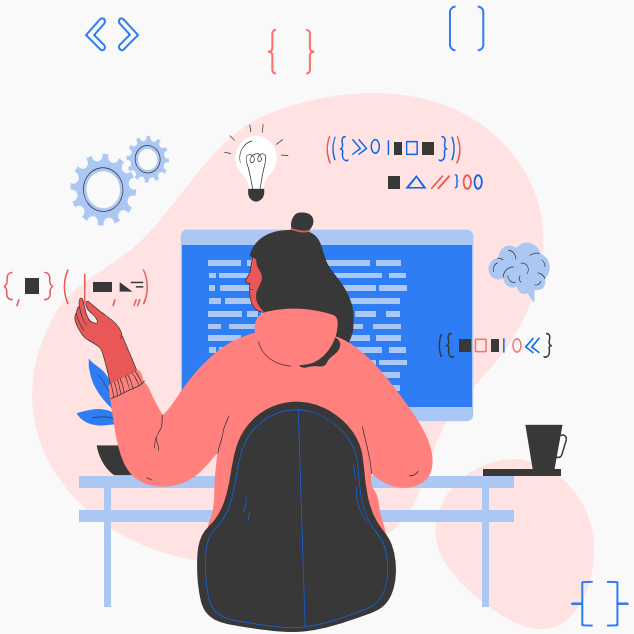
Desarrollo Web en Entorno Cliente

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.

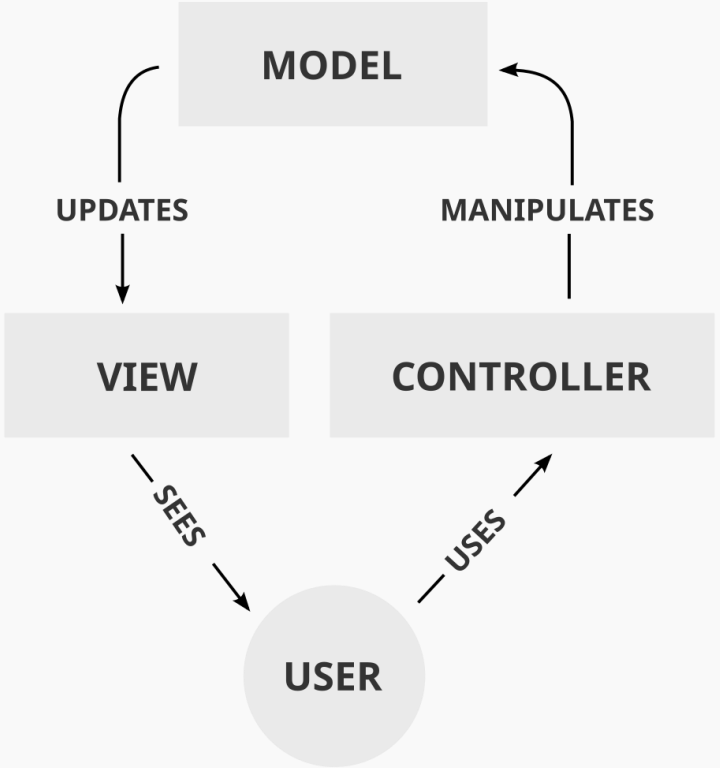
Sus principales responsabilidades son:

- 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:

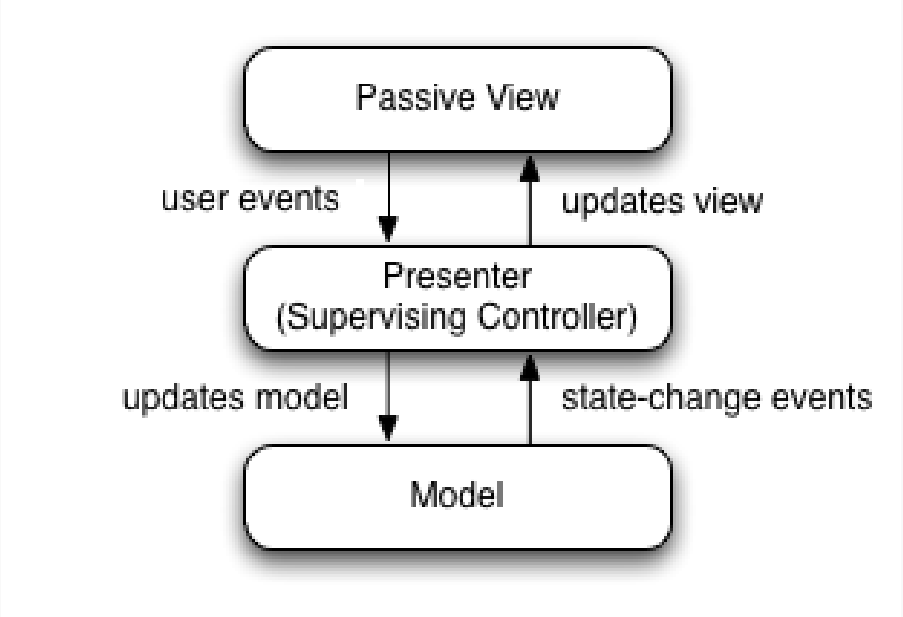
- La Vista es más pasiva
- El Presenter contiene gran parte de la lógica
- Se reduce la dependencia directa entre Vista y Modelo

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 presenter 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,