cambios en los datos.

En lugar de:
- Preguntar constantemente por cambios
- Actualizar manualmente la interfaz

La aplicación se basa en eventos y notificaciones.

Esta arquitectura se basa en el **patrón observador**. Este patrón define:
- Un **sujeto** que mantiene un estado
- Uno o varios **observadores** que están atentos a este estado

Cuando el **estado** cambia, los **observadores** son notificados automáticamente.

Funcionamiento del patrón observador

El **funcionamiento básico** es:

- Un **objeto** mantiene un **estado**.
- Otros objetos se **suscriben** a ese **estado**.
- Cuando el estado cambia, se notifica a los **observadores**.
- Cada observador **reacciona** al cambio.

Este patrón permite desacoplar las partes de la aplicación. En desarrollo web, este patrón se utiliza para:

- Actualizar interfaces automáticamente
- Reaccionar a cambios en datos
- Sincronizar distintas partes de la aplicación

En esta arquitectura:

- El **estado** es una pieza central
- Los cambios en el estado generan reacciones
- La interfaz refleja siempre el estado actual

De esta manera se reduce el código repetitivo, los errores de sincronización y la dependencia entre componentes,

Trabajo con datos en aplicaciones web

Todas las aplicaciones web modernas trabajan con datos, los cuales pueden mostrarse al usuario, modificarse, guardarse o eliminarse.

Independientemente de la arquitectura utilizada, el tratamiento de datos es fundamental.

Operaciones CRUD

El trabajo con datos se resume en cuatro operaciones básicas, conocidas como **CRUD**

- **Create**: crear datos
- **Read**: leer datos
- **Update**: modificar datos
- **Delete**: eliminar datos

Estas operaciones aparecen constantemente en aplicaciones reales.

CRUD y patrones de desarrollo

En los patrones vistos (MVC, MVP, MVVM) las operaciones CRUD forman parte de la lógica de la aplicación.

- El **Modelo** es el encargado de gestionar los datos
- Las operaciones CRUD se realizan sobre el Modelo
- La **Vista** nunca debe gestionar directamente los datos

¿De dónde vienen los datos?

En aplicaciones sencillas los datos pueden estar:

- En **memoria**
- En **archivos locales**
- En el **almacenamiento del navegador**

En aplicaciones reales, los datos suelen estar:

- En un **servidor**
- Accesibles a través de **APIs**.

Para trabajar con estos datos, es necesaria la comunicación **cliente-servidor**.

CRUD y comunicación cliente-servidor

Las operaciones CRUD suelen implicar:

- Enviar **peticiones** al **servidor**
- Recibir **respuestas con datos**
- **Actualizar el estado** de la aplicación

A continuación estudiaremos cómo se organiza esta comunicación y cómo se realizan estas operaciones de forma asíncrona

Arquitectura cliente-servidor

¿Qué es el modelo cliente-servidor?

Las aplicaciones web modernas suelen basarse en la arquitectura **cliente-servidor,** un estilo arquitectónico **distribuido**.

Se trata de uno de los modelos más utilizados en sistemas distribuidos y en aplicaciones web.

Este modelo divide a la aplicación en **dos partes** principales, cada una con responsabilidades distintas y bien definidas:

- **Cliente**: el cual solicita información o envía datos
- **Servidor**: que recibe la petición, la procesa y devuelve una respuesta

Cliente y servidor pueden ejecutarse en máquinas diferentes, y se comunican a través de la red.

¿Qué es el modelo cliente-servidor?

En el modelo cliente-servidor:

- El **Cliente** inicia la comunicación
- El **Servidor** recibe peticiones, las procesa y devuelve respuestas

De esta manera, la capacidad de procesamiento se reparte entre ambas partes.

Sin embargo, una de sus mayores ventajas es la **centralización** de la **gestión de la información**.

De esta manera, los **datos y la lógica** quedan centralizados en el **servidor**, y la **interfaz** y la interacción con el usuario se realiza en los **clientes**.

El cliente en una aplicación web

El **Cliente** en una aplicación web:

- Se ejecuta en el navegador del usuario
- Gestiona la interfaz de usuario
- Responde a eventos tanto del usuario como del propio navegador
- Solicita datos al servidor cuando es necesario

El cliente NO gestiona directamente los datos persistentes ni accede a bases de datos.

En este módulo, el cliente se desarrolla principalmente con **HTML, CSS y JavaScript**.

El servidor en una aplicación web

El **Servidor** en una aplicación web:

- Recibe peticiones del cliente
- Procesa y gestiona los datos
- Accede a las bases de datos
- Aplica las reglas de la lógica de negocio
- Devuelve respuestas al cliente

El servidor centraliza la **gestión de la información** y puede atender a múltiples clientes simultáneamente.

El servidor NO gestiona la interfaz de usuario.

Organización de la comunicación cliente-servidor

La comunicación entre cliente y servidor se realiza mediante:

- El envío de **peticiones**
- La recepción de **respuestas**

Esta comunicación debe estar bien organizada para evitar errores, dependencias innecesarias y problemas de mantenimiento.

En esta arquitectura, el **Cliente** es quien inicia siempre la comunicación, mientras que el **Servidor** espera peticiones y responde a ellas. El servidor no envía información al cliente si este no la solicita previamente.

Esta comunicación se realiza mediante protocolos de red.

Protocolos de red

La comunicación entre cliente y servidor no se realiza de forma arbitraria, sino siguiendo protocolos de comunicación.

Un protocolo define:

- cómo se envían los datos
- en qué formato
- quién inicia la comunicación,
- cómo se responde a una petición.

Los protocolos garantizan que cliente y servidor puedan entenderse.

Algunos ejemplos de protocolos son: HTTP, HTTPS, FTP, SMTP, POP3, IMAP, etc...

TCP/IP: la base de la comunicación en red

TCP/IP es el conjunto de protocolos **fundamental** que permite la comunicación entre dispositivos a través de una red.

Proporciona:

- **IP**: direccionamiento de los dispositivos
- **TCP**: envío fiable y ordenado de datos.

Todos los protocolos de nivel aplicación se apoyan sobre TCP/IP

Protocolos de nivel aplicación

Sobre TCP/IP se construyen protocolos de nivel aplicación, cada uno diseñado para un tipo de comunicación concreto

- **HTTP / HTTPS** → comunicación web
- **FTP** → transferencia de archivos
- **SMTP, POP3, IMAP** → correo electrónico

Cada protocolo tiene un **propósito específico**.

HTTP (HyperText Transfer Protocol)

HTTP es el protocolo utilizado para la comunicación entre:

- Navegadores web
- Servidores web.

Características principales:

- Se basa en un modelo de petición-respuesta
- Es sin estado (**stateless**). Eso quiere decir que cada petición HTTP es **independiente**.
- Permite intercambiar recursos y datos (HTML, JSON, etc.)

Es el protocolo principal de las aplicaciones web.

HTTPS: comunicación segura

HTTPS es la versión segura de HTTP.

Añade:

- Cifrado de la información
- Protección frente a interceptaciones
- Mayor seguridad en la transmisión de datos

Actualmente, HTTPS es el estándar en aplicaciones web modernas.

FTP (File Transfer Protocol)

FTP es un protocolo diseñado para la transferencia de archivos entre cliente y servidor.

Características:

- No está pensado para aplicaciones web dinámicas
- No se utiliza para APIs ni para intercambio de datos de aplicaciones web.

Su finalidad es distinta a la de HTTP, busca únicamente **subir y bajar archivos**, nada más.

Se utiliza para subir y bajar archivos a un servidor, descargar archivos de un servidor, gestionar ficheros remotos...

Ejemplos: transferir copias de seguridad, mover archivos grandes entre sistemas...

Otros protocolos de comunicación

Existen otros protocolos de aplicación, como:

- **SMTP**: envío de correos electrónicos
- **POP3 / IMAP**: recepción de correos.

Estos protocolos no se utilizan para la comunicación típica de aplicaciones web cliente-servidor.

Cliente-servidor y limitaciones de la comunicación

La comunicación cliente-servidor:

- Depende de la red
- Puede tardar, no es inmediata

Si no se gestiona correctamente, puede bloquear la interfaz de usuario.

Para evitar que la aplicación se bloquee mientras espera respuestas del servidor

- Las peticiones deben realizarse de forma **asíncrona**
- La interfaz debe seguir siendo **usable**

Esto introduce la necesidad del concepto de **asíncronía en JavaScript**.

Asincronía en JavaScript

Asincronía en JavaScript

JavaScript es un lenguaje:

- **Monohilo** (single-thread)
- Basado en **eventos**.

Esto significa que:

- Solo puede ejecutar **una tarea a la vez**
- Si una tarea tarda, **bloquea** el resto.

Para evitar bloquear la aplicación, JavaScript utiliza **asincronía**.

Ejecución síncrona

En la **ejecución síncrona**

- Cada instrucción **espera** a que termina la **anterior**
- El flujo del programa es **secuencial**

Si una operación tarda:

- El programa se queda bloqueado
- La interfaz deja de responder

```
console.log("Inicio");
alert("Operación lenta");
console.log("Fin");
```

En este código, mientras alert está activo el programa se detiene y el usuario no puede interactuar.

Ejecución asíncrona

En la **ejecución asíncrona**

- Una **operación** se **lanza**
- El programa **continúa ejecutándose**
- El resultado se gestiona **cuando está dispobible**

Este modelo es fundamental para:

- Realizar peticiones al servidor
- Usar temporizadores
- Realizar operaciones lentas

Ejecución asíncrona

```
console.log("Inicio");

setTimeout(() => {
  console.log("Operación asíncrona");
}, 2000);

console.log("Fin");
```

La salida de la ejecución de este código sería:

```
Inicio
Fin
Operación asíncrona
```

Callbacks

En JavaScript algunas operaciones tardan en completarse, y no podemos detener el programa esperando el resultado.

Por esta razón, lanzamos la operación e indicamos mediante un callback qué hacer cuando termine.

Un **callback** es una función que:

- Se pasa como argumento a otra función
- No se ejecuta inmediatamente
- Se ejecuta cuando una operación termina

Se utiliza para indicar **qué debe hacerse** cuando una tarea **asíncrona** finaliza.

Callbacks

```
function obtenerDatos(callback) {
  setTimeout(() => {
    callback("Datos recibidos");
  }, 2000);
}

function transformarDato(datoAModificar) {
  console.log(datoAModificar + "\n Datos modificados");
}

obtenerDatos(transformarDato);
```

Callbacks

Ventajas de los callbacks

- Son fáciles de entender en ejemplos simples
- Representan claramente el flujo de ejecución
- Permiten separar responsabilidades

Limitaciones de los callbacks

Cuando el número de operaciones crece, los callbacks pueden provocar:

- Código muy anidado
- Dificultad para seguir el flujo
- Repetición de manejo de errores
- Dificultad de mantenimiento

Callbacks anidados

Imagina que queremos una función que obtenga un usuario, luego obtenga los pedidos de ese usuario, luego los detalles de los pedidos y por último los muestre por pantalla.

Con callbacks sería algo de este estilo, algo cada vez menos legible y mantenible y que ni siquiera está manejando posibles errores:

```
obtenerUsuario(function(usuario) {
  obtenerPedidos(usuario.id, function(pedidos) {
    obtenerDetalles(pedidos, function(detalles) {
      mostrarResultado(detalles);
    });
  });
});
```

Callback hell

```
asyncFunc('something', function (err, data) {
    asyncFunc('something', function (err, data) {
        asyncFunc('something', function (err, data) {
            asyncFunc('something', function (err, data) {
                asyncFunc('something', function (err, data) {
                    asyncFunc('something', function (err, data) {
                        asyncFunc('something', function (err, data) {
                            asyncFunc('something', function (err, data) {
                                asyncFunc('something', function (err, data) {
                                    // do the final action
                                });
                            });
                        });
                    });
                });
            });
        });
    });
});
```

Promesas

Las promesas surgen como solución a los problemas de los callbacks. Una promesa es una abstracción que permite representar el resultado de una operación asíncrona sin necesidad de gestionar manualmente los callbacks.

Una **promesa** representa una operación que se completará en el futuro o fallará. Básicamente se trata de un objeto que representa el resultado de una operación asíncrona. Ese resultado:

- Puede estar disponible ahora
- Puede estar disponible en el futuro
- Puede no llegar nunca por un error

Una promesa **NO contiene el resultado directamente**, sino la promesa de que ese resultado llegará.

Estados de una promesa

Una promesa puede encontrarse en uno de estos tres estados:

- **Pendiente (pending)**: la operación aún no ha terminado
- **Resuelta (fulfilled)**: la operación ha terminado correctamente
- **Rechazada (rejected)**: la operación ha fallado

Una vez que una promesa se resuelve o se rechaza, **su estado no cambia**.

Creación de una promesa

Una promesa se crea utilizando el constructor de la clase nativa de JavaScript **Promise.**

```
const promesa = new Promise((resolve, reject) => {
  // operación asíncrona
});
```

Cuando se crea una promesa:

- Comienza en estado **pendiente**
- JavaScript espera a que se invoque **resolve** o **reject** para cambiar este estado

resolve y reject

Al crear una promesa, JavaScript proporciona dos funciones:

- **resolve(valor)**: indica que la operación ha terminado correctamente
- **reject(error)**: indica que ha ocurrido un error

Estas funciones:

- Sólo pueden llamarse UNA VEZ
- Determinan el estado final de una promesa

```
const promesa = new Promise((resolve, reject) => {
  const todoVaBien = true;

  if (todoVaBien) {
    resolve("Operación correcta");
  } else {
    reject("Ha ocurrido un error");
  }
});
```

Ejemplo de una promesa

```
const promesa = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("Datos obtenidos correctamente");
  }, 2000);
});
```

En este ejemplo:

- La operación tarda 2 segundos
- Cuando termina, la promesa se resuelve
- El valor pasado a *resolve* será el resultado

Consumo de una promesa con then()

El método **then()** se utiliza para indicar qué hacer cuando una promesa se resuelve y para utilizar el resultado de esta promesa.

```
promesa.then((resultado) => {
  console.log(resultado);
});
```

El parámetro del callback de *then*:

- Recibe el valor pasado a **resolve**
- Se ejecuta sólo cuando la promesa se ha resuelto

Manejo de errores con catch()

Si la promesa falla, el error se gestiona con el método **catch()**

```
promesa
  .then(resultado => {
    console.log(resultado);
  })
  .catch(error => {
    console.error(error);
  });
```

Esto permite:

- Separar claramente el flujo correcto del flujo de error. Si la promesa es rechazada, se ejecuta el método catch, y si no el then.
- Evitar múltiples callbacks de error
- Evitar que el error rompa el programa

Encadenamiento de promesas

En muchas situaciones reales:

- Una operación asíncrona depende del resultado de otra
- No podemos ejecutar la segunda hasta que la primera termine

Por ejemplo, si queremos obtener datos, luego procesarlos y después mostrarlos.

Para estos casos se utiliza el **encadenamiento de promesas.**

Esto permite:

- Ejecutar operaciones asíncronas en orden
- Usar el resultado de una operación en la siguiente.

Qué devuelve un then()

La idea fundamental es que **cada llamada a then() devuelve una nueva promesa**.

Cuando utilizamos then()

- El código dentro de then() se ejecuta cuando la promesa se resuelve
- Lo que se devuelve dentro de then() se pasa al siguiente then()

Ese valor puede ser
- Un valor normal
- Otra promesa

Encadenamiento con valores normales

Si un then() devuelve un valor normal:

```
promesa
  .then(resultado => {
    return resultado.toUpperCase();
  })
  .then(resultadoModificado => {
    console.log(resultadoModificado);
  });
```

JavaScript

- Convierte automáticamente ese valor en una promesa resuelta
- Pasa el valor al siguiente then()

Encadenamiento con valores normales

Cuando encadenamos promesas con valores normales es como si estuviéramos creando una promesa y usando su resultado para crear otra nueva promesa.

En JavaScript estos dos códigos serían equivalentes

```
promesa
  .then(resultado => {
    return resultado.toUpperCase();
  })
  .then(resultadoModificado => {
    console.log(resultadoModificado);
  });
```

```
const p2 = promesa.then(resultado => {
  return resultado.toUpperCase();
});

p2.then(resultadoModificado => {
  console.log(resultadoModificado);
});
```

Devolver otra promesa en un then()

En muchos casos, la siguiente operación que se tiene que hacer es también asíncrona. Una solución válida y muy habitual es hacer que el then() también **devuelva otra promesa**.

```
function transformarDato(dato) {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve(dato + " procesado");
    }, 1000);
  });
}
```

Esta función devuelve una promesa, tarda un segundo y devuelve un nuevo valor cuando termina.

Encadenamiento devolviendo otra promesa

```
const promesa = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("Datos obtenidos correctamente");
  }, 2000);
});
```

```
function transformarDato(dato) {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve(dato + " procesado");
    }, 1000);
  });
}
```

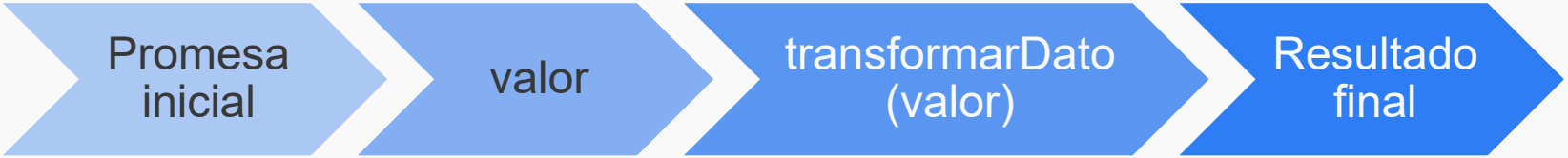
```
promesa
.then(valor => {
return transformarDato(valor);
})
.then(resultadoFinal => {
console.log(resultadoFinal);
});
```

Aquí ocurre lo siguiente:

- El primer then devuelve una promesa
- JavaScript espera automáticamente a que se resuelva
- El resultado final pasa automáticamente al siguiente then

Encadenamiento devolviendo otra promesa

Flujo visual del encadenamiento



Error común en el encadenamiento

```
promesa.then(valor => {
  transformarDato(valor);
});
```

Aquí la promesa no se devuelve, y como consecuencias:

- JavaScript no espera
- El flujo se rompe
- El encadenamiento no funciona correctamente

Manejo de errores en cadenas de promesas

Una de las grandes ventajas de las promesas es el manejo de errores.

En una cadena de promesas cualquier error producido en cualquier **then** o en cualquier promesa intermedia se propaga automáticamente hasta el primer **catch**

```
promesa
.then(valor => {
return transformarDato(valor);
})
.then(resultadoFinal => {
console.log(resultadoFinal);
})
.catch(error => mostrarError(error));
```

Con un **único catch**:

- Se capturan todos los errores de la cadena
- No es necesario repetir el manejo de errores
- El código es más limpio y coherente

Callbacks vs promesas

Callbacks

- Más intuitivos al principio
- Gestión manual del flujo
- Anidamiento al crecer

Promesas

- Flujo lineal
- Encadenamiento automático
- Mejor mantenimiento

```
obtenerUsuario(function(usuario) {
  obtenerPedidos(usuario.id, function(pedidos) {
    obtenerDetalles(pedidos, function(detalles) {
      mostrarResultado(detalles);
    }, mostrarError);
  }, mostrarError);
}, mostrarError);
```

```
obtenerUsuario()
.then(usuario => obtenerPedidos(usuario.id))
.then(pedidos => obtenerDetalles(pedidos))
.then(detalles => mostrarResultado(detalles))
.catch(error => mostrarError(error));
```

async y await

Aunque las **promesas** resuelven los problemas de los **callbacks**, el uso continuado de **then()** puede resultar poco legible cuando hay muchas operaciones encadenadas.

Para mejorar la legibilidad del código, JavaScript introduce las palabras clave **async** y **await**.

¿Qué es async?

La palabra clave **async** se utiliza para declarar una función asíncrona.

Una función **async**:

- Siempre devuelve una promesa
- Aunque aparentemente devuelva un valor normal

Esto significa que cualquier **valor devuelto** se convierte **automáticamente** en una **promesa resuelta**.

¿Qué es await?

La palabra clave **await** se utiliza dentro de funciones **async**.

Permite:

- Esperar el resultado de una promesa
- Sin bloquear la ejecución del programa

await detiene la ejecución de la función, pero **no bloquea** el hilo principal.

Ejemplo básico con async y await

```
async function obtenerDatos() {
  const valor = await promesa;
  console.log(valor);
}
```

Aunque el código parece secuencial, no lo es.

- El código sigue siendo asíncrono
- JavaScript gestiona la espera automáticamente

Es importante entender que **async/await** NO sustituye a las promesas, sólo es una forma más legible de trabajar con ellas. Internamente JavaScript sigue trabajando con promesas

Async/await y encadenamiento

El encadenamiento de promesas también puede expresarse con **await.**

El siguiente código es equivalente a varios **then()** encadenados.

```
async function procesoCompleto() {
  const v1 = await promesa;
  const v2 = await transformarDato(v1);
  console.log(v2);
};
```

Manejo de errores con try/catch

Con **async / await**, los errores se gestionan utilizando bloques **try / catch**, de forma similar al código síncrono.

```
async function ejemplo() {
  try {
    const valor = await promesa;
    console.log(valor);
  } catch (error) {
    console.error(error);
  }
};
```

Manejo de errores con try/catch

Con **async / await**, los errores se gestionan utilizando bloques **try / catch**, de forma similar al código síncrono.

```
async function ejemplo() {
  try {
    const valor = await promesa;
    console.log(valor);
  } catch (error) {
    console.error(error);
  }
};
```

Comunicación asíncrona con el servidor

Comunicación asíncrona con el servidor

En aplicaciones web reales, la asincronía se utiliza principalmente para comunicarse con servidores.

Las peticiones al servidor:

- Pueden tardar
- No deben bloquear la interfaz
- Deben realizarse de forma asíncrona

Las aplicaciones web modernas se comunican con servidores mediante **APIs REST**

Estas APIs:

- Utilizan el protocolo HTTP
- Intercambian datos normalmente en formato JSON
- Permiten realizar operaciones CRUD

¿Qué es una API REST?

Una **API REST** es una interfaz que permite a un cliente comunicarse con un servidor para acceder y modificar datos.

REST se basa en:

- El protocolo HTTP
- Recursos identificados por URLs
- Operaciones estándar

Una API REST **no es una tecnología**, es una **forma de organizar la comunicación**

¿Qué es una API REST?

En una API REST:

- Los datos se organizan como **recursos**
- Cada recurso tiene una URL

Ejemplos:

- **/usuarios**
- **/usuarios/5**
- **/pedidos**
- **/productos/12**

El cliente trabaja **sobre recursos**, no sobre bases de datos directamente.

Relación entre HTTP y CRUD

Las APIs REST utilizan los métodos HTTP para realizar operaciones CRUD:

- **GET** → Leer datos (Read)
- **POST** → Crear datos (Create)
- **PUT / PATCH** → Modificar datos (Update)
- **DELETE** → Eliminar datos (Delete)

Esta relación es **fundamental** en aplicaciones web modernas.

Ejemplos:

- GET /usuarios → obtener todos los usuarios
- GET /usuario/3 → obtener un usuario en concreto
- POST /usuarios → crear un usuario
- PUT /usuarios/3 → modificar un usuario en concreto
- DELETE /usuarios/3 → eliminar un usuario

Formato JSON en APIs REST

Las APIs REST intercambian datos normalmente en formato **JSON**

Este formato:

- Es fácil de leer
- Es ligero
- Es independiente del lenguaje
- Es fácil de convertir a objetos JavaScript.

```json
{
  "id": 1,
  "nombre": "Ana",
  "edad": 22
}
```

Comunicación con el servidor: API fetch

Desde JavaScript:

- El cliente realiza peticiones HTTP
- Recibe respuestas del servidor
- Procesa los datos recibidos

Para realizar estas peticiones, JavaScript utiliza **fetch**.

fetch es una API nativa de JavaScript que permite realizar peticiones HTTP desde el navegador. Se caracteriza porque:

- Trabaja de forma asíncrona
- Devuelve promesas
- Se integra perfectamente con **async/await**

Flujo completo de una petición con fetch

1. El cliente realiza una petición con **fetch**
2. El servidor recibe la petición
3. El servidor procesa los datos
4. El servidor devuelve una respuesta
5. El cliente procesa la respuesta
6. La interfaz se actualiza

Todo este proceso se realiza **sin recargar la página**.

Estructura general de una petición fetch

Antes de ver ejemplos concretos, es importante entender la **estructura general** de una llamada a **fetch**.

```
fetch(url, opciones)
```

- **url** → dirección del recurso en el servidor
- **opciones** → objeto que configura la petición (opcional)

Si no se indican opciones, **fetch** realiza una petición **GET** por defecto.

El objeto de opciones de fetch

El segundo parámetro de **fetch** es un **objeto de configuración**, el cual nos permite controlar **cómo** se realiza la petición

```
fetch(url, {
  method: "METHOD",
  headers: {
    "Header": "valor"
  },
  body: datos
});
```

Parámetro *method*

El parámetro **method** indica el tipo de petición HTTP que se quiere realizar.

Los valores más habituales son:

- “**GET**”: obtener datos
- “**POST**”: enviar o crear datos
- “**PUT**”: modificar datos
- “**DELETE**”: eliminar datos

Si no se especifica, el método por defecto es **GET**.

Parámetro *headers*

Los **headers** permiten enviar información adicional al servidor

Se utilizan especialmente cuando:

- Se envían datos
- Se trabaja con JSON
- El servidor necesita saber el formato de los datos

```
headers: {
  "Content-Type": "application/json"
}
```

Parámetro *body*

El parámetro **body** contiene los datos que se envían al servidor.

Características:

- Sólo se usa en POST, PUT o PATCH
- Normalmente se envía en formato JSON
- Debe convertirse con JSON.stringify()

```
body: JSON.stringify({
  nombre: "Ana",
  edad: 22
})
```

¿Qué parámetros se usan según el método HTTP?

Método	method	headers	body
GET	Sí	No (opcional)	No
POST	Sí	Sí	Sí
PUT	Sí	Sí	Sí
DELETE	Sí	Sí (a veces)	No

Ejemplo GET con fetch

```
fetch("https://jsonplaceholder.typicode.com/posts")
  .then(response => response.json())
  .then(posts => {
    console.log(posts);
  })
  .catch(error => {
    console.error(error);
  });
```

Ejemplo GET con async/await

```
async function obtenerPosts() {
  try {
    const response = await fetch("https://jsonplaceholder.typicode.com/posts");
    const posts = await response.json();
    console.log(posts);
  } catch (error) {
    console.error(error);
  }
}

obtenerPosts();
```

Ejemplo POST con fetch

```
fetch("https://jsonplaceholder.typicode.com/posts/1", {
  method: "PUT",
  headers: {
    "Content-Type": "application/json"
  },
  body: JSON.stringify({
    title: "Post actualizado",
    body: "Nuevo contenido",
    userId: 1
  })
})
  .then(response => response.json())
  .then(data => console.log(data));
```

Ejemplo DELETE con fetch

```
fetch("https://jsonplaceholder.typicode.com/posts/1", {
  method: "DELETE"
})
  .then(() => {
    console.log("Recurso eliminado");
  });
```

Gestión de errores en fetch

Es importante entender que:

- **Fetch** sólo falla **por errores de red**
- Errores HTTP (404, 500) **NO lanzan error automáticamente**

Por eso es habitual comprobar lo siguiente:

```
if (!response.ok) {
  throw new Error("Error HTTP");
}
```

Ejemplo completo con control de errores

```
async function cargarUsuarios() {
  try {
    const response = await fetch("https://jsonplaceholder.typicode.com/users");

    if (!response.ok) {
      throw new Error("Error en la respuesta");
    }

    const usuarios = await response.json();
    console.log(usuarios);
  } catch (error) {
    console.error(error);
  }
}
```

AJAX: concepto y evolución

AJAX (Asynchronous JavaScript And XML) es una **técnica** que permite a una aplicación web comunicarse con el servidor de forma asíncrona sin recargar la página. AJAX **no es una tecnología concreta**, sino una forma de trabajar.

Tradicionalmente AJAX se implementaba con:

- XMLHttpRequest
- Código largo y difícil de mantener
- Manejo manual de estados y errores

Este enfoque hoy se considera **obsoleto**. Actualmente, AJAX se implementa utilizando:

- fetch
- Promesas
- async/await

El concepto es el mismo, la sintaxis es más clara y moderna